CONTRIBUTIONS TO PARAMETERIZED COMPLEXITY

BY

CATHERINE MCCARTIN

A THESIS SUBMITTED TO

VICTORIA UNIVERSITY OF WELLINGTON

IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN COMPUTER SCIENCE

WELLINGTON, NEW ZEALAND

SEPTEMBER 2003

To my mother, Carol Richardson.

18 March 1938 - 18 April (Good Friday) 2003

Abstract

This thesis is presented in two parts. In Part One we concentrate on algorithmic aspects of parameterized complexity. We explore ways in which the concepts and algorithmic techniques of parameterized complexity can be fruitfully brought to bear on a (classically) well-studied problem area, that of scheduling problems modelled on partial orderings. We develop efficient and constructive algorithms for parameterized versions of some classically intractable scheduling problems.

We demonstrate how different parameterizations can shatter a classical problem into both tractable and (likely) intractable versions in the parameterized setting; thus providing a roadmap to efficiently computable restrictions of the original problem.

We investigate the effects of using width metrics as restrictions in the setting of online presentations. The online nature of scheduling problems seems to be ubiquitous, and online situations often give rise to input patterns that seem to naturally conform to restricted width metrics. However, so far, these ideas from topological graph theory and parameterized complexity do not seem to have penetrated into the online algorithm community.

Some of the material that we present in Part One has been published in [52] and [77].

In Part Two we are oriented more towards structural aspects of parameterized complexity. Parameterized complexity has, so far, been largely confined to consideration of computational problems as decision or search problems. We introduce a general framework in which one may consider parameterized counting problems, extending the framework developed by Downey and Fellows for decision problems.

As well as introducing basic definitions for tractability and the notion of a parameterized counting reduction, we also define a basic hardness class, $\#W[1]$, the parameterized analog of Valiant's class $\#P$. We characterize $\#W[1]$ by means of a fundamental counting problem, #SHORT TURING MACHINE ACCEPTANCE, which we show to be complete for this class. We also determine $\#W[1]$-completeness, or $\#W[1]$-hardness, for several other parameterized counting problems.

Finally, we present a normalization theorem, reworked from the framework developed by Downey and Fellows for decision problems. characterizing the $\#W[t]$, ($t \in \mathbb{N}$), parameterized counting classes.

Some of the material that we present in Part Two has been published in [78].

Acknowledgements

I am grateful to many people who have helped this thesis along the road to completion.

I owe the largest debt of thanks to Professor Rod Downey, my supervisor, whom I count as both friend and mentor over many years now. It has been an undeserved privilege to work with someone of such world class talent, who has always been willing to go the extra mile in support of his student.

I thank Mike Fellows for many productive discussions, and for his generous and willing help throughout this venture. The work presented in Chapter 4 is joint work with Mike.

I thank Linton Miller for his work on an implementation of the algorithm described in Section 3.2.

I also thank Fiona Alpass, Bruce Donald, Martin Grohe, Mike Hallet, James Noble, Venkatesh Raman, Daniela Rus, and Geoff Whittle, for their helpful advice and support.

I sincerely thank my family; my parents, Norman and Carol Richardson, my children, Mary, Harriet, and Charlie, and especially my husband, Dan McCartin. They have withstood, with good grace, all manner of absences and anxieties on my part. I have received a great deal of help and encouragement from all of them.

Finally, I would like to acknowledge the aid of several fellowships and scholarships which have supported me while I completed graduate studies. During 1999-2001 I received a Victoria University of Wellington Postgraduate Scholarship, as well as Marsden Fund support under the project “Structural Aspects of Computation,” (Downey). In 1990-92, while at Cornell University, I was supported by a John McMullen Graduate Fellowship (Cornell University) and a New Zealand Federation of University Women Postgraduate Fellowship.

Table of contents

List of Figures

Part I

Parameterized Scheduling Problems

CHAPTER 1
INTRODUCTION

In practice, many situations arise where controlling one aspect, or parameter, of the input can significantly lower the computational complexity of a problem. For instance, in database theory, the database is typically huge, say of size n, whereas queries are typically small; the relevant parameter being the size of an input query $k = |\varphi|$. If n is the size of a relational database, and k is the size of the query, then determining whether there are objects described in the database that have the relationship described by the query can be solved trivially in time $O(n^k)$. On the other hand, for some tasks it may be possible to devise an algorithm with running time say $O(2^k n)$. This would be quite acceptable while k is small.

This was the basic insight of Downey and Fellows [46] . They considered, for instance, the following two well-known graph problems:

VERTEX COVER

Instance: A graph $G = (V, E)$ and a positive integer k.

Question: Does G have a vertex cover of size at most k?
(A *vertex cover* is a set of vertices $V' \subseteq V$ such that,
for every edge $uv \in E$, $u \in V'$ or $v \in V'$.)

DOMINATING SET

Instance: A graph $G = (V, E)$ and a positive integer k.

Question: Does G have a dominating set of size at most k?
(A *dominating set* is a set of vertices $V' \subseteq V$ such that,
$\forall u \in V,\ \exists v \in V' \ :\ uv \in E$.)

They observed that, although both problems are NP-complete, the parameter k contributes to the complexity of these two problems in two qualitatively different ways.

They showed that VERTEX COVER is solvable in time $0(2^k k^2 \ +\ kn)$, where

($n = |V|$), for a fixed k [10]. After many rounds of improvement, the current best known algorithm for VERTEX COVER runs in time $O(1.285^k + kn)$ [37]. In contrast, the best known algorithm for DOMINATING SET is still just the brute force algorithm of trying all k-subsets, with running time $O(n^{k+1})$.

The table below shows the contrast between these two kinds of complexity.

	$n = 50$	$n = 100$	$n = 150$
$k = 2$	625	2,500	5,625
$k = 3$	15,625	125,000	421,875
$k = 5$	390,625	6,250,000	31,640,625
$k = 10$	1.9×10^{12}	9.8×10^{14}	3.7×10^{16}
$k = 20$	1.8×10^{26}	9.5×10^{31}	2.1×10^{35}

Table 1.1: The Ratio $\frac{n^{k+1}}{2^k n}$ for Various Values of n and k.

These observations are formalized in the framework of *parameterized complexity theory* [48] . The notion of *fixed-parameter tractability* is the central concept of the theory. Intuitively, a problem is fixed-parameter tractable if we can somehow confine the any "bad" complexity behaviour to some limited aspect of the problem, the parameter.

More formally, we consider a *parameterized language* to be a subset $L \subseteq \Sigma^* \times \Sigma^*$. If L is a parameterized language and $\langle \sigma, k \rangle \in L$ then we refer to σ as the *main part* and k as the *parameter*. A parameterized language, L, is said to be fixed-parameter tractable (FPT) if membership in L can be determined by an algorithm (or a k-indexed collection of algorithms) whose running time on instance $\langle \sigma, k \rangle$ is bounded by $f(k)|\sigma|^\alpha$, where f is an arbitrary function and α is a constant independent of both $|\sigma|$ and the parameter k.

Usually, the parameter k will be a positive integer, but it could be, for instance, a graph or algebraic structure. In this part of the thesis, with no loss of generality, we will identify the domain of the parameter k as the natural numbers $\mathcal{N}$, and consider languages $L \subseteq \Sigma^* \times \mathcal{N}$.

Following naturally from the concept of fixed-parameter tractability are appropriate notions of parameterized problem reduction.

Apparent fixed-parameter *intractability* is established via a completeness program. The main sequence of parameterized complexity classes is

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[t] \cdots \subseteq W[P] \subseteq AW[P] \subseteq XP$$

This sequence is commonly termed the W-hierarchy . The complexity class $W[1]$ is the parameterized analog of NP. The defining complete problem for $W[1]$ is given here.

SHORT NDTM ACCEPTANCE

Instance:	A nondeterministic Turing machine M and a string x.
Parameter:	A positive integer k.
Question:	Does M have a computation path accepting x in $\leq k$ steps?

In the same sense that *NP*-completeness of $q(n)$-STEP NDTM ACCEPTANCE provides us with strong evidence that no *NP*-complete problem is likely to be solvable in polynomial time, $W[1]$-completeness of SHORT NDTM ACCEPTANCE provides us with strong evidence that no $W[1]$-complete problem is likely to be fixed-parameter tractable. It is conjectured that all of the containments here are proper, but all that is currently known is that FPT is a proper subset of XP.

Parameterized complexity theory has been well-developed during the last ten years. It is widely applicable, in part because of hidden parameters such as treewidth, pathwidth, and other graph width metrics, that have been shown to significantly affect the computational complexity of many fundamental problems modelled on graphs.

The aim of this part of the thesis is to explore ways in which the concepts and algorithmic techniques of parameterized complexity can be fruitfully brought to bear on a (classically) well-studied problem area, that of scheduling problems modelled on partial orderings.

We develop efficient and constructive FPT algorithms for parameterized versions of some classically intractable scheduling problems.

We demonstrate how parameterized complexity can be used to "map the boundary of tractability" for problems that are generally intractable in the classical setting. This seems to be one of the more practically useful aspects of parameterized complexity. Different parameterizations can shatter a classical problem into both tractable and (likely) intractable versions in the parameterized setting; thus providing a roadmap to efficiently computable restrictions of the general problem.

Finally, we investigate the effects of using width metrics of graphs as restrictions in the setting of online presentations. The online nature of scheduling problems seems to be ubiquitous, and online situations often give rise to input patterns that seem to naturally conform to restricted width metrics. However, so far, these ideas from topological graph theory and parameterized complexity do not seem to have gained traction in the online algorithm community.

CHAPTER 2
ALGORITHMIC PARAMETERIZED COMPLEXITY

If we compare classical and parameterized complexity it is evident that the framework provided by parameterized complexity theory allows for more finely grained complexity analysis of computational problems. We can consider many different parameterizations of a single classical problem, each of which leads to either a tractable or (likely) intractable version in the parameterized setting.

The many levels of parameterized intractability are prescribed by the W-hierarchy. In Part Two of this thesis we consider parameterized intractability, and the W-hierarchy, more closely. In this part of the thesis we are more concerned with *fixed-parameter tractability*; we want to develop practically efficient algorithms for parameterized versions of scheduling problems that are modelled on partial orderings.

We use the following standard definitions[1].

Definition 2.1 (Fixed Parameter Tractability). *A parameterized language $L \subseteq \Sigma^* \times \Sigma^*$ is fixed-parameter tractable if there is an algorithm that correctly decides, for input $\langle \sigma, k \rangle \in \Sigma^* \times \Sigma^*$, whether $\langle \sigma, k \rangle \in L$ in time $f(k)n^\alpha$, where n is the size of the main part of the input σ, $|\sigma| = n$, k is the parameter, α is a constant (independent of k), and f is an arbitrary function.*

Definition 2.2 (Parameterized Transformation). *A parameterized transformation from a parameterized language L to a parameterized language L' is an algorithm that computes, from input consisting of a pair $\langle \sigma, k \rangle$, a pair $\langle \sigma', k' \rangle$ such that:*

[1]There are (at least) three different possible definitions of fixed-parameter tractability, depending on the level of uniformity desired. These relativize to different definitions of reducibility between problems. The definitions we use here are the standard working definitions. In the context of this thesis, the functions f and g, used in these definitions, will be simple, computable functions.

1. *$\langle \sigma, k \rangle \in L$ if and only if $\langle \sigma', k' \rangle \in L'$,*

2. *$k' = g(k)$ is a function only of k, and*

3. *the computation is accomplished in time $f(k)n^{\alpha}$, where $n = |\sigma|$, α is a constant independent of both n and k, and f is an arbitrary function.*

A collection of distinctive techniques has been developed for FPT algorithm design. Some of these techniques rely on deep mathematical results and give us general algorithms, often non-constructive, pertaining to large classes of problems. Others are simple, yet widely applicable, algorithmic strategies that rely on the particular combinatorics of a single problem. We review here the main techniques and results that have proved useful in the design of FPT algorithms so far.

2.1 Elementary methods

2.1.1 Bounded search trees

Many parameterized problems can be solved by the construction of a search space (usually a search tree) whose size *depends only upon the parameter.* Once we have such a search space, we need to find some efficient algorithm to process each point in the search space. The most often cited application of this technique is for the VERTEX COVER problem.

Consider the following easy algorithm for finding a vertex cover of size k (or determining that none such exists) in a given graph $G = (V, E)$:

We construct a binary tree of height k. We begin by labelling the root of the tree with the empty set and the graph G. Now we pick any edge $uv \in E$. In any vertex cover of G we must have either u or v, in order to cover the edge uv, so we create children of the root node corresponding to these two possibilities. The first child is labelled with $\{u\}$ and $G - u$, the second with $\{v\}$ and $G - v$. The set of vertices labelling a node represents a possible vertex cover, and the graph labelling a node represents what remains to be covered in G. In the case of the first child we have determined that u will be in our possible vertex cover, so we delete u from G, together with all its incident edges, as these are all now covered by a vertex in our possible vertex cover.

In general, for a node labelled with a set S of vertices and subgraph H of G, we arbitrarily choose an edge $uv \in E(H)$ and create the two child nodes labelled, respectively, $S \cup \{u\}$, $H - u$, and $S \cup \{v\}$, $H - v$. At each level in the search tree the size of the vertex sets that label nodes will increase by one. Any node that is labelled with a subgraph having no edges must also be labelled with a vertex set that covers all edges in G. Thus, if we create a node at height at most k in the tree that is labelled with a subgraph having no edges, then a vertex cover of size at most k has been found.

There is no need to explore the tree beyond height k, so this algorithm runs in time $O(2^k n)$, where $n = |V(G)|$.

In many cases, it is possible to significantly improve the $f(k)$, the function of the parameter that contributes exponentially to the running time, by shrinking the search tree. In the case of VERTEX COVER, Balasubramanian et al [11] observed that, if G has no vertex of degree 3 or more, then G consists of a collection of cycles. If G is sufficiently large, then it cannot have a size k vertex cover. Thus, at the expense of an additive constant factor (to be invoked when we encounter any subgraph in the search tree having no degree ≥ 3 vertex), we need consider only graphs containing vertices of degree 3 or greater.

We again construct a binary tree of height k. We begin by labelling the root of the tree with the empty set and the graph G. Now we pick any vertex $v \in V$ of degree 3 or greater. In any vertex cover of G we must have either v or *all of its neighbours*, so we create children of the root node corresponding to these two possibilities. The first child is labelled with $\{v\}$ and $G - v$, the second with $\{w_1, w_2, \ldots, w_p\}$, the neighbours of v, and $G - \{w_1, w_2, \ldots, w_p\}$. In the case of the first child, we are still looking for a size $k - 1$ vertex cover, but in the case of the second child we need only look for a vertex cover of size $k - p$, where $p \geq 3$. Thus, the bound on the size of the search tree is now somewhat smaller than 2^k.

Using a recurrence relation to determine a bound on the number of nodes in this new search tree, it can be shown that this algorithm runs in time $O([5^{1\backslash 4}]^k n)$, where $n = |V(G)|$.

2.1.2 Reduction to a problem kernel

This method relies on reducing a problem instance I to some "equivalent" instance I', where the size of I' is *bounded by some function of the parameter*. Any solution found through exhaustive analysis of I' can be lifted to a solution for I.

For instance, continuing with the VERTEX COVER problem, Sam Buss [29] observed that, for a simple graph G, any vertex of degree greater than k must belong to every k-element vertex cover of G (otherwise all the neighbours of the vertex must be included, and there are more than k of these).

This leads to the following algorithm for finding a vertex cover of size k (or determining that none such exists) in a given graph $G = (V, E)$:

1. Find all vertices in G of degree $\geq k$, let p equal the number of such vertices. If $p > k$, then answer "no". Otherwise, let $k' = k - p$.

2. Discard all p vertices found of degree $\geq k$ and edges incident to these vertices.

3. If the resulting graph G' has more than $k'(k+1)$ vertices, or more than $k'k$ edges, answer "no" (k' vertices of degree at most k can cover at most $k'k$ edges, incident to $k'(k+1)$ vertices).

   Otherwise G' has size bounded by $g(k) = k'(k+1)$, so in time $f(k)$, for some function f, we can exhaustively search G' for a vertex cover of size k', perhaps by trying all k'-size subsets of vertices of G'. Any k'-vertex cover of G' found, plus the p vertices from step 1, constitutes a k-vertex cover of G.

The algorithm given above constructs a problem kernel (G') with size bounded by $k'(k+1)$ (with $k' \leq k$), We note here that Chen *et al* [37] have exploited a well-known theorem of Nemhauser and Trotter [79] to construct a problem kernel for VERTEX COVER having only $\leq 2k$ vertices. This seems to be the best that one could hope for, since a problem kernel of size $(2-\epsilon)k$, with constant $\epsilon > 0$, would imply a factor $2-\epsilon$ polynomial-time approximation algorithm for VERTEX COVER. The existence of such an algorithm is a long-standing open question in the area of approximation algorithms for *NP*-hard problems.

2.1.3 Interleaving

It is often possible to combine the two methods outlined above. For instance, for the VERTEX COVER problem, we can first reduce any instance to a problem kernel and then apply the search tree method to the kernel itself.

Niedermeier and Rossmanith [80] have developed the technique of *interleaving* bounded search trees and kernelization. They show that applying kernelization repeatedly during the course of a search tree algorithm can significantly improve the overall time complexity in many cases.

Suppose we take any fixed-parameter algorithm that satisfies the following conditions: The algorithm works by first reducing an instance to a problem kernel, and then applying a bounded search tree method to the kernel. Reducing any given instance to a problem kernel takes at most $P(|I|)$ steps and results in a kernel of size at most $q(k)$, where both P and q are polynomially bounded. The expansion of a node in the search tree takes $R(|I|)$ steps, where R is also bounded by some polynomial. The size of the search tree is bounded by $O(\alpha^k)$. The overall time complexity of such an algorithm running on instance (I, k) is

$O(P(|I|) + R(q(k))\alpha^k)$.

The idea developed in [80] is basically to apply kernelization at any step of the search tree algorithm where this will result in a significantly smaller problem instance. To expand a node in the search tree labelled by instance (I, k) we first check whether or not $|I| > c \cdot q(k)$, where $c \geq 1$ is a constant whose optimal value will depend on the implementation details of the algorithm. If $|I| > c \cdot q(k)$ then we apply the kernelization procedure to obtain a new instance (I', k'), with $|I'| \leq q(k)$, which is then expanded in place of (I, k). A careful analysis of this approach shows that the overall time complexity is reduced to

$O(P|I| + \alpha^k)$.

This really does make a difference. In [81] the 3-HITTING SET problem is given as an example. An instance (I, k) of this problem can be reduced to a kernel of size k^3 in time $O(|I|)$, and the problem can be solved by employing a search tree of size 2.27^k. Compare a running time of $O(2.27^k \cdot k^3 + |I|)$ (without interleaving) with a running time of $O(2.27^k + |I|)$ (with interleaving).

Note that, although the techniques of *reduction to problem kernel* and *bounded* *search tree* are simple algorithmic strategies, they are not part of the classical toolkit of polynomial-time algorithm design since they both involve costs that are exponential in the parameter.

2.1.4 Color-coding

This technique is useful for problems that involve finding small subgraphs in a graph, such as paths and cycles. Introduced by Alon *et al* in [5] , it can be used to derive seemingly efficient randomized FPT algorithms for several subgraph isomorphism problems. So far, however, in contrast to kernelization and bounded search tree methods, this approach has not lead to any (published) implementations or experimental results.

We formulate a parameterized version of the SUBGRAPH ISOMORPHISM problem as follows:

SUBGRAPH ISOMORPHISM

Instance:	A graph $G = (V, E)$ and a graph $H = (V^H, E^H)$ with $\lvert V^H \rvert = k$.
Parameter:	A positive integer k.
Question:	Is H isomorphic to a subgraph in G?

The idea is that, in order to find the desired set of vertices, V' in G, isomorphic to H, we randomly color all the vertices of G with k colors and expect that, with some high degree of probability, all vertices in V' will obtain different colors. In some special cases of the SUBGRAPH ISOMORPHISM problem, dependent on the nature of H, this will simplify the task of finding V'.

If we color G uniformly at random with k colors, a set of k distinct vertices will obtain different colors with probability $(k!)/k^k$. This probability is lower-bounded by e^{-k}, so we need to repeat the process e^k times to have probability 1 of obtaining the required coloring.

We can de-randomize this kind of algorithm using *hashing* , but at the cost of extending the running time. We need a list of colorings of the vertices in G such that, for *each* subset $V' \subseteq V$ with $\lvert V' \rvert = k$ there is at least one coloring in the list by which all vertices in V' obtain different colors. Formally, we require a k-perfect family of hash functions from $\{1, 2, ..., \lvert V \rvert\}$, the set of vertices in G, onto $\{1, 2, ..., k\}$, the set of colors.

Definition 2.3 (k-Perfect Hash Functions). *A k-perfect family of hash functions is a family $\mathcal{H}$ of functions from $\{1, 2, ..., n\}$ onto $\{1, 2, ..., k\}$ such that, for each $S \subset n$ with $|S| = k$, there exists an $h \in \mathcal{H}$ such that h is bijective when restricted to S.*

By a variety of sophisticated methods, Alon *et al* [5] have proved the following:

Theorem 2.1. *Families of k-perfect hash functions from $\{1, 2, ..., n\}$ onto $\{1, 2, ..., k\}$ can be constructed which consist of $2^{O(k)}$ $\log n$ hash functions. For such a hash function, h, the value $h(i)$, $1 \leq i \leq n$, can be computed in linear time.*

We can color G using each of the hash functions from our k-perfect family in turn. If the desired set of vertices V' exists in G, then, for at least one of these colorings, all vertices in V' will obtain different colors as we require.

We now give a very simple example of this technique. The subgraph that we will look for is a k non-blocker, that is, a set of k vertices that does not induce a *solid neighbourhood* in G. Each of the k vertices in our subgraph must have a neighbour that is *not* included in the subgraph.

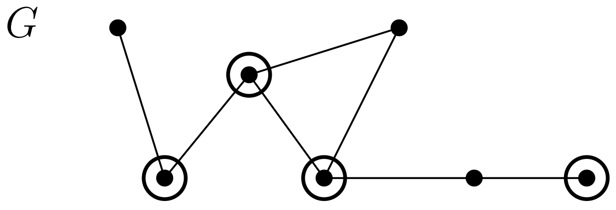


Figure 2.1: Graph G having a size 4 non-blocking set.

k-NON-BLOCKER

Instance: A graph $G = (V, E)$.

Parameter: A positive integer k.

Question: Is there a subgraph in G that is a size k non-blocker?

We use $2k$ colors. If the k non-blocker exists in the graph, there must be a coloring that assigns a different color to each vertex in the non-blocker and a different color to each of the (at most) k *outsider* witness neighbours for the non-blocker. We can assume that our graph G has size at least $2k$, otherwise we can solve the problem by an exhaustive search.

For each coloring, we check every k-subset of colors from the $2k$ possible and decide if it *realizes* the non blocker. That is, for each of the k chosen colors, is there a vertex of this color with a neighbour colored with one of the non-chosen colors? If we answer "yes" for each of the chosen colors, then there must be a set of k vertices each having a neighbour outside the set.

A deterministic algorithm will need to check $2^{O(k)} log|V|$ colorings, and, for each of these, $O(2^{2k})$ k-subset choices. We can decide if a k-subset of colors realizes the non-blocker in time $O(k \cdot |V|^2)$. Thus, our algorithm is FPT, but, arguably, not practically efficient.

Note that this particular problem is trivial, since the answer is "yes" for any $k \leq \lceil |V|/2 \rceil$. A size k non-blocker implies a size $|V| - k$ dominating set, and vice versa, and it is trivial to show that any graph has a size $\lfloor |V|/2 \rfloor$ dominating set.

More interesting examples of applications of color-coding to subgraph isomorphism problems, based on dynamic programming, can be found in [5].

2.2 Methods based on bounded treewidth

Faced with intractable graph problems, many authors have turned to study of various restricted classes of graphs for which such problems can be solved efficiently. A number of graph "width metrics" naturally arise in this context which restrict the inherent complexity of a graph in various senses.

The idea here is that a useful width metric should admit efficient algorithms for many (generally) intractable problems on the class of graphs for which the width is small.

This leads to consideration of these measures from a parameterized point of view. The corresponding naturally parameterized problem has the following form:

Let $w(G)$ denote any measure of graph width.

Instance:	A graph G and a positive integer k.
Parameter:	k.
Question:	Is $w(G) \leq k$?

One of the most successful measures in this context is the notion of *treewidth* which arose from the seminal work of Robertson and Seymour on graph minors and immersions [92]. Treewidth measures, in a precisely defined way, how 'tree-like' a graph is. The fundamental idea is that we can lift many results from trees to graphs that are "tree-like".

Related to treewidth is the notion of *pathwidth* which measures, in the same way, how "path-like" a graph is.

Many generally intractable problems become fixed-parameter tractable for the class of graphs that have bounded treewidth or bounded pathwidth, with the parameter being the treewidth or pathwidth of the input graph. Furthermore, treewidth and pathwidth subsume many graph properties that have been previously mooted, in the sense that tractability for bounded treewidth or bounded pathwidth implies tractability for many other well-studied classes of graphs. For example, planar graphs with radius k have treewidth at most $3k$, series parallel multigraphs have treewidth 2, chordal graphs (graphs having no induced cycles of length 4 or more) with maximum clique size k have treewidth at most $k-1$, graphs with bandwidth at most k (see Section 5.7) have pathwidth at most k.

In this section we review definitions and background information on treewidth and pathwidth for graphs. We also describe related techniques and results that have proved useful in the design of FPT algorithms.

A graph G has treewidth at most k if we can associate a tree T with G in which each node represents a subgraph of G having at most $k+1$ vertices, such that all vertices and edges of G are represented in at least one of the nodes of T, and for each vertex v in G, the nodes of T where v is represented form a subtree of T. Such a tree is called a *tree decomposition* of G, of *width* k.

We give a formal definition here:

Definition 2.4. *[Tree decomposition and Treewidth]*
Let $G=(V,E)$ be a graph. A tree decomposition, TD, of G is a pair $(T,\mathcal{X})$ where

- *$T=(I,F)$ is a tree, and*
- *$\mathcal{X}=\{X_i \mid i \in I\}$ is a family of subsets of V, one for each node of T, such that*
    1. *$\bigcup_{i\in I} X_i = V$,*
    2. *for every edge $\{v,w\} \in E$, there is an $i \in I$ with $v \in X_i$ and $w \in X_i$, and*

3. for all $i, j, k \in I$, if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

The treewidth or width of a tree decomposition $((I, F), \{X_i \mid i \in I\})$ is $max_{i \in I}|X_i| - 1$. The treewidth of a graph G, denoted by $tw(G)$, is the minimum width over all possible tree decompositions of G.

A tree decomposition is usually depicted as a tree in which each node i contains the vertices of X_i.

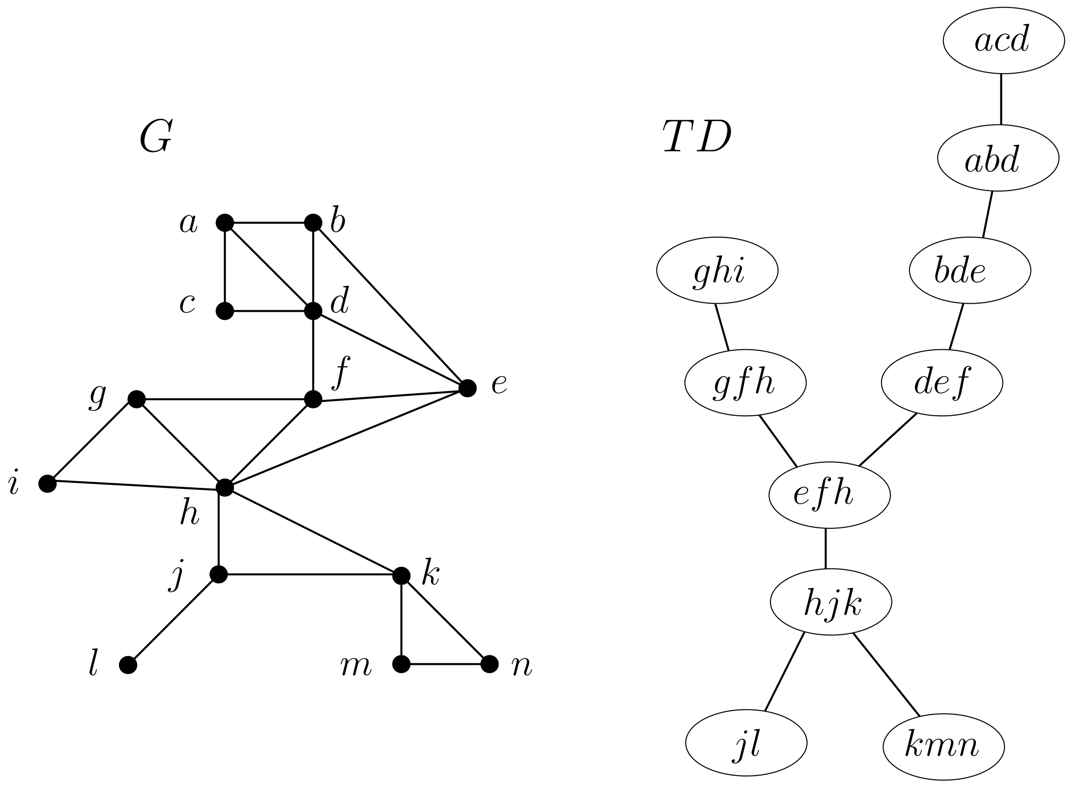


Figure 2.2: Graph G of treewidth 2, and TD, a width 2 tree decomposition of G.

Definition 2.5. *[Path decomposition and Pathwidth]*
A path decomposition, PD, of a graph G is a tree decomposition $(P, \mathcal{X})$ of G where P is simply a path (i.e. the nodes of P have degree at most two). The pathwidth of G, denoted by $pw(G)$ is the minimum width over all possible path decompositions of G.

Let $PD = (P, \mathcal{X})$ be a path decomposition of a graph G with $P = (I, F)$ and $\mathcal{X} = \{X_i \mid i \in I\}$. We can represent PD by the sequence $(X_{i_1}, X_{i_2}, \ldots, X_{i_t})$ where $(i_1, i_2, \ldots, i_t)$ is the path representing P.

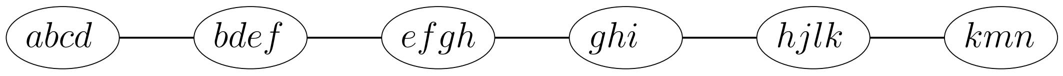


Figure 2.3: A width 3 path decomposition of the graph G.

Any path decomposition of G is also a tree decomposition of G, so the pathwidth of G is at least equal to the treewidth of G. For many graphs, the pathwidth will be somewhat larger than the treewidth. For example, let B_k denote the complete binary tree of height k and order $2^k - 1$, then $tw(B_k) = 1$, but $pw(B_k) = k$.

Graphs of treewidth and pathwidth at most k are also called *partial k-trees* and *partial k-paths* , respectively, as they are exactly the subgraphs of k-trees and k-paths. There are a number of other important variations equivalent to the notions of treewidth and pathwidth (see, e.g., Bodlaender [15]) . For algorithmic purposes, the characterizations provided by the definitions given above tend to be the most useful.

2.2.1 Properties of tree and path decompositions

We include here a number of well-known properties of tree and path decompositions that will be relied upon later in this thesis, in particular in Chapter 5.

Lemma 2.1 (See [15], [53]). *For every graph $G = (V, E)$:*

1. *The treewidth (pathwidth) of any subgraph of G is at most the treewidth (pathwidth) of G.*
2. *The treewidth (pathwidth) of G is the maximum treewidth (pathwidth) over all components of G.*

Lemma 2.2 (Connected subtrees). *Let $G = (V, E)$ be a graph and $TD = (T, \mathcal{X})$ a tree decomposition of G.*

1. *For all $v \in V$, the set of nodes $\{i \in I \mid v \in X_i\}$ forms a connected subtree of T.*
2. *For each connected subgraph G' of G, the nodes in T which contain a vertex of G' induce a connected subtree of T.*

Proof:

1. Immediate from property 3 in the definition of a tree decomposition. If $v \in X_i$ and $v \in X_k$ then if j is on the path from i to k in T, we have $v \in (X_i \cap X_k) \subseteq X_j$.

2. We give the following proof, paraphrased from [53]:

   Claim: Let $u, v \in V$, and let $i, j \in I$ be such that $u \in X_i$ and $v \in X_j$. Then each node on the path from i to j in T contains a vertex of every path from u to v in G.

   Proof of claim: Let $u, v \in V$, and let $P = (u, e_1, e_2, \ldots, v)$ be a path from u to v in G. We use induction on the length of P. If P has length zero, then $u = v$ and the result holds by property 3 of a tree decomposition.

   Suppose P has length one or more. Let $i.j \in I$ be such that $u \in X_i$ and $v \in X_j$. Let P' be a subpath of P from w_1 to v. Let $l \in I$ be such that $u, w_1 \in X_l$. By the induction hypothesis, each node on the path from l to j in T contains a vertex of P'. If i is on the path from l to j in T then we are done. If i is not on the path from l to j, then each node on the path from i to l in T contains u, and hence each node on the path from i to j either contains u or a vertex of P'. □

   Now, suppose that there is a connected subgraph G' of G which does not induce a subtree of T. Then there are nodes $i, j \in I$ such that X_i contains a vertex u of G', X_j contains a vertex v of G', and there is a node l on the path from i to j which does not contain a vertex of G'. However, since there is a path from u to v in G' and hence in G, each node on the path from i to j in T does contain a vertex of G', by the argument given above. This gives a contradiction. □

Consider part 1 of this lemma applied to path decompositions. If $G = (V, E)$ is a graph and $PD = (P, \mathcal{X})$ is a path decomposition of G, then for all $v \in V$, the set of nodes $\{i \in I \mid v \in X_i\}$ forms a connected subpath of P. We call this connected subpath of P the *thread* of v for PD.

Lemma 2.3 (Clique containment). *Let $G = (V, E)$ be a graph and let $TD = (T, \mathcal{X})$ be a tree decomposition of G with $T = (I, F)$ and $\mathcal{X} = \{X_i \mid i \in I\}$. Let $W \subseteq V$ be a clique in G. Then there exists an $i \in I$ with $W \subseteq X_i$.*

Proof: (From [53]) We prove this by induction on $|W|$. If $|W| = 1$, then there is an $i \in I$ with $W \subseteq X_i$ by definition. Suppose $|W| > 1$. Let $v \in W$. By the induction hypothesis there is a node $i \in I$ such that $W - \{v\} \subseteq X_i$. Let $T' = (I', F')$ be the subtree of T induced by the nodes containing v. If $i \in I'$, then $W \subseteq X_i$. Suppose $i \notin I'$. Let $j \in I'$ be such that j is the node of T' that has the shortest distance to i. We show that $W \subseteq X_j$. Let $w \in W - \{v\}$. Each path from a node in T' to node i uses node j. As there is an edge $\{v, w\} \in E$, there is a node $j' \in I'$ such that $v, w \in X_{j'}$. The path from j' to i uses node j and hence $w \in X_j$. $\square$

The following lemma is useful for the design of algorithms on graphs with bounded treewidth or bounded pathwidth.

Lemma 2.4 (Nice tree decomposition). *Suppose the treewidth of a graph $G = (V, E)$ is at most k. G has a tree decomposition $TD = (T, \mathcal{X})$ ($T = (I, F)$, $\mathcal{X} = \{X_i \mid i \in I\}$), of width k, such that a root r of T can be chosen, such that every node $i \in I$ has at most two children in the rooted tree with r as the root, and*

1. *If a node $i \in I$ has two children, j_1 and j_2, then $X_{j_1} = X_{j_2} = X_i$ (i is called a* **join** *node).*

2. *If a node $i \in I$ has one child j, then*
   either $X_i \subset X_j$ and $|X_i| = |X_j| - 1$ (i is called a **forget** *node),*
   or $X_j \subset X_i$ and $|X_j| = |X_i| - 1$ (i is called an **introduce** *node).*

3. *If a node $i \in I$ is a leaf of T, then $|X_i| = 1$ (i is called a* **start** *node).*

4. *$|I| = O(k \cdot |V|)$.*

A tree decomposition of this form is called a *nice* tree decomposition. Similarly, if G is a graph having pathwidth at most k, then G has a nice path decomposition of width k, satisfying conditions 2-4 above.

In [68] it is shown that any given tree (path) decomposition of width k can be transformed into a nice tree (path) decomposition of width k in linear time.

2.2.2 Finding tree and path decompositions

We mentioned above that many intractable problems become fixed-parameter tractable for the class of graphs that have bounded treewidth or bounded pathwidth. A more accurate statement would be to say that many intractable problems become *theoretically* tractable for this class of graphs, in the general case.

The typical method employed to produce FPT algorithms for problems restricted to graphs of bounded treewidth (pathwidth) proceeds in two stages (see [16]) .

1. Find a bounded-width tree (path) decomposition of the input graph that exhibits the underlying tree (path) structure.

2. Perform dynamic programming on this decomposition to solve the problem.

In order for this approach to produce practically efficient algorithms, as opposed to proving that problems are theoretically tractable, it is important to be able to produce the necessary decomposition reasonably efficiently.

Many people have worked on the problem of finding progressively better algorithms for recognition of bounded treewidth graphs, and construction of associated decompositions.

As a first step, Arnborg, Corneil, and Proskurowski [7] showed that if a bound on the treewidth (pathwidth) of the graph is known, then a decomposition that acheives this bound can be found in time $O(n^{k+2})$, where n is the size of the input graph and k is the bound on the treewidth (pathwidth). They also showed that determining the treewidth or pathwidth of a graph in the first place is NP-hard.

Robertson and Seymour [93] gave the first FPT algorithm, $O(n^2)$, for k-TREEWIDTH. Their algorithm, based upon the minor well-quasi-ordering theorem (see [92]), is highly non-constructive, non-elementary, and has huge constants.

The early work of [7, 93] has been improved upon in the work of Lagergren [71], Reed [88], Fellows and Langston [51], Matousek and Thomas [76], Bodlaender [14], and Bodlaender and Kloks [22], among others.

Polynomial time approximation algorithms have been found by Bodlaender, Gilbert, Hafsteinsson, and Kloks [20]. They give a polynomial time algorithm which, given a graph G with treewidth k, will find a tree decomposition of width at most $O(k \log n)$ of G. They also give a polynomial time algorithm which, given a graph G with pathwidth k, will find a path decomposition of width at most $O(k \log^2 n)$.

Parallel algorithms for the constructive version of k-TREEWIDTH are given by Bodlaender [13], Chandrasekharan and Hedetniemi [36], Lagergren [71], and Bodlaender and Hagerup [21].

Bodlaender [14] gave the first linear-time FPT algorithms for the constructive versions of both k-TREEWIDTH and k-PATHWIDTH, although the $f(k)$'s involved mean that treewidth and pathwidth still remain parameters of theoretical interest only, at least in the general case.

Bodlaender's algorithms recursively invoke a linear-time FPT algorithm due to Bodlaender and Kloks [22] which "squeezes" a given width p tree decomposition of a graph G down to a width k tree (path) decomposition of G, if G has treewidth (pathwidth) at most k. A small improvement to the Bodlaender/Kloks algorithm would substantially improve the performance of Bodlaender's algorithms.

Perkovic and Reed [84] have recently improved upon Bodlaender's work, giving a streamlined algorithm for k-TREEWIDTH that recursively invokes the Bodlaender/Kloks algorithm no more than $O(k^2)$ times, while Bodlaender's algorithms may require $O(k^8)$ recursive iterations.

For some graph classes, the optimal treewidth and pathwidth, or good approximations of these, can be found using practically efficient polynomial time algorithms. Examples are chordal bipartite graphs, interval graphs, permuation graphs, circle graphs, [23] and co-graphs [24].

For planar graphs, Alber *et al* [2, 4] have introduced the notion of a *layerwise separation property* pertaining to the underlying parameterized problem that one might hope to solve via a small-width tree decomposition. The *layerwise separation property* holds for all problems on planar graphs for which a linear size problem kernel can be constructed.

The idea here is that, for problems having this property, we can exploit the layer structure of planar graphs, along with knowledge about the structure of "yes" instances of the problem, in order to find small separators in the input graph such that each of the resulting components has small treewidth. Tree decompositions for each of the components are then merged with the separators to produce a small-width tree decomposition of the complete graph.

This approach leads to, for example, algorithms that solve VERTEX COVER and DOMINATING SET on planar graphs in time $2^{O(\sqrt{k})} \cdot n$, where k is the size of the vertex cover, or dominating set, and n is the number of graph vertices. The algorithms work by constructing a tree decomposition of width $O(\sqrt{k})$ for the kernel graph, and then performing dynamic programming on this decomposition.

2.2.3 Dynamic programming on tree decompositions

An algorithm that uses dynamic programming on a *tree* works by computing some value, or table of values, for each node in the tree. The important point is that the value for a node can be computed using only information directly associated with the node itself, along with values already computed for the children of the node.

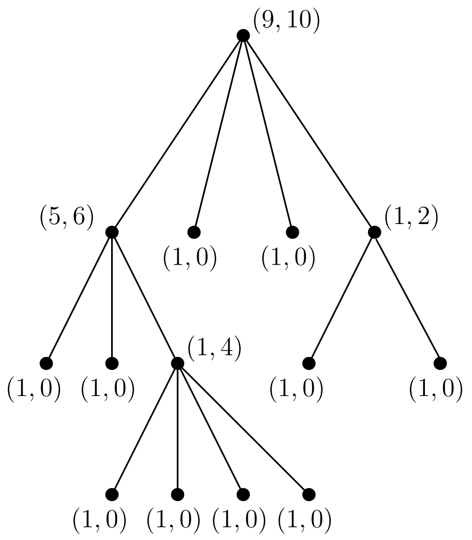
As a simple example of this technique, we consider the MAXIMUM INDEPENDENT SET problem restricted to trees. We want to find a subset of the vertices of a given graph for which there is no edge between any two vertices in the subset, and the subset is as large as possible. This problem is NP-hard in the general case, but if the input graph is a tree, we can solve this problem in time $O(n)$, where n is the number of vertices in the graph, using dynamic programming.

Let T be our input tree, we arbitrarily choose a node (vertex) of T to be the root, r. For each node v in T let T_v denote the subtree of T containing v and all its descendants (relative to r). For each node v in the tree we compute two integers, y_v and n_v, that denote the size of a maximum independent set of T_v that contains v, and the size of a maximum independent set of T_v that doesn't contain v, respectively. It follows that the size of a maximum independent set of T is the maximum of y_r and n_r.

We compute values for each node in T starting with the leaves, which each get the value pair $(y_v = 1, n_v = 0)$. To compute the value pair for an internal node v we need to know the value pairs for each of the children of v. Let $\{c_1, c_2, ..., c_i\}$ denote the children of v. Any independent set containing v cannot contain any of the children of v, so $y_v = n_{c_1} + n_{c_2} + ... + n_{c_i} + 1$. An independent set that doesn't contain v may contain either some, none, or all of the children of v, so $n_v = \max(y_{c_1}, n_{c_1}) + \max(y_{c_2}, n_{c_2}) + ... \max(y_{c_i}, n_{c_i})$.

We work through the tree, computing value pairs level by level, until we reach the root, r, and finally compute $\max(y_r, n_r)$. If we label each node in T with its value pair as we go, we can make a second pass over the tree, in a top-down fashion, and use these values to *construct* a maximum independent set of T.

Extending the idea of dynamic programming on *trees* to dynamic programming on *bounded-width tree decompositions* is really just a matter of having to construct more complicated tables of values. Instead of considering a single vertex at each



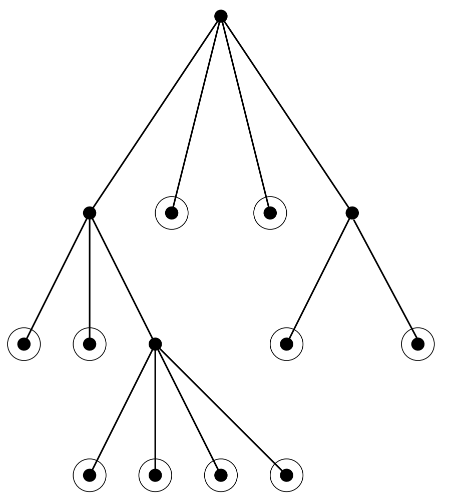


Figure 2.4: A tree with nodes labeled by value pairs (y_v, n_v), and a depiction of the maximum independent set of the tree computed from the value pairs.

node and how it interacts with the vertices at its child nodes, we now need to consider a reasonably small *set* of vertices represented at each node, and how this small set of vertices can interact with each of the small sets of vertices represented at its child nodes.

Suppose that $TD = (T, \mathcal{X})$ is a (rooted) tree decomposition of graph G with width k. For each node i in T let X_i be the set of vertices represented at node i, let T_i be the subtree of T rooted at i, and let V_i be the set of all vertices present in the nodes of T_i. Let G_i be the subgraph of G induced by V_i. We let r denote the root of T. Note that $G_r = G$.

The important property of tree decompositions that we rely on is that G_i is only "connected" to the rest of G via the vertices in X_i. Consider a vertex $v \in V(G)$ such that $v \notin V_i$, and suppose that there is a vertex $u \in V_i$ that is adjacent to v. We know that u is present in node i of T, or some node that is a descendant of node i, and we know that u must be present in some node of T in which v is present, and this cannot be node i or any descendant of node i. The set of nodes in which u is present must form a subtree of T, so it must be the case that u is present in node i and that $u \in X_i$.

Now consider the MAXIMUM INDEPENDENT SET problem restricted to graphs of treewidth k.

We assume that we have a rooted *binary* tree decomposition $TD = (T, \mathcal{X})$ of width k of our input graph G, with $T = (I, F)$ and $\mathcal{X} = \{X_i \mid i \in I\}$. If we have any tree decomposition of width k of G it is an easy matter to produce a rooted binary tree decomposition of width k of G, with $O(n)$ nodes, where n is the number of vertices in G. See, for example, [53] for details.

For each node i in T we compute a table of values S_i. For each set of vertices $Q \subseteq X_i$, we set $S_i(Q)$ to be the size of the largest independent set S in G_i with $S \cap X_i = Q$, we set $S_i(Q)$ to be $-\infty$ if no such independent set exists. The maximum size of any independent set in G will be $\max\{S_r(Q) \mid Q \subseteq X_r\}$, where r is the root of T.

For each leaf node i and each $Q \subseteq X_i$, we set $S_i(Q) = |Q|$ if Q is an independent set in G, and set $S_i(Q) = -\infty$ otherwise.

To compute the table of values for an internal node i we use the table of values for each of the children of i. Let j and l denote the children of node i.

For each $Q \subseteq X_i$, we need to consider all the entries in the tables S_j and S_l for vertex sets $J \subseteq X_j$ where $J \supseteq (Q \cap X_j)$, and $L \subseteq X_l$ where $L \supseteq (Q \cap X_l)$.

Let $\mathcal{J} = \{J \subseteq X_j \mid J \supseteq (Q \cap X_j)\}$, and $\mathcal{L} = \{L \subseteq X_l \mid L \supseteq (Q \cap X_l)\}$.

$$S_i(Q) = \begin{cases} \max\{S_j(J) + S_l(L) - |Q \cap J| - |Q \cap L| + |Q|\ \},\ J \in \mathcal{J}, L \in \mathcal{L}, \\ \qquad \text{if } Q \text{ is an independent set in } G, \text{ and} \\ -\infty, \text{ otherwise.} \end{cases}$$

We work through T, computing tables of values level by level, until we reach the root, r, and finally compute $\max\{S_r(Q) \mid Q \subseteq X_r\}$. We can *construct* a maximum independent set of G by using the information computed in the tables in a top-down fashion.

We need to build $O(n)$ tables, one for each node in the tree decomposition. Each table S_i has size $O(2^{k+1})$. If we assume that all the edge information for each X_i is retained in the representation of the tree decomposition (see [53]), then each entry in the table of a leaf node can be computed in time $O((k + 1) \cdot k)$ and each entry in the table of an internal node can be computed in time $O(2^{3k+3} + ((k + 1) \cdot k))$ from the tables of its children. Thus, the algorithm is linear-time FPT.

The most important factor in dynamic programming on tree decompositions is the size of the tables produced. The table size is usually $O(c^k)$, where k is the width of the tree decomposition and c depends on the combinatorics of the underlying problem that we are trying to solve. We can trade off different factors in the design of such algorithms. For example, a fast approximation algorithm that produces a tree decomposition of width $3k$, or even k^2, for a graph of treewidth k would be quite acceptable if c is small.

2.3 Algorithmic meta-theorems

Descriptive complexity theory relates the logical complexity of a problem description to its computational complexity. In this context there are some useful results that relate to fixed-parameter tractability. We can view these results not as an end in themselves, but as being useful "signposts" in the search for practically efficient fixed-parameter algorithms.

We will consider graph properties that can be defined in *first-order logic* and *monadic second-order logic.*

In first order logic we have an (unlimited) supply of *individual* variables, one for each vertex in the graph. Formulas of first-order logic (FO) are formed by the following rules:

1. **Atomic formulas:** $x = y$ and $R(x_1, ..., x_k)$, where R is a k-ary relation symbol and $x, y, x_1, ..., x_k$ are individual variables, are FO-formulas.
2. **Conjunction, Disjunction:** If ϕ and ψ are FO-formulas, then $\phi \wedge \psi$ is an FO-formula and $\phi \vee \psi$ is an FO-formula.
3. **Negation:** If ϕ is an FO-formula, then $\neg\phi$ is an FO-formula.
4. **Quantification:** If ϕ is an FO-formula and x is an individual variable, then $\exists x\ \phi$ is an FO-formula and $\forall x\ \phi$ is an FO-formula.

We can state that a graph has a clique of size k using an FO-formula,

$$\exists x_1 ... x_k \bigwedge_{1 \leq i \leq j \leq k} E x_i x_j$$

We can state that a graph has a dominating set of size k using an FO-formula,

$$\exists x_1 ... x_k \, \forall y \bigvee_{1 \leq i \leq k} \Big(Ex_i y \, \vee \, (x_i = y) \Big)$$

In monadic second-order logic we have an (unlimited) supply of individual variables, one for each vertex in the graph, and *set* variables, one for each subset of vertices in the graph. Formulas of monadic second-order logic (MSO) are formed by the rules for FO and the following additional rules:

1. **Additional atomic formulas:** For all set variables X and individual variables y, Xy is an MSO-formula.

2. **Set quantification:** If ϕ is an MSO-formula and X is a set variable, then $\exists X \, \phi$ is an MSO-formula, and $\forall X \, \phi$ is an MSO-formula.

We can state that a graph is k-colorable using an MSO-formula,

$$\exists X_1 , , , \exists X_k \left(\forall x \bigvee_{i=1}^{k} X_i x \, \wedge \, \forall x \forall y \Big(Exy \rightarrow \bigwedge_{i=1}^{k} \neg (X_i x \wedge X_i y) \Big) \right)$$

The problems that we are interested in are special cases of the *model-checking problem.*

Let Φ be a class of formulas (logic), and let $\mathcal{D}$ be a class of finite relational structures. The model-checking problem for Φ on $\mathcal{D}$ is the following problem.

Instance: A structure $\mathcal{A} \in \mathcal{D}$, and a sentence (no free variables) $\phi \in \Phi$.

Question: Does $\mathcal{A}$ satisfy ϕ?

The model-checking problems for FO and MSO are PSPACE-complete in general. However, as the following results show, if we restrict the class of input structures then in some cases these model-checking problems become tractable.

The most well-known result, paraphrased here, is due to Courcelle [41].

Theorem 2.2 (Courcelle 1990). *The model-checking problem for MSO restricted to graphs of bounded treewidth is linear-time fixed-parameter tractable.*

Detleef Seese [106] has proved a converse to Courcelle's theorem.

Theorem 2.3 (Seese 1991). *Suppose that $\mathcal{F}$ is any family of graphs for which the model-checking problem for MSO is decidable, then there is a number n such that, for all $G \in \mathcal{F}$, the treewidth of G is less than n.*

Courcelle's theorem tells us that if we can define the problem that we are trying to solve as a model-checking problem, and we can define the particular graph property that we are interested in as an MSO-formula, then there is an FPT algorithm that solves the problem for input graphs of bounded treewidth. The theorem by itself doesn't tell us how the algorithm works.

The automata-theoretic proof of Courcelle's theorem given by Abrahamson and Fellows [1] provides a generic algorithm that relies on dynamic programming over labelled trees. See [48] for extensive details of this approach. However, this generic algorithm is really just further proof of *theoretical* tractability. The importance of the theorem is that it provides a powerful engine for demonstrating that a large class of problems is FPT. If we can couch a problem in the correct manner then we know that it is "worth looking" for an efficient FPT algorithm that works on graphs of bounded treewidth.

The next result concerns graphs of bounded *local treewidth*. The local tree width of a graph G is defined via the following function.

$$ltw(G)(r) = \max \{tw(N_r(v)) \mid v \in V(G)\}$$

where $N_r(v)$ is the neighbourhood of radius r about v.

A graph G has bounded local treewidth if there is a function $f : \mathbb{N} \to \mathbb{N}$ such that, for $r \geq 1$, $ltw(G)(r) \leq f(r)$. The concept is a relaxation of bounded treewidth. Instead of requiring that the treewidth of the graph overall is bounded by some constant, we require that, for each vertex in the graph, the treewidth of each neighbourhood of radius r about that vertex is bounded by some uniform function of r.

Examples of classes of graphs that have bounded local treewidth include graphs of bounded treewidth (naturally), graphs of *bounded degree*, *planar* graphs, and graphs of *bounded genus*.

Frick and Grohe [55] have proved the following theorem.

Theorem 2.4 (Frick and Grohe 1999). *Parameterized problems that can be described as model-checking problems for FO are fixed-parameter tractable on classes of graphs of bounded local treewidth.*

This theorem tells us, for example, that parameterized versions of problems such as DOMINATING SET, INDEPENDENT SET, or SUBGRAPH ISOMORPHISM are FPT on planar graphs, or on graphs of bounded degree. As with Courcelle's theroem, it provides us with a powerful engine for demonstrating that a large class of problems is FPT, but leaves us with the job of finding practically efficient FPT algorithms for these problems.

The last meta-theorem that we will present has a somewhat different flavour. We first need to introduce some ideas from topological graph theory.

A graph H is a *minor* of a graph G iff there exists some subgraph, G^H of G, such that H can be obtained from G^H by a series of *edge contractions*.

We define an edge contraction as follows. Let $e = \{x, y\}$ be an edge of the graph G^H. By G^H/e we denote the graph obtained from G^H by *contracting* the edge e into a new vertex v_e which becomes adjacent to all former neighbours of x and of y. H can be obtained from G^H by a series of edge contractions iff there are graphs $G_0, ..., G_n$ and edges $e_i \in G_i$ such that $G_0 = G^H$, $G_n \simeq H$, and $G_{i+1} = G_i/e_i$ for all $i < n$.

Note that every subgraph of a graph G is also a minor of G. In particular, every graph is its own minor.

A class of graphs $\mathcal{F}$ is *minor-closed* if, for every graph $G \in \mathcal{F}$, every minor of G is also contained in $\mathcal{F}$. A very simple example is the class of graphs with no edges. Another example is the class of acyclic graphs. A more interesting example is the following:

Let us say that a graph $G = (V, E)$ is *within k vertices* of a class of graphs $\mathcal{F}$ if there is a set $V' \subseteq V$, with $|V'| \leq k$, such that $G[V - V'] \in \mathcal{F}$. If $\mathcal{F}$ is any minor-closed class of graphs, then, for every $k \geq 1$, the class of graphs within k vertices of $\mathcal{F}$, $\mathcal{W}_k(\mathcal{F})$, is also minor-closed.

Note that for each integer $k \geq 1$, the class of graphs of treewidth or pathwidth at most k is minor-closed.

We say that a partial ordering $\preceq$ on graphs is a *well partial order* iff, given any infinite sequence $G_1, G_2, \ldots$ of graphs, there is some $i \leq j$ such that $G_i \preceq G_j$.

In a long series of papers, collectively entitled "Graph Minors", almost all appearing in the *Journal of Combinatorial Theory B*, Robertson and Seymour [90–105] have proved the following:

Theorem 2.5 (Robertson and Seymour). *Finite graphs are well partial ordered by the minor relation.*

Let $\mathcal{F}$ be a class of graphs which is closed under taking of minors, and let H be a graph that is not in $\mathcal{F}$. Each graph G which has H as a minor is not in $\mathcal{F}$, otherwise H would be in $\mathcal{F}$. We call H a *forbidden minor* of $\mathcal{F}$. A *minimal* forbidden minor of $\mathcal{F}$ is a forbidden minor of $\mathcal{F}$ for which each proper minor is in $\mathcal{F}$. The set of all minimal forbidden minors of $\mathcal{F}$ is called the *obstruction set* of $\mathcal{F}$.

Theorem 2.5 implies that any minor-closed class of graphs $\mathcal{F}$ must have a *finite* obstruction set. Robertson and Seymour have also shown that, for a fixed graph H, it can be determined whether H is a minor of a graph G in time $O(f(|H|) \cdot |G|^3)$.

We can now derive the following theorem:

Theorem 2.6 (Minor-closed membership). *If $\mathcal{F}$ is a minor-closed class of graphs then membership of a graph G in $\mathcal{F}$ can be determined in time $O(f(k) \cdot |G|^3)$, where k is the collective size of the graphs in the obstruction set for $\mathcal{F}$.*

This meta-theorem tells us that if we can define a graph problem via membership in a minor-closed class of graphs $\mathcal{F}$, then the problem is FPT, with the parameter being the collective size of the graphs in the obstruction set for $\mathcal{F}$. However, it is important to note two major difficulties that we now face. Firstly, for a given minor-closed class $\mathcal{F}$, theorem 2.5 proves the *existence* of a finite obstruction set, but provides no method for *obtaining* the obstruction set. Secondly, the minor testing algorithm has very large hidden constants (around 2^{500}), and the sizes of obstruction sets in many cases are known to be very large.

Concerted efforts have been made to find obstruction sets for various graph classes. The obstruction sets for graphs with treewidth at most one, two and three [8], and for graphs with pathwidth at most one and two [67], are known, as well as obstruction sets for a number of other graph classes (see [44]). Fellows and Langston [51], Lagergren and Arnborg [72], and, more recently, Cattell *et al* [32] have considered the question of what information about a minor-closed graph class $\mathcal{F}$ is required for the effective computation of the obstruction set for $\mathcal{F}$.

So, finally, here again we have a theorem that provides us with a powerful engine for demonstrating that a large class of problems is FPT, but the job of finding practically efficient FPT algorithms for such problems certainly remains.

CHAPTER 3
FPT ALGORITHMS

3.1 Introduction

In this chapter we present FPT algorithms for two scheduling problems. The first of these is the "jump number problem". This problem is included in the "lineup of tough customers" given in [46]. At the time of publication, these problems had resisted strenuous efforts to show that they were hard for $W[1]$, yet seemed likely to be intractable. Here, we show that the jump number problem is, in fact, fixed parameter tractable and give a linear-time constructive algorithm that solves this problem[1]. The material that we present has been published in [77]. Our algorithm is quite straightforward, and is based on the construction of a bounded search tree. Recently, Grohe [59] has shown that another of these tough customers, the crossing number problem, is also fixed parameter tractable. He gives a quadratic-time constructive algorithm to solve that problem which depends heavily on the structure theory for graphs with excluded minors and algorithmic ideas developed in that context.

Our second problem is a parameterized version of the "linear extension count" problem. In the second part of this thesis we build a general framework in which to consider parameterized counting problems, which extends the framework for parameterized decision problems developed by Downey and Fellows. We are concerned there with the insights that parameterized complexity can provide about the enumeration of small substructures at a general problem level, where the parameter of interest corresponds to the nature of the structures to be counted. The linear extension count problem that we consider here is an example from a class of related, but

[1]A working version of the algorithm described in this section has been implemented in C by Linton Miller, Victoria University of Wellington, New Zealand. The C-code is available at `http://www-ist.massey.ac.nz/mccartin`

seemingly much simpler, decision problems, where we parameterize on the number of structures that we hope to find.

3.2 Jump number

Suppose a single machine performs a set of jobs, one at a time, with a set of precedence constraints prohibiting the start of certain jobs until some others are already completed. Any job which is performed immediately after a job which is not constrained to precede it requires a "jump", which may entail some fixed additional cost. The scheduling problem is to construct a schedule to minimize the number of jumps. This is known as the "jump number problem".

We can model the jobs and their precedence constraints as a finite ordered set (partial ordering). A schedule is a *linear extension* of this ordered set.

A *linear extension* $L = (P, \preceq)$ of a finite ordered set $(P, \leq)$ is a total ordering of the elements of P in which $a \prec b$ in L whenever $a < b$ in P. L is the linear sum $L = C_1 \oplus C_2 \oplus \cdots C_m$ of disjoint chains $C_1, C_2, \ldots, C_m$ in P, whose union is all of P, such that $x \in C_i, y \in C_j$ and $x < y$ in P implies $i \leq j$. If the maximum element of C_i is incomparable with the minimum element of C_{i+1} then $(\max C_i, \min C_{i+1})$ is a jump in L. The number of jumps in L is denoted by $s_L(P)$ and the *jump number* of P is

$$s(P) = min\{s_L(P) : L \text{ is a linear extension of } P\}$$

By Dilworth's theorem [43], the minimum number of chains which form a partition of P is equal to the size of a maximum antichain. This number is the *width* $w(P)$ of P. Thus $s(P) \geq w(P) - 1$.

Pulleyblank [85], and also Bouchitte and Habib [25], have proved that the problem of determining the jump number $s(P)$ of any given ordered set P is NP-complete. There exist polynomial-time algorithms for some special classes of ordered sets defined by forbidden substructures eg. N-free (see [89]), cycle-free (see [49]), or restricted by bounding some of their parameters eg. width (see [39]), number of dummy arcs in the arc representation (see [109]).

El-Zahar and Schmerl [50] have shown that the decision problem whether $s(P) \leq k$, where k is an integer fixed through all instances of the problem, is solvable in polynomial time. However, their approach involves finding $s(Q)$ for each $Q \subseteq P$ with $|Q| \leq (k+2)!$. There are in the order of $|P|^{(k+2)!}/(k+2)!$ such $Q \subseteq P$, hence the algorithm has a running time in excess of $O(n^{k!})$, where $n = |P|$ and k is a fixed parameter bounding $s(P)$. Note that the running time, while polynomial, is exceedingly impractical even for small k. Furthermore, note that while the algorithm is polynomial time, the exponent of $|P|$ increases as k increases. This is also the case for the algorithm given in [39], where the exponent of $|P|$ increases as the width parameter, $w(P)$, increases.

In this section we explore the following problem.

JUMP NUMBER

Instance:	A finite ordered set P, and a positive integer k.
Parameter:	k
Question:	Is $s(P) \leq k$? If so, output $L(P)$ with $s_L(P) \leq k$.

Our goal is to "get k out of the $|P|$ exponent", and thus produce an FPT algorithm. We give a constructive algorithm that outputs $L(P)$ with $s_L(P) \leq k$ if the answer is *yes*, and outputs *no* otherwise, and that runs in time $O(k^2 k! |P|)$.

Such an algorithm is of some interest, since many instances of the problem have a relatively small number of precedence constraints.

3.2.1 Structural properties of the ordered set

We begin by noting that if $s(P) \leq k$ then no element of P covers more than $k+1$ elements, or is covered by more than $k+1$ elements, as this would imply a set of incomparable elements of size $> k+1$. Thus, if we represent P as a Hasse diagram, then no element of P has indegree or outdegree $> k+1$.

For any algorithm solving a graph problem, the input graph must be stored in memory. There are many ways to represent graphs in memory. If the parameter k is small compared to $|P|$, then the incidence matrix for P will be sparse. In order to take advantage of the fact that P has both indegree and outdegree bounded by $k+1$ we make use of an *adjacency list representation.* We associate with each element of P an inlist and an outlist, each of length $\leq k+1$. The inlist of an element a contains pointers to elements covered by a. The outlist of a contains pointers to elements which cover a. It is most efficient to use double pointers when implementing these lists, so each list entry consists of a pointer to an element, and a pointer from that element back to the list entry.

Given an incidence matrix for P we could produce an adjacency list representation in $O(n^2)$ time, where $n = |P|$. Instead, we assume that P is input directly to our algorithm using an adjacency list representation, in the manner of [53].

Note that the adjacency list representation implicitly gives us an $O(kn)$ problem kernel, where $n = |P|$.

k=1 case, a skewed ladder

If $k = 1$, then we are asking "is $s(P) \leq 1$?". A *yes* answer means the structure of P is very restricted.

If the answer is *yes*, then P must be able to be decomposed into at most two disjoint chains, C_1 and C_2, and any *links* between these chains must create only a *skewed ladder*, i.e. any element $a \in C_1$ comparable to any element $b \in C_2$ must obey the property $a < b$.

A width k+1 skewed ladder

Suppose we have a fixed k and find that $w(P) = k + 1$. Then, if we obtain a *yes* answer to the question "Is $s(P) \leq k$?", the structure of P is again very restricted.

If the answer is *yes*, then P must be able to be decomposed into a width $k + 1$ skewed ladder, i.e. $k + 1$ disjoint chains such that these chains $C_1, C_2, \ldots, C_{k+1}$ can be ordered so that:

1. any element $a \in C_i$ comparable to any element $b \in C_j$ where $i < j$ must obey the property $a < b$.
2. $w(P_i) = i$, where $P_i = \bigcup\{C_1, \ldots, C_i\}$.

3.2.2 An FPT algorithm

For any $a \in P$ let $D(a) = \{x \in P : x \leq a\}$. We say that a is *accessible* in P if $D(a)$ is a chain. An element a is *maximally accessible* if $a < b$ implies b is not accessible.

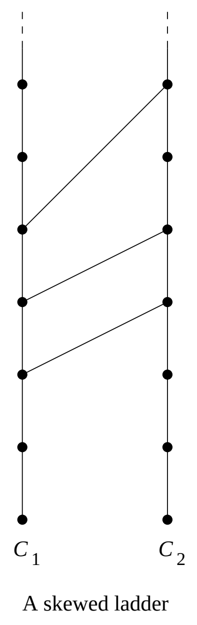


Figure 3.1: A skewed ladder.

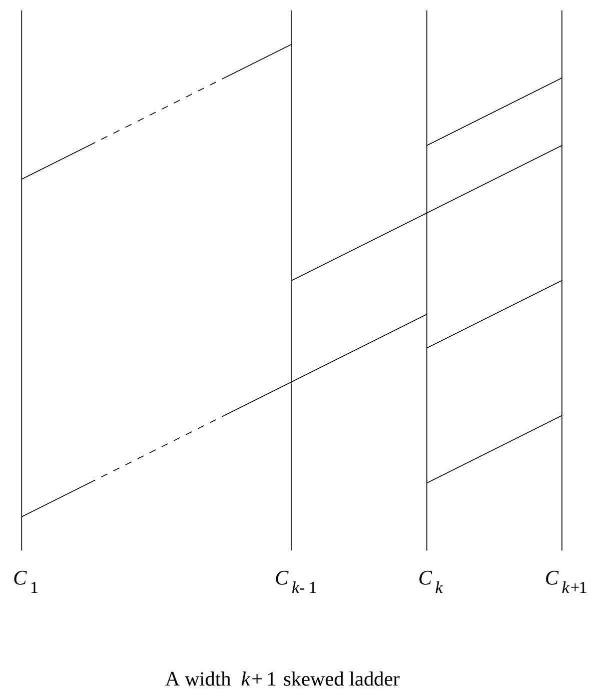


Figure 3.2: A width $k+1$ skewed ladder.

In any linear extension, $L = C_1 \oplus C_2 \oplus \cdots \oplus C_m$ of P, each $a \in C_i$ is accessible in $\bigcup_{j \geq i} C_j$. In particular, C_1 contains only elements accessible in P. If C_1 does not contain a maximally accessible element we can transform L to $L' = C_1' \oplus C_2' \oplus \cdots \oplus C_m'$ where C_1' does contain a maximally accessible element, without increasing the number of jumps [108].

We use the fact that $s(P) \leq k$ iff $s(P - D(a)) \leq k - 1$ for some maximally accessible element $a \in P$.

Let us prove this fact. Suppose $s(P) \leq k$. Then we have $L(P) = C_1 \oplus C_2 \oplus \cdots \oplus C_m$ where $s_L(P) \leq k$ and C_1 contains a maximally accessible element, say a. $L' = C_2 \oplus \cdots \oplus C_m$ is a linear extension of $P - D(a)$ with $s_{L'}(P - D(a)) \leq k - 1$. Suppose $s(P - D(a)) \leq k - 1$ for some maximally accessible element $a \in P$, then we have $L(P - D(a)) = C_1 \oplus \cdots \oplus C_m$ where $s_L(P - D(a)) \leq k - 1$. $L' = D(a) \oplus C_1 \oplus \cdots \oplus C_m$ is a linear extension of P with $s_{L'}(P) \leq k$. $\square$

JUMP NUMBER

Instance: A finite ordered set P, and a positive integer k.

Parameter: k

Question: Is $s(P) \leq k$? If so, output $L(P)$ with $s_L(P) \leq k$.

We build a search tree of height at most k.

- Label the root of the tree with P.
- Find all maximally accessible elements in P. If there are more than $k + 1$ of these, then $w(P) > k + 1$ and the answer is *no*, so let's assume that there are $\leq k + 1$ maximally accessible elements.
- If there are exactly $k + 1$ maximally accessible elements, then P has width at least $k + 1$ so check whether P has a width $k + 1$ skewed ladder decomposition. If so, answer *yes* and output $L(P)$ as shown in the text following. If not, answer *no*.
- If there are $< k + 1$ maximally accessible elements, then for each of these elements $a_1, a_2, \ldots, a_m$ produce a child node labelled with $(D(a_i), P_i = P - D(a_i))$ and ask "Does P_i have jump number $k - 1$?" If so, we output $L(P) = D(a_i) \oplus L(P_i)$.

3.2.3 Algorithm analysis

The search tree built by the algorithm will have height at most k, since at each level we reduce the parameter value by 1.

The branching at each node is bounded by the current parameter value, since we only branch when we have strictly less than (parameter + 1) maximally accessible elements to choose from.

Thus, the number of paths in the search tree is bounded by $k!$, and each path may contain at most k steps.

At each step we must find all maximally accessible elements in P, and either branch and produce a child node for each, or recognize and output the skewed ladder decomposition of the current correct width.

Finding all maximally accessible elements can be done with $O((k+1)n)$ computations, when the parameter is k, as shown below.

To produce a child node corresponding to some maximally accessible element a_i, labelled $(D(a_i), P_i = P - D(a_i))$, requires $O((k+1)n)$ computations, when the parameter is k, as shown below.

To recognize and output the skewed ladder decomposition of width say m, requires at most $O((m(m+1)/2)n)$ computations, as shown below.

On each path, we will, at some point, need to recognize a skewed ladder. Suppose this occurs after q iterations of the "find all maximally accessible elements" phase, where the parameter decreases by 1 for each of these iterations. The skewed ladder to be recognized will have width $(k+1)-(q-1)$.

The first q steps will require $O((k+1)n)$, $O(((k+1)-1)n)$, $\ldots O(((k+1)-(q-1))n)$ computations respectively. To recognize the skewed ladder will require $O((m(m+1)/2)n)$ computations, where $m = (k+1)-(q-1)$. Therefore, this path will require $O(((k+1)-(q-1))n + ((k+1)(k+2)/2)n)$ computations.

Therefore, no path requires more than $O((k+1)n + ((k+1)(k+2)/2)n)$ computations.

Thus, the running time of the algorithm is $O(k^2 k! n)$.

Finding all maximally accessible elements

We input P to the algorithm as a list of elements, where we associate with each element an inlist and an outlist. The inlist of a contains pointers to elements covered by a. The outlist of a contains pointers to elements which cover a.

As mentioned earlier, if we take the current parameter to be k, then no element of P can have indegree or outdegree $> k+1$, since otherwise we can immediately answer *no*. Therefore, we will access at most $k+1$ elements in any inlist or outlist, and output *no* if further elements are present.

We begin by finding all minimal elements of P by running through the list of elements to find all those whose inlist is empty.

For each of these elements we create a chain structure, which to begin with is just the element itself, say a. We copy this chain structure for each element in the outlist of a that is accessible (i.e.whose inlist contains only a), and append the new element to its copy of a to create a new chain. We recursively apply this process to each of the new elements to build up at most $k+1$ chains, each ending in a maximally accessible element.

We access each outlist at most once, each inlist at most twice, and copy any element at most $k+1$ times. Therefore, for parameter k, we can find all maximally accessible elements in time $O((k+1)n)$.

Producing child nodes

If the current parameter is k and we find at most k maximally accessible elements, we branch and produce a child node for each maximally accessible element a_i, labelled with the chain $D(a_i)$, and a new ordered set $P_i = P - D(a_i)$.

To produce $P_i = P - D(a_i)$, we take a copy of P and alter it by removing $D(a_i)$ and all links to it. If we implement the inlists and outlists using double pointers, so that each list entry consists of a pointer to an element, and a pointer from that element back to the list entry, then we can do this in time $O((k+1)n)$. Each element of $D(a_i)$ has its inlist and outlist traversed to remove links that occur in the lists of other elements, then the element itself is removed.

Recognizing a skewed ladder

We consider the $k+1$ maximally accessible elements which force the skewed ladder recognition.

If P can indeed be decomposed into a width $k+1$ skewed ladder, then at least one of these elements must be associated with a *top* chain, i.e. a maximal chain in which no element is below an element in any other chain, and whose removal leaves a skewed ladder of width k. Thus, for at least one of the maximally accessible elements it must be the case that all the elements above it have outdegree ≤ 1.

We build a chain structure recursively above each of the $k+1$ maximally accessible elements, as described above, discarding it if we find an element with outdegree > 1, and marking it as a *possible top* chain if we reach a maximal element (having outdegree 0).

We then check, in turn, below each maximally accessible element that is the start of a *possible top* chain.

We add to the chain recursively downwards, by looking at any element in the inlist of the current element (there will be at most 1), and adding it to the chain if it has outdegree 1. If we reach an element that has outdegree > 1 we do not include it. Instead we need to ensure that this element lies below one of the remaining maximally accessible elements, since this element and all its predecessors will have to be incorporated into a width k skewed ladder containing all the remaining k maximally accessible elements (we obtain the necessary information by keeping track of the number of copies made of each element during the "find all maximally accessible elements" phase).

If we reach a minimal element, or an acceptable element of outdegree > 1, we remove the *top* chain found and all links to it, and then check that the remaining elements form a skewed ladder of width k.

We do $O((k{+}1)n)$ computations to find a *top* chain of a width $k{+}1$ skewed ladder, since we look at each element and its inlist and outlist at most once. However, each time we remove a *top* chain, the width parameter is reduced by 1, so the maximum number of elements accessed in any inlist or outlist is also reduced by 1.

Thus, we can recognize a skewed ladder of width $k+1$ in time
$O(((k+1)(k+2)/2)n)$.

3.2.4 Further improvements

Now we have a constructive *FPT* algorithm that runs in in time $O(k^2 k! n)$, further efforts should concentrate on pruning the search tree in order to lower the $k!$ factor in the running time.

For example, when we consider the set of maximally accessible elements at each stage, we can first choose any element a that is maximal in P, since in this case $D(a)$ must be a maximal chain, all of whose elements are below, or incomparable to, all others. Conversely, we should not choose a non-maximal element a where $D(a)$ is an isolated chain, as such a chain can be incorporated at a later stage without increasing the jump number.

In [110] Syslo introduces *strongly greedy chains* which have the following property: if an ordered set P contains a strongly greedy chain C, then C may be taken as the first chain in an optimal linear extension of P. However, not every ordered set contains a strongly greedy chain.

$D(a)$ is strongly greedy if either a is maximal in P, or there exists $q \in P$, $q \neq a$, such that q and a are covered by the same set of elements and the set $Q(q) = D(q) \cup \{r : r \text{ covers an element below } q\}$ is *N-free* in P. Thus, the running time to test whether $D(a)$ is strongly greedy is dominated by $|Q(q)|$, if such a candidate element q is present in P. It might be useful to perform such a test in cases where $|Q(q)|$ is sufficiently small.

These heuristics can be applied in specific cases, when these arise. However, there may be strategies which can be applied generally to produce a more sparse search tree.

3.3 Linear extension count

As mentioned in the introduction to this chapter, in the second part of this thesis we develop a general framework in which to consider parameterized counting problems. There, we are considering functions, where the input is a parameterized problem instance and the output is a natural number. We are concerned with the complete enumeration of small substructures, and the parameter of interest corresponds to the nature of the structures to be counted. The linear extension count problem that we consider in this section is a “counting flavoured” decision problem, where we parameterize on the number of structures that we hope to find.

We give a simple linear-time constructive FPT algorithm, again based on a bounded search tree, for the following problem.

LINEAR EXTENSION COUNT

Instance:	A finite ordered set P, and a positive integer k.
Parameter:	k
Question:	Is $\#L(P)$, the number of distinct linear extensions of P, $\leq k$? If so, output all the distinct linear extensions of P.

The classical (unparameterized) counting version of this problem, where the input is a finite ordered set P, and the output is the total number of distinct linear extensions of P, has been shown to be $\#P$-complete [27]. The parameterized version of the problem that we consider here has been shown to be in randomized FPT [28], but was not known to be in FPT [48].

Note that we can combine this kind of restricted counting parameterization with the usual kind of parameterization for decision problems. For example, the “k-jump linear extension count problem” takes as input a finite ordered set P and positive integers k and k', and decides whether or not the number of distinct linear extensions of P, that each have at most k jumps, is $\leq k'$. The “k-width tree decomposition count problem” takes as input a graph G and positive integers k and k', and decides whether or not the number of distinct k-width tree decompositions of G is $\leq k'$.

In both these cases, the base problem is in the class FPT. We have an FPT algorithm to decide whether or not there is at least one linear extension with at most k jumps for a given finite ordered set P. As mentioned in Section 2.2, there is a linear-time FPT algorithm, due to Bodlaender [14], that decides whether or not there is at least one tree decomposition, of width at most k, for a given graph G. However, it is certainly not clear whether either of these counting flavoured decision problems is in FPT.

In both cases, the FPT algorithms for the base problems bound the search space by means of considering only a single representative from each of a small (parameter-dependent) number of equivalence classes of partial solutions. In the case of jump number, the equivalence classes of partial solutions are represented by maximally accessible elements. It is the unravelling of these equivalence classes, to account for *all* possible solutions, that confounds us when we try to decide whether there are at most k of these.

3.3.1 An FPT algorithm

We restate here the problem that we are trying to solve.

LINEAR EXTENSION COUNT

Instance:	A finite ordered set P, and a positive integer k.
Parameter:	k
Question:	Is $\#L(P)$, the number of distinct linear extensions of $P \leq k$? If so, output all the distinct linear extensions of P.

We build a bounded size search tree, this time by bounding the total number of branches in the tree to be at most k. Each branch has length at most $|P|$.

We begin by noting that if $\#L(P) \leq k$ then no element of P covers more than k elements, or is covered by more than k elements, as this would imply a set of incomparable elements of size $> k$. These elements could be ordered in any of $k!$ ways in a linear extension of P, thus producing $k!$ distinct linear extensions. This makes it clear that we can be far more severe with this requirement, however, for the sake of simplicity, we will make do with a bound of k here.

This will allow us to input P to the algorithm as an adjacency list representation where we associate with each element an inlist and an outlist, this time each of length $\leq k$. The inlist of an element a contains pointers to elements covered by a. The outlist of a contains pointers to elements which cover a.

As for the jump number problem, the adjacency list representation implicitly gives us an $O(kn)$ problem kernel, where $n = |P|$.

We build a search tree with at most k branches.

- Label the root of the tree with P and $\emptyset$. Set #leaves = 1.
- Choose a leaf l in the current tree:
    1. Let P^l be the finite ordered set that labels l. Find all minimal elements in P^l. Let m be the number of minimal elements found.

2. If (#leaves $+ m - 1) > k$ then answer *no*.
3. Otherwise, for each of the minimal elements of P^l, $a_1, a_2, \ldots, a_m$, produce a child node labelled with $P^l_i = P^l - a_i$ and a_i. Set #leaves $=$ (#leaves $+ m - 1$).

- Continue expanding leaves in the tree until either we answer *no* at step 2 of some expansion, or all leaves in the tree are labelled with $\emptyset$ (the empty finite ordered set) and some element a of P.
- If we reach the point where all leaves in the tree are labelled with $\emptyset$ and we have not already answered *no*, then #leaves gives us the number of distinct linear extensions of P, and we can read these off from the $\leq k$ branches each of length $|P|$ that we have built.

3.3.2 Algorithm analysis

The tree that we build will have at most $kn + 1$ nodes.

At the root node, we can find all minimal elements in the original ordered set P in time $O(n)$, by running through the adjacency list of elements to find all those whose inlist is empty. We allow at most k of these and, for each, we produce a child node labelled with a new ordered set.

Suppose that we have a node l, labelled with an ordered set P^l, and that we have a list of minimal elements of P^l. We produce each of the (at most k) child nodes of l as follows:

Let a be the minimal element that we need to remove from P^l. To produce an adjacency list representing $P^l - a$ we alter the adjacency list of P^l. The element a has its outlist traversed to remove links that occur in the inlists of other elements, then a itself is removed. At the same time we update the list of minimal elements of P^l to get a new list of minimal elements for $P^l - a$. Each of the elements in the outlist of a will have a removed from its inlist. If this leaves an empty inlist, then the element is added to the list of minimal elements for $P^l - a$. Any minimal elements of P^l, except a, are also put in the list of minimal elements for $P^l - a$.

If we implement the inlists and outlists using double pointers, so that each entry consists of a pointer to an element, and a pointer from that element back to the list entry, then we can alter the old adjacency list to represent the new ordered set, and produce the list of minimal elements for that ordered set, in time $O(k)$. We use the length of the list of minimal elements to update #leaves.

Thus, the root node can be produced in time $O(n)$, and each of the kn non-root nodes can be produced in time $O(k)$, by altering the information labelling the parent node.

For some nodes, it may be the case that we need more than one copy of the information labelling the parent node. However, note that there are at most k leaves existing at all stages of building the tree, so no more than k distinct adjacency lists will be needed at any stage.

Thus, the running time of the algorithm is bounded by $O(k^2 \cdot n)$.

CHAPTER 4
MAPPING THE TRACTABILITY BOUNDARY

4.1 Introduction

In this chapter we demonstrate how parameterized complexity techniques can be used to "map the boundary" between feasible and intractable parameterizations of a given classical problem. The work that we present is joint work with Mike Fellows, published in [52].

We can consider many different parameterizations of a single classical problem, each of which leads to either a tractable or (likely) intractable version in the parameterized setting. As we show here, restrictions to a problem that have no bearing from the point of view of NP-completeness can have a strong affect on the problem complexity when then problem is viewed as a parameterized problem. An algorithm designer faced with a "hard" problem should be tempted to look for reasonable parameterizations of the problem in order to discover those for which something may be practically achievable.

We consider the parameterized complexity of *scheduling to minimize tardy tasks.* Scheduling to minimize tardy tasks is a well-known problem that has several related variations. The majority of these are known to be NP-complete.

We concentrate on the following scenario. We are given a set T of tasks, each of unit length and having an individual deadline $d(t) \in Z^+$, a set of precedence constraints on T, and a positive integer $k \leq |T|$.

We ask "Is there a one-processor schedule for T that obeys the precedence constraints and contains no more than k late tasks?"

This problem is NP-complete even if the precedence constraints are modelled by a partial order consisting only of chains, i.e. each task has at most one immediate predecessor and at most one immediate successor. It can be solved in polynomial time if $k = 0$, or if the set of precedence constraints is empty.

We also make the dual inquiry “Is there a one-processor schedule for T that obeys the precedence constraints and contains at least k tasks that are on time i.e. no more than $|T| - k$ late tasks?”

Within the framework of classical complexity theory, these two questions are different instances of the same problem. However, from the parameterized point of view, they give rise to two separate problems which may be studied independently of one another.

We model the jobs and their precedence constraints as a finite ordered set, or partial order. A schedule is a *linear extension* of this partial order.

We recall the following definitions.

A *linear extension* $L = (\preceq, P)$ of a partial order $(P, \leq)$ is a total ordering of the elements of P in which $a \prec b$ in L whenever $a < b$ in P.

If $(P, \leq)$ is a partial order, a subset X of P is a *chain* iff any two distinct elements of X are comparable, and an *antichain* iff no two distinct elements of X are comparable.

By Dilworth’s theorem [43], the minimum number of chains which form a partition of $(P, \leq)$ is equal to the size of a maximum antichain. This number is the *width* of $(P, \leq)$, denoted $w(P)$.

In this chapter, we show that, in the general case, each of the problems defined by the questions above is hard for the parameterized complexity class $W[1]$. This gives us strong evidence for the likely parameterized intractability of these problems. For either problem, there does not exist a constant c, such that for all fixed k, the problem can be solved in time $O(n^c)$ (in other words, no FPT algorithm) unless an unlikely collapse occurs in the W-hierarchy. For a detailed definition of the class $W[1]$, and a description of the W-hierarchy, see Chapter 7.

In contrast, in the case where the set of precedence constraints can be modelled by a partial order of *bounded width*, we show that both problems are FPT.

4.2 Parameterized complexity of schedules to minimize tardy tasks

Classically, we consider the following two questions as just different instances of the same NP-complete problem, but as parameterized problems they are notably different. In the first case we ask for a schedule with no more than k late tasks, in the second case we ask for a schedule with no more than $|T| - k$ late tasks.

k-LATE TASKS

Instance:	A set T of tasks, each of unit length and having an individual deadline $d(t) \in Z^+$; and a set $(P, \leq)$ of precedence constraints on T.
Parameter:	A positive integer $k \leq |T|$.
Question:	Is there a one-processor schedule for T that obeys the precedence constraints and contains no more than k late tasks?

k-TASKS ON TIME

Instance:	A set T of tasks, each of unit length and having an individual deadline $d(t) \in Z^+$; and a set $(P, \leq)$ of precedence constraints on T.
Parameter:	A positive integer $k \leq |T|$.
Question:	Is there a one-processor schedule for T that obeys the precedence constraints and contains at least k tasks that are on time?

From the parameterized point of view we have two separate problems, each with parameter k. It is true that T has a schedule with at least k tasks on time iff T has a schedule with at most $|T| - k$ late tasks. This gives us a polynomial-time reduction, transforming an instance (T, k) of k-TASKS ON TIME into an instance (T, k') of k-LATE TASKS. However, this is not a parameterized transformation, since $k' = |T| - k$ is not purely a function of k.

Indeed, while we show here that both these problems are $W[1]$-hard, it is possible that they inhabit quite separate regions of the W-hierarchy.

Our results rely on the following theorem from [30].

Theorem 4.1. *k-CLIQUE is complete for the class $W[1]$.*

k-CLIQUE is defined as follows:

Instance:	A graph $G = (V, E)$.
Parameter:	A positive integer k.
Question:	Is there a set of k vertices $V' \subset V$ that forms a complete subgraph of G (that is, a clique of size k)?

Theorem 4.2. *k-LATE TASKS is $W[1]$-hard.*

Proof : We transform from k-CLIQUE.

$(G = (V, E),\ k) \rightarrow (T, (P, \leq),\ k')$ where $k' = k(k-1)/2 + k$.

We set up T and $(P, \leq)$ as follows:

For each vertex v in V, T contains a task t_v. For each edge e in E, T contains a task s_e. The partial order relation constrains any edge task to be performed before the 2 vertex tasks corresponding to it's endpoints. So we have a two-layered partial order as shown below.

We set the deadline for edge tasks to be $|E| - k(k-1)/2$, the deadline for the vertex tasks to be $|E| - k(k-1)/2 + (|V| - k)$. At most $k(k-1)/2 + k$ tasks can be late. At least $k(k-1)/2$ edge tasks will be late. The bound is only achieved if $k(k-1)/2$ tasks in the top row are late and they only block k tasks in the bottom row.

Thus, a YES for an instance $(T, (P, \leq), k')$ means that the $k(k-1)/2$ edge tasks that are late only block k vertex tasks, and these correspond to a clique in G. $\square$

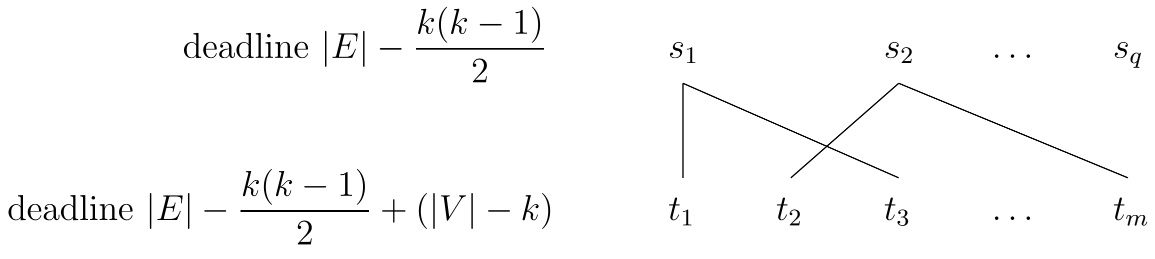

deadline $|E| - \frac{k(k-1)}{2}$
deadline $|E| - \frac{k(k-1)}{2} + (|V| - k)$
s_1
s_2
$\ldots$
s_q
t_1
t_2
t_3
$\ldots$
t_m

Figure 4.1: Gadget for k-LATE TASKS transformation.

Theorem 4.3. *k-TASKS ON TIME is $W[1]$-hard.*

Proof : Again, we transform from k-CLIQUE.

$(G = (V, E),\ k) \to (T, (P, \leq),\ k')$ where $k' = k(k + 1)/2$.

We set up T and $(P, \leq)$ as follows:

For each vertex v in V, T contains a task t_v. For each edge e in E, T contains a task s_e. The partial order relation constrains any edge task to be performed after the 2 vertex tasks corresponding to it's endpoints. So we have a two-layered partial order as shown below.

We set the deadline for vertex tasks to be k, and set the deadline for edge tasks to be $k(k + 1)/2$. Therefore, we can only do at most k vertex tasks on time, then $k(k - 1)/2$ edge tasks on time.

Thus, a YES for an instance $(T, (P, \leq), k')$ means that the $k(k-1)/2$ edge tasks done on time fall below the k vertex tasks done on time, and these correspond to a clique in G. □

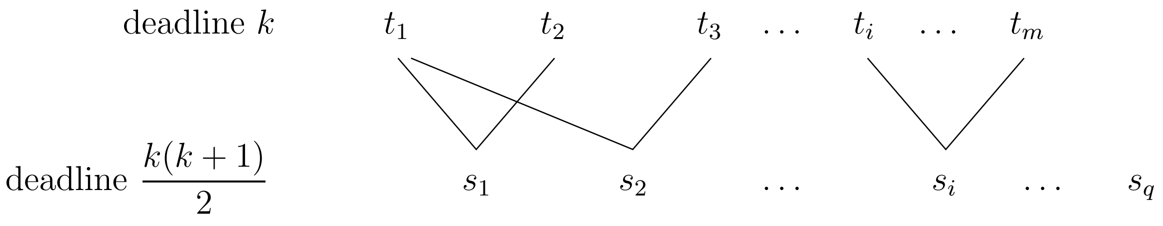


Figure 4.2: Gadget for k-TASKS ON TIME transformation.

4.3 Parameterized complexity of bounded-width schedules to minimize tardy tasks

The two results above rely heavily on the $O(n)$ width of the constructed partial order. This leads to consideration of partial orders in which the width is bounded, and treated as a parameter.

We now recast the problems using a new parameter (k, m) , $k \leq |T|$, $m = w(P)$. Under this parameterization, we find that both problems are fixed-parameter tractable (FPT).

4.3.1 FPT algorithm for k-LATE TASKS

We first compute $w(P)$ and decompose $(P, \leq)$ into a set of $w(P)$-many chains. We can either fix our choice of m, so that $m = w(P)$, or abort if $w(P)$ is outside the range we wish to consider.

To compute $w(P)$ and a decomposition of $(P, \leq)$ requires the following:

T is the ground set of P. Form a bipartite graph $K(P)$ whose vertex set consists of two copies T' and T'' of T. In the graph, let (x', y'') be an edge iff $x < y$ in P. Then a maximum matching in $K(P)$ corresponds uniquely to a minimum partition of $(P, \leq)$ into chains [26].

Any bipartite matching algorithm can be used to find the optimum partition, the best known algorithm being $O(n^{2.5})$ [61].

Now let $n = |T|$. There are n tasks to schedule, and one of these has to go in the *nth* position in the final ordering. This must be some maximal element in the partial order $(P, \leq)$, since it cannot be an element that has a successor.

There are at most m maximal elements to choose from, so consider these. If there is a maximal element with deadline $\geq n$ then put it in the *nth* position. This will not affect the rest of the schedule adversely, since no other element is forced into a later position by this action. We now ask "Is there a schedule for the remaining $n - 1$ elements that has at most k late tasks?".

If there is no maximal element with deadline $\geq n$ then one of the maximal elements in $(P, \leq)$ has to go in the *nth* position and will be late, but which one should be chosen? Delaying on particular chains may be necessary in obtaining an optimal schedule. For example, in the $W[1]$-hardness result given in the last section, it is important to leave the clique elements until last.

In this case, we begin to build a bounded search tree. Label the root of the tree with $\emptyset$. For each of the (at most m) maximal elements produce a child node, labelled with a partial schedule having that element in the *nth* position, and ask "Is there a schedule for the remaining $n - 1$ tasks that has at most $k - 1$ late tasks?".

The resulting tree will have at most m^k leaves. We branch only when forced to schedule a late task, which can happen at most k times, and each time produce at most m children.

On each path from root to leaf at most $O(mn)$ computations are done.

We can use an adjacency list representation of $(P, \leq)$, with a separate inlist and outlist for each element, containing respectively the elements covered by, and covering, that element. Thus, a maximal element is one whose outlist is empty.

Each element is inserted into the schedule after considering at most m maximal elements of the current partial order.

After scheduling a maximal element and removing it from $(P, \leq)$, we can find any new maximal elements by looking at the inlist of the removed element and checking for any element whose outlist is m, so this routine takes time $O(m)$.

Thus, the running time of the algorithm is $O(m^{k+1}n + n^{2.5})$.

4.3.2 FPT algorithm for k-TASKS ON TIME

As before, we first compute $w(P)$ and decompose $(P, \leq)$ into a set of $w(P)$-many chains.

This time, we will reduce the input to a problem kernel. That is, a new problem instance whose size is bounded by a function of the parameter.

We first run through each chain separately. For each element in a chain, we count the number of its predecessors in all the chains and check whether this is less than its deadline.

If we find any chain with k elements that could be scheduled on time we are done. We schedule this chain, along with all the necessary predecessors from other chains, as an initial segment of the schedule.

Otherwise, there are $< mk$ elements that could possibly be on time in the final schedule. There less than m^{mk} ways to order these "possibly on-time" elements relative to one another (some of these orderings may be incompatible with the original constraints). Their order relative to others in the same chain is fixed, so we have at most m choices for the one which occurs first in the final schedule, at most m choices for the one which occurs second , and so on.

We can try out all the possible orderings, throwing in the necessary predecessors along the way. If one of the orderings is successful, we report "yes", otherwise "no".

To check each ordering requires $O(n)$ computations.

We again use the adjacency list representation of $(P, \leq)$, described above.

We work backwards from each of the elements of the ordering in turn, scheduling the predecessors of the first element, followed by the first element itself, then predecessors of the second that do not lie below the first, and so on.

This can be done by recursively checking the inlists of each element and its predecessors. No element of $(P, \leq)$ will be considered more than once, since, if we encounter an already-scheduled element, we can assume that all its predecessors have been dealt with. If we find an element of the ordering has already been scheduled when we come to consider it, we can discard this particular ordering immediately, as it must be incompatible with the original constraints.

The same approach can be used in the initial counting process, and in scheduling a chain as an initial segment, if a suitable candidate is discovered.

Thus, the running time of the algorithm is $O(m^{mk}n + n^{2.5})$.

4.3.3 Dual parameterizations

For *dual* parameterizations, such as those we present in this chapter, it is often the case that one is FPT while the other is $W[1]$-hard. Frequently, switching from considering the "maximum number of bad things" to considering the "minimum number of good things" can produce this significant change in problem complexity in the parameterized setting. The dual parameterizations that we present are two of only a very few known exceptions where this rule of thumb does not apply.

CHAPTER 5
ONLINE WIDTH METRICS

5.1 Introduction

Real-life online situations often give rise to input patterns that seem to be “long” and “narrow”, that is, pathlike. One could argue that the most compelling reason for attempting to solve a problem online is that the end of the input is “too long coming” according to some criteria that we have. Given such a situation, we attempt to do the best we can with the partial information available at each timestep.

The usual theoretic model for online problems has the input data presented to the algorithm in units, one unit per timestep. The algorithm produces a string of outputs: after seeing t units of input, it needs to produce the tth unit of output. Thus, the algorithm makes a decision based only on partial information about the whole input string, namely the part that has been read so far. How good the decision of the algorithm is at any given step t may depend on the future inputs, inputs that the algorithm has not yet seen.

In this chapter we make a start on an investigation into *online width metrics*. Our long-term goals are two-fold.

Firstly, we seek to understand the effects of using width metrics as restrictions on the input to online problems. As mentioned above, online situations often give rise to input patterns that seem to naturally conform to restricted width metrics, in particular to bounded pathwidth. We might expect to obtain online algorithms having good performance, for various online problems, where we restrict the allowed input to instances having some form of bounded pathwidth.

Secondly, we seek to understand the effects of restricting the *presentation* of the input to some form of bounded width decomposition or layout. The method of presentation of the input structure to an algorithm has a marked effect on performance. Indeed, this observation underpins the study of online algorithms in the first place. Apart from noting that, for nearly all problems, we can generally do better in the offline case, where we have available complete information about the input in advance, consider the following two versions of the (offline) regular language recognition problem. In the first case, we are given a regular expression defining a regular language. We have a simple linear-time strategy to determine whether or not a given input string is a member of the language, we convert the regular expression into a deterministic finite automaton, and then run the automaton on the given input string. In the second case, we are given a Turing machine that halts on all, and only, strings that are in the language. The Turing machine defines the same regular language, but in a potentially far less useful manner. We can run the Turing machine on a given input string, but, at any particular timestep, unless the machine has already halted, we are unable to determine whether or not the string is in the language. It may just be the case that the machine will halt, but not yet.

To lay the foundations of the program described here, we concentrate on *online graph coloring*. Graph coloring is closely related to the problem of covering an ordered set by chains or antichains, so we can expect any results obtained to be broadly applicable in our chosen area of scheduling problems modelled by partial orderings. For example, online graph coloring can be applied to processor assignment and register allocation problems. There has been a considerable amount of work done on online graph coloring. However, approaches similar to the one we are taking are notably absent from the literature.

5.2 Competitiveness

We measure the performance of an online algorithm, or gauge the difficulty of an online problem, using the concept of *competitiveness*, originally defined by Sleator and Tarjan [107] (see also Manasse, McGeoch, and Sleator [75]).

Suppose that P is an online problem, and A is an online algorithm for P. Let $c \geq 1$ be a constant. We say that A is *c-competitive* if, for any instance I of problem P,

$$cost_A(I) \leq c \cdot cost_{opt}(I) + b$$

where *opt* is an optimal offline algorithm that sees all information about the input in advance, and b is a constant independent of I. In other words, A pays at most $c + O(1)$ times the optimal cost, for any given input string.

We say that a given online problem P is c-competitive if there exists a c-competitive algorithm for P, and we say that it is *no better than c-competitive* if there exists no c'-competitive algorithm for P for any $c' \leq c$.

An online algorithm A is said to be *optimal* for P if A is c-competitive and P is no better than c-competitive.

Competitive analysis measures algorithm performance relative to what is achievable by an omniscient algorithm, rather than in absolute terms. If an algorithm A is c-competitive, then we say that A has a *performance ratio* of c.

5.3 Online presentations

We define an *online presentation* of a graph G as a structure $G^< = (V, E, <)$ where $<$ is a linear ordering of V. G is presented one vertex per timestep, v_1 at time 1, v_2 at time 2, . . . and so on. At each step, the edges incident with the newly introduced vertex and those vertices already "visible" are also presented. In this thesis we use the terms *online presentation* and *online graph* interchangeably.

Let $V_i = \{v_j \mid j \leq i\}$ and $G_i^< = G^<[V_i]$, the online subgraph of $G^<$ induced by V_i. An algorithm that solves some online problem on G will make a decision regarding v_i (and/or the edges incident with v_i) using only information about $G_i^<$.

An extension of this model adds the notion of *lookahead*, where the algorithm is shown a limited portion of the remaining graph at the time that it must make each decision. In this case the algorithm will make a decision regarding v_i (and/or the edges incident with v_i) using only information about $G_{i+l}^<$, for some $l \geq 1$. Another way to view lookahead is as limited revision, being able to see the next l units of input at the time that we make each decision is essentially the same as being able to revise the last l decisions each time we see the next unit of input.

Now, we introduce a different method of presenting a graphlike structure online.

First, fix some arbitrary constant (parameter) k. At each timestep we present one new *active* vertex that may be incident with at most k active vertices previously presented. Once a vertex has been presented we may render some of the current set of active vertices *inactive* in preparation for the introduction of the next new vertex. At no point do we allow more than $k+1$ active vertices, and we do not allow a new vertex to be incident with any inactive vertex.

These requirements mean that any graph presented in this fashion must have bounded pathwidth (pathwidth k). We are, in effect, presenting the graph as a path decomposition, one node per timestep. We denote such an online presentation of a graph G as $G^{<\,\text{path}\,k}$.

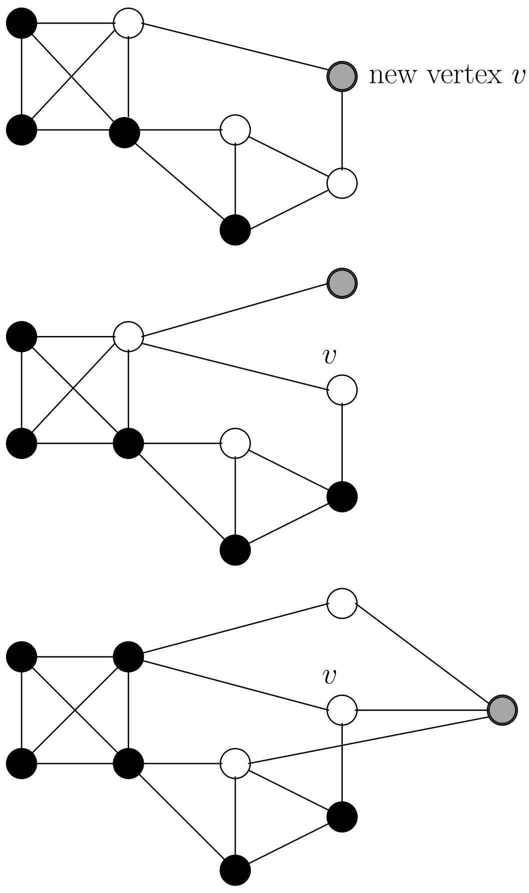


Figure 5.1: Online presentation of a pathwidth 3 graph using 4 active vertices. Vertex v remains active for 3 timesteps.

We can add the further requirement that any vertex may *remain active* for at most l timesteps, for some arbitrary constant (parameter) l.

We say that a path decomposition of width k, in which every vertex of the underlying graph belongs to at most l nodes of the path, has pathwidth k and *persistence* l, and say that a graph that admits such a decomposition has *bounded persistence pathwidth.*

This method of presentation further restricts the class of graphs that can be presented, but in a seemingly quite natural fashion. We are assuming, in a sense, at most k parallel "channels", where dependencies are limited to short timeframes, eg. if vertex (event) b appears much later than vertex a, then b can be affected only indirectly by a.

5.4 Online coloring

An online algorithm A for coloring an online graph $G^<$ will determine the color of the ith vertex of $G^<$ using only information about $G^<_i$. A colors the vertices of $G^<$ one at a time in the order $v_1 < v_2, \cdots$, and at the time a color is irrevocably assigned to v_i, the algorithm can only see $G^<_i$. The number of colors that A uses to color $G^<$ is denoted $\chi_A(G^<)$. The maximum of $\chi_A(G^<)$ over all online presentations of G is denoted by $\chi_A(G)$. If Γ is a class of graphs, the maximum of $\chi_A(G)$ over all G in Γ is denoted by $\chi_A(\Gamma)$. The *online chromatic number* of Γ is the minimum of $\chi_A(\Gamma)$ over all online algorithms A.

A simple, but important, example of an online algorithm is *First-Fit*, which colors the vertices of $G^<$ with an initial sequence of the colors $\{1, 2, \ldots\}$ by assigning to v_i the least color that has not already been assigned to any vertex in $G^<_i$ that is adjacent to v_i. In Section 5.5 we present some results concerning the First-Fit coloring of graphs of bounded pathwidth.

Szegedy [111] has shown that, for any online coloring algorithm A and integer k, there is an online graph $G^<$ on at most $k(2^k - 1)$ vertices with chromatic number k (the minimum number of colors required to color G offline) on which A will use $2^k - 1$ colors. This yields a lower bound of $\Omega(\frac{n}{(log\, n)^2}$ for the performance ratio of any online coloring algorithm on general graphs. Note that the worst possible performance ratio on general graphs is n. Lovasz, Saks, and Trotter [73] have given an algorithm that achieves a performance ratio $O(\frac{n}{(log\, n)^*}$ on all graphs.

Online coloring of some restricted classes of graphs has been considered. In the bipartite (2-colorable) case it can be shown that, for any online coloring algorithm A and integer k, there is an online tree $T^<$ with 2^{t-1} vertices on which A will use

$\geq t$ colors. Thus, we get a lower bound of $\Omega(log\ n)$ for any online algorithm on bipartite graphs. Lovasz, Saks, and Trotter [73] give an algorithm that colors any bipartite online graph using at most $1 + 2\ log\ \ n$ colors.

Kierstead and Trotter [66] have given an online coloring algorithm that achieves a performance ratio of 3 on interval graphs, which is also best possible. Kierstead [64] has shown that First-Fit has a constant performance ratio on the class of interval graphs. Gyarfas and Lehel [60] have shown that First-Fit achieves a constant performance ratio on split graphs, complements of bipartite graphs, and complements of chordal graphs.

One approach that is similar in flavour to ours is presented by Irani [62]. He introduces the notion of *d-inductive* graphs. A graph G is *d-inductive* if the vertices of G can be ordered in such a way that each vertex is adjacent to at most d higher numbered vertices. Such an ordering on the vertices is called an *inductive order.* As for a path or tree decomposition, an inductive order is not necessarily unique for a graph. An inductive order of a graph G defines an *inductive orientation* of G, obtained by orienting the edges from the higher numbered vertices to the lower numbered vertices. Notice that, in an inductive orientation, the indegree of each vertex is bounded by d. Hence, any d-inductive graph is $d + 1$ colorable. In this thesis, for consistency of terminology, we prefer to call a d-inductive order an *inductive layout* of *width d.*

In [62] it is shown that, if G is a d-inductive graph on n vertices, then First-Fit uses at most $O(d \cdot log\, n)$ colors to color any online presentation $G^<$ of G. Moreover, for any online coloring algorithm A, there exists a d-inductive online graph $G^<$ such that A uses at least $O(d \cdot log\, n)$ colors to color $G^<$.

5.5 Online coloring of graphs with bounded pathwidth

We consider two ways in which to formulate the problem of online coloring of graphs with bounded pathwidth.

We can define a parameterized "promise" problem, where we fix a bound k on the pathwidth of any input graph G, and then proceed to present G as a structure

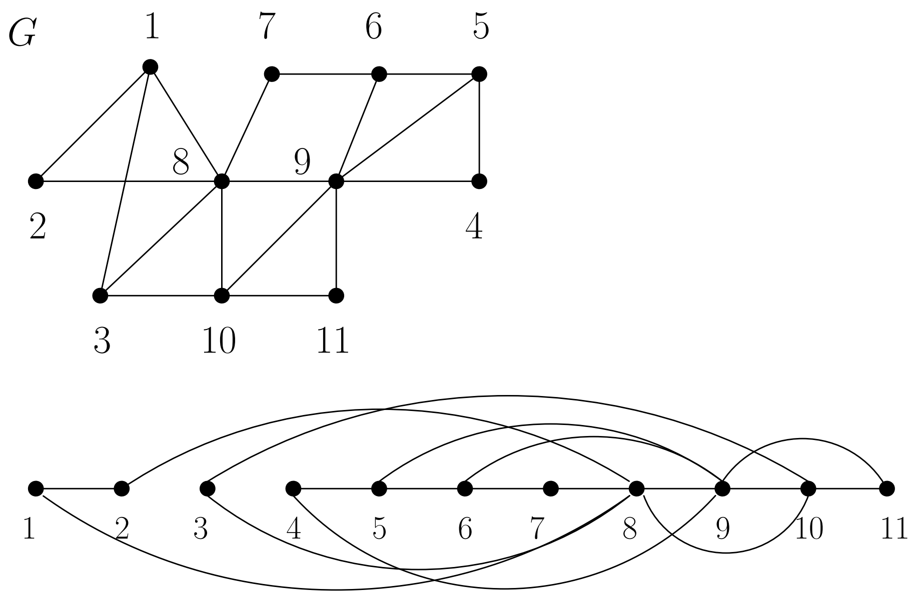


Figure 5.2: Graph G having treewidth 2, and an inductive layout of width 2 of G.

$G^< = (V, E, <)$ where $<$ is an *arbitrary* linear ordering of V.

Alternatively, we can define a parameterized "presentation" problem, where we fix a bound k on the pathwidth of any input graph G, and then proceed to present G as an implicit path decomposition, in the manner described in Section 5.3 above.

5.5.1 Preliminaries

We first give some preliminary definitions and observations that will be used in the rest of this chapter.

A connection between graphs of bounded pathwidth, and graphs that admit an inductive layout of bounded width is given by the following lemma.

Lemma 5.1. *Any graph G of pathwidth k, or treewidth k, is k-inductive.*

Proof: In Section 2.2.1 we introduced the notion of a *nice* tree decomposition. Recall:

If $G = (V, E)$ is a graph with treewidth at most k then G has a tree decomposition $TD = (T, \mathcal{X})$, $T = (I, F)$, $\mathcal{X} = \{X_i \mid i \in I\}$, of width k, such that a root r of T can be chosen, such that every node $i \in I$ has at most two children in the rooted tree with r as the root, and

1. If a node $i \in I$ has two children, j_1 and j_2, then $X_{j_1} = X_{j_2} = X_i$ (i is called a **join** node).

2. If a node $i \in I$ has one child j, then either $X_i \subset X_j$ and $|X_i| = |X_j| - 1$ (i is called a **forget** node), or $X_j \subset X_i$ and $|X_j| = |X_i| - 1$ (i is called an **introduce** node).

3. If a node $i \in I$ is a leaf of T, then $|X_i| = 1$ (i is called a **start** node).

4. $|I| = O(k \cdot |V|)$.

Given a nice tree (path) decomposition, $TD(G)$, of a graph G, we can produce an inductive layout of width k by traversing $TD(G)$ in a breadth-first manner, leaves to root. At each **forget** node, X_i, we place the vertex that is removed, $v \in X_j - X_i$, in the next position in the inductive layout.

At the time that each vertex v is placed in the layout there will be at most k vertices to which v may be adjacent that have not yet been placed, namely those vertices that appear in the forget node X_i. We denote this node as $remove(v)$. The nature of a tree decomposition means that v cannot be adjacent to any vertex that appears only in nodes higher in the tree than $remove(v)$. Any vertex adjacent to v that appears only in nodes lower in the tree than $remove(v)$ will have already been placed in the layout. □

A graph $G = (V, E)$ is an *interval graph* if there is a function ψ which maps each vertex of V to an interval of the real line, such that for each $u, v \in V$ with $u \neq v$,

$$\psi(u) \cap \psi(v) \neq \emptyset \Leftrightarrow (u, v) \in E.$$

The function ψ is called an *interval realization* for G.

The relation between interval graphs and graphs of bounded pathwidth is captured by the following lemma.

Lemma 5.2. *A graph G has pathwidth at most k if and only if G is a subgraph of an interval graph G', where G' has maximum clique size at most $k + 1$.*

Proof: $\Rightarrow$ Let $G = (V, E)$ be a graph and let $PD = (X_1, \ldots, X_t)$ be a path decomposition of width k for G.

Let G' be the supergraph of G with $E' = \{\ (u,v) \mid \exists_{1 \leq i \leq t}\ u, v \in X_i\}$. It can be seen that PD is also a path decomposition of width k for G'. The nature of a path decomposition means that the maximum clique size of G' must be $k+1$.

It remains to show that G' is an interval graph. Let $\psi : V \to \{1, \ldots, n\}$ be defined as follows. For each $v \in V$, if the subsequence of PD consisting of all nodes containing v is $(X_j, \ldots, X_l)$ then $\psi(v) = [j, l]$. For each pair of vertices $u, v \in V$, $(u,v) \in E'$ if and only if $\psi(u)$ and $\psi(v)$ overlap.

$\Leftarrow$ Let $G = (V, E)$ be a graph and let $G' = (V', E')$ be an interval graph, with maximum clique size $k+1$, such that $E \subseteq E'$. Note that if $V' \neq V$ then the subgraph of G' induced by V is an interval graph with maximum clique size at most $k+1$ that contains G as a subgraph. Thus, we can assume that $V = V'$.

Let $\psi : V \to I$ be an interval realization for G'. For each vertex v, $\psi(v) = [l_v, r_v]$ for some integers l_v and r_v. Let $u_1, \ldots, u_{|V|}$ be an ordering of V such that for all i, j with $1 \leq i < j \leq |V|$ we have $l_{u_i} \leq l_{u_j}$. For each $1 \leq i \leq |V|$, let $X_i = \{v \in V \mid l_{u_i} \in \psi(v)\}$. Each X_i contains at most $k+1$ vertices, since the maximum clique size of G' is $k+1$. Thus, $PD = (X_1, \ldots, X_{|V|})$ is a path decomposition of width at most k for G' and, therefore, for G. $\square$

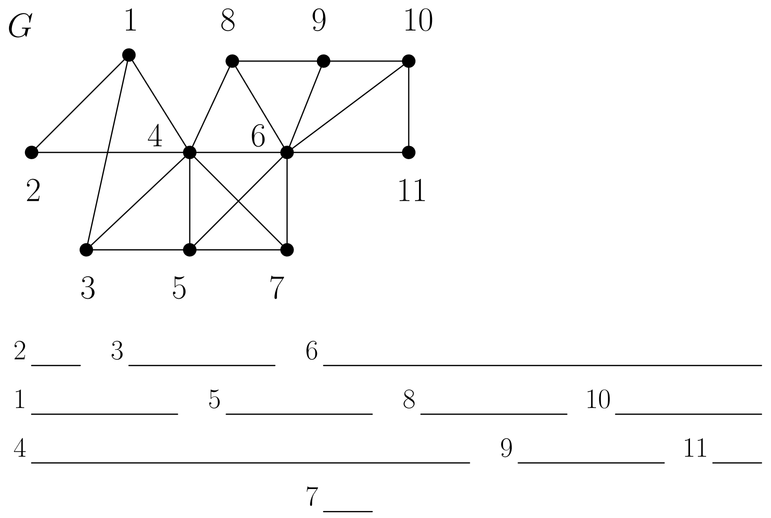


Figure 5.3: Interval graph G having pathwidth 3, and an interval realization of G.

5.5.2 The presentation problem

Let us now consider the "presentation" problem for online coloring of graphs of bounded pathwidth. In tandem, we will also consider the presentation problem for d-inductive graphs.

Suppose we have any inductive layout of width d, $\sigma = \{v_0, v_1, \ldots, v_n\}$, for a given graph G. Recall that that each vertex in σ is adjacent to at most d higher numbered vertices. Consider the inverse ordering $\sigma' = \{v_n, v_{n-1}, \ldots, v_0\}$. σ' defines an *inductive orientation* of G with each vertex in σ' adjacent to at most d lower numbered vertices.

Lemma 5.3. *If G is a d-inductive graph and σ' is an inductive orientation of G, then First-Fit will use at most $d + 1$ colors to color $G^{<\sigma'}$.*

Proof: Suppose we have a pool of $d + 1$ colors with which to color $G^{<\sigma'}$. At each step i, $0 \leq i \leq n$, the newly presented vertex v_i will be adjacent to at most d already-colored vertices, colored using at most d of the $d + 1$ colors available. We will always have at least one unused color with which to color the newly presented vertex v_i. $\square$

On the other hand, there is a d-inductive graph G, and an inductive layout of width d for G, $\sigma = \{v_0, v_1, \ldots, v_n\}$, such that First-Fit requires $O(d \cdot log\, n)$ colors to color $G^{<\sigma}$. The d-inductive online graph $G^{<}$ used to demonstrate the lower bound $O(d \cdot log\, n)$ in [62] is, in fact, presented as an inductive layout of width d.

Recall that, if G is a d-inductive graph on n vertices, then $O(d \cdot log\, n)$ is also an upper bound on the number of colors required by First-Fit to color any online presentation $G^{<}$ of G. Thus, an inductive layout of width d for a given d-inductive graph G may characterize the worst-possible online presentation for First-Fit acting on G.

In the case of graphs of bounded pathwidth, if we undertake to present a graph G in the form of an implicit path decomposition, then we are effectively enforcing the presentation to be, if not best-possible, then at least "very good" for First-Fit acting on G. If G is presented as $G^{<\,\text{path}\,k}$, an implicit path decomposition of width k, then $G^{<\,\text{path}\,k}$ is also a width k inductive *orientation* of G, as the trivial proof of the following lemma makes clear.

Lemma 5.4. *If G is a graph of pathwidth k, presented in the form of an implicit path decomposition, then First-Fit will use at most $k+1$ colors to color $G^{< \text{path}\, k}$.*

Proof: In the online presentation $G^{< \text{path}\, k}$ each vertex v presented will be adjacent to at most k *active* vertices. Suppose we have a pool of $k+1$ colors with which to color $G^{< \text{path}\, k}$. At each step, the current active vertices will be colored using at most k of the $k+1$ colors available and we will always have at least one unused color with which to color the newly presented vertex v. □

This is best possible in the sense that the chromatic number (and, therefore, the online chromatic number) of the class of graphs of pathwidth k is $k+1$. The class of graphs of pathwidth k admits graphs that contain a clique of size $k+1$ and these cannot be colored using fewer than $k+1$ colors. Note that $G^{<\text{path}\, k}$ may not contain all of the information required to color G *optimally* online, as the following lemma shows.

Lemma 5.5. *For each $k \geq 0$, there is a tree T of pathwidth k presented as $T^{< \text{path}\, k}$ on which First-Fit can be forced to use $k+1$ colors.*

Proof: Suppose T_0 is a connected tree with pathwidth 0, then T must consist of a single vertex (any graph of pathwidth 0 must consist only of isolated vertices) so First-Fit will color $T_0^{< \text{path}\, 0}$ with one color.

Suppose T_1 is a connected tree with pathwidth 1 that has at least two vertices. Each vertex of $T_1^{< \text{path}\, 1}$ can be adjacent to at most one active vertex at the time of presentation. Since T_1 is connected, there must be vertices that are adjacent to an active vertex at the time of presentation in any $T_1^{< \text{path}\, 1}$. Thus, First-Fit will need to use two colors to color any $T_1^{< \text{path}\, 1}$.

Now, suppose that for any $0 \leq t < k$, there is a tree T_t of pathwidth t, and a presentation $T_t^{< \text{path}\, t}$, on which First-Fit can be forced to use $t+1$ colors. We build a connected tree T_k with pathwidth k, and a presentation $T_k^{< \text{path}\, k}$, on which First-Fit will be forced to use $k+1$ colors.

We order the trees T_t, $0 \leq t < k$, and their presentations, in descending order $[T_{k-1}, T_{k-2}, \ldots, T_0]$, and concatenate the presentations together in this order to obtain a new presentation $T_{con}^{<}$. Note that the subsequence $T_t^{< \text{path}\, t}$ of $T_{con}^{<}$ will have at most $(t+1) \leq k$ active vertices at any stage.

To obtain $T^{<\,\mathrm{path}\,k}$, we alter $T^{<}_{con}$ as follows. For each t, $0 \leq t < k$, we choose the vertex v_t from $T_t^{<\,\mathrm{path}\,t}$ that is colored with color $t+1$ by First-Fit and allow it to remain active throughout the rest of $T^{<}_{con}$. Every other vertex from $T_t^{<\,\mathrm{path}\,t}$ is rendered inactive at the conclusion of $T_t^{<\,\mathrm{path}\,t}$ in the concatenated presentation. Thus, at any stage of $T^{<}_{con}$, there will be at most $k+1$ active vertices, and at the conclusion of $T^{<}_{con}$ there will be k active vertices, one from each of the $T_t^{<\,\mathrm{path}\,t}$, $0 \leq t < k$. These k active vertices will be colored with colors $1, 2, \ldots, k$ respectively. We now present one new vertex, adjacent to each of the k active vertices, which must be colored with color $k+1$. $\square$

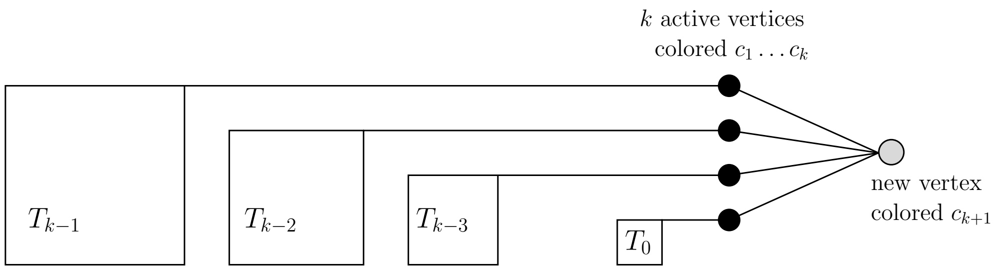


Figure 5.4: Schema of $T^{<\,\mathrm{path}\,k}$ on which First-Fit can be forced to use $k+1$ colors.

5.5.3 The promise problem

Now, let us consider the "promise" problem for online coloring of graphs of bounded pathwidth. Here, we present a graph G, having pathwidth k, as a structure $G^{<} = (V, E, <)$ with $<$ an *arbitrary* linear ordering of V.

We first consider the case where G is a *tree* of pathwidth k. We need the following result for trees of bounded pathwidth, given in [53].

Lemma 5.6. *Let T be a tree and let $k \geq 1$. $T = (V, E)$ is a tree of pathwidth at most k if and only if there is a path $P = (v_1, \ldots, v_s)$ in T such that $T[V - V(P)]$ has pathwidth at most $k-1$. That is, if and only if T consists of a path with subtrees of pathwidth at most $k-1$ connected to it.*

Lemma 5.7. *First-Fit will use at most $3k+1$ colors to color any $T^{<}$ where T is a tree of pathwidth k.*

Proof: Let $k = 0$, then T consists only of an isolated vertex, and First-Fit requires only one color to color $T^{<}$.

Let k be ≥ 1. Suppose that the bound holds for $k-1$: for any tree T of pathwidth at most $k-1$ First-Fit colors any $T^<$ with at most $3k-2$ colors.

By lemma 5.6, any tree T of pathwidth k consists of a path P and a collection of subtrees of pathwidth at most $k-1$, each connected to a single vertex on the path P.

Let v_i be a vertex appearing in one of the subtrees of T. When v_i is presented in $T^<$, at time i, it will be colored by First-Fit using a color chosen from the first $3k-1$ colors of $\{1, 2, \ldots\}$.

Let T^i be the subtree in which v_i appears. Let p_i be the path vertex to which the subtree T^i is connected. Let $V_i = \{v_j \mid j \leq i\}$ and $T_i^< = T^<[V_i]$, the online subgraph of $T^<$ induced by V_i.

Suppose that p_i is not present in $T_i^<$. Then the component of $T_i^<$ containing v_i is a tree of pathwidth at most $k-1$, disjoint from all other components of $T_i^<$. Thus, First-Fit will color v_i using a color chosen from the first $3k-2$ colors of $\{1, 2, \ldots\}$.

Suppose that p_i is present in $T_{i-1}^<$. Then, in $T_i^<$, v_i will be adjacent to at most one vertex in the component of $T_{i-1}^<$ containing p_i. Suppose that this is the case and that this vertex has been colored with some color C_i. Consider the other components of $T_{i-1}^<$ to which v_i may become connected. Together with v_i, these form a tree of pathwidth at most $k-1$. First-Fit will require at most $3k-2$ colors to color this tree, but C_i cannot be used to color v_i. If $C_i \notin \{1, 2, \ldots, 3k-2\}$ then First-Fit will color v_i using a color chosen from $\{1, 2, \ldots, 3k-2\}$. If $C_i \in \{1, 2, \ldots, 3k-2\}$ then First-Fit will color v_i using a color chosen from $\{1, 2, \ldots, 3k-1\} - C_i$. If, in $T_i^<$, v_i is not connected to the component of $T_{i-1}^<$ containing p_i, then First-Fit will color v_i using a color chosen from $\{1, 2, \ldots, 3k-2\}$.

Let v_i be a vertex appearing in the path of T. When v_i is presented in $T^<$, at time i, it will be colored by First-Fit using a color chosen from the first $3k+1$ colors of $\{1, 2, \ldots\}$.

Let $V_i = \{v_j \mid j \leq i\}$ and $T_i^< = T^<[V_i]$, the online subgraph of $T^<$ induced by V_i.

In $T_i^<$, v_i may be adjacent to single vertices from each of many subtrees. Note that, in $T_{i-1}^<$, each of the subtrees that becomes connected to v_i is disjoint from all other components of $T^{<}_{i-1}$, so any such subtree will have been colored only with colors from $\{1, 2, \ldots, 3k-2\}$.

The path vertex v_i can also be connected to (at most) two other path vertices already colored. If v_i is not connected to any other path vertex then v_i will be colored by First-Fit using a color chosen from the first $3k-1$ colors of $\{1, 2, \ldots\}$. If v_i is connected to only one other path vertex then v_i will be colored by First-Fit using a color chosen from the first $3k$ colors of $\{1, 2, \ldots\}$. If v_i is connected to two other path vertices then v_i will be colored by First-Fit using a color chosen from the first $3k+1$ colors of $\{1, 2, \ldots\}$. □

Lemma 5.8. *For each $k \geq 0$, there is an online tree $T^{<}$, of pathwidth k, such that First-Fit will use $3k+1$ colors to color $T^{<}$.*

Proof: The proof given for lemma 5.7 suggests a way in which to present a tree of pathwidth k that will require $3k+1$ colors.

Let $k = 0$, then T consists only of an isolated vertex, and First-Fit requires $3k+1 = 1$ color to color $T^{<}$.

Let k be ≥ 1. Suppose that there is an online tree $T^{<}$, of pathwidth $k-1$, such that First-Fit is forced to use $3(k-1)+1 = 3k-2$ colors to color $T^{<}$. We build an online tree $T^{<}$, of pathwidth k, such that First-Fit is forced to use $3k+1$ colors to color $T^{<}$.

First present four sets of trees having pathwidth $k-1$. Each set contains $3k-2$ disjoint trees, all identical, which are presented one after another in such a way that First-Fit is forced to use $3k-2$ colors on each of them.

Now present a path vertex p_1 and connect p_1 to a single vertex from each of the trees in the first set so that the neighbours of p_1 use each of the colors in $\{1, 2, \ldots, 3k-2\}$. The vertex p_1 must be colored by First-Fit with color $3k-1$.

Now present a path vertex p_2 and connect p_2 to a single vertex from each of the trees in the second set so that the neighbours of p_2 use each of the colors in $\{1, 2, \ldots, 3k-2\}$. The vertex p_2 must be colored by First-Fit with color $3k-1$.

Now present a path vertex p_3 and connect p_3 to a single vertex from each of the trees in the third set, and also to the path vertex p_1, so that the neighbours of p_3 use each of the colors in $\{1, 2, \ldots, 3k-1\}$. The vertex p_3 must be colored by First-Fit with color $3k$.

Now present a path vertex p_4 and connect p_4 to a single vertex from each of the trees in the fourth set, and also to path vertices p_2 and p_3, so that the neighbours of p_4 use each of the colors in $\{1, 2, \ldots, 3k\}$. The vertex p_4 must be colored by First-Fit with color $3k + 1$. □

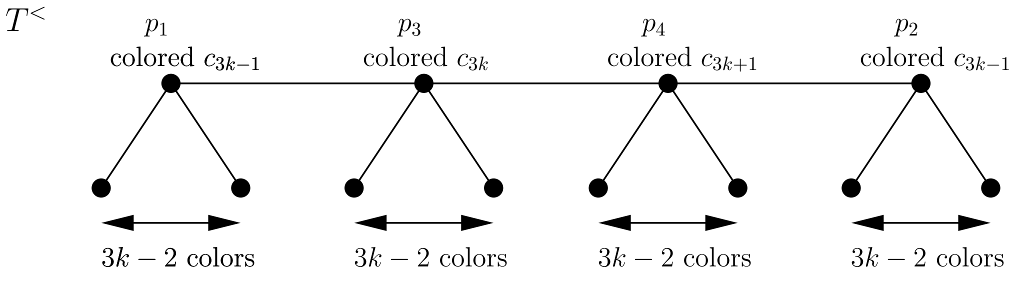


Figure 5.5: Schema of online tree $T^{<}$ of pathwidth k on which First-Fit is forced to use $3k + 1$ colors.

We now consider the "promise" problem for online coloring in the case of general graphs of pathwidth at most k.

Recall that a graph G of pathwidth k is a subgraph of an interval graph G', where G' has maximum clique size at most $k+1$. Kierstead and Trotter [66] have given an online algorithm that colors any online interval graph $G^{<}$, having maximum clique size at most $k + 1$, using $3k + 1$ colors. Thus, any graph of pathwidth k can be colored online using at most $3k + 1$ colors.

In the case of First-Fit the situation is less well understood. First-Fit does have a constant performance ratio on graphs of pathwidth at most k. Kierstead [64] has shown that for every online interval graph $G^{<}$, with maximum clique size at most $k + 1$, First-Fit will use at most $40(k + 1)$ colors. In [65] Kierstead and Qin have improved the constant here to 25.72.

Chrobak and Slusarek [38] have used an induction argument to prove that there exists an online interval graph $G^{<}$, and a constant c, where c is the maximum clique size of $G^{<}$, such that First-Fit will require at least $4.4\,c$ colors to color $G^{<}$. Such an interval graph $G^{<}$ will have pathwidth $c - 1$ and chromatic number c. Thus, the performance ratio of First-Fit on general graphs of pathwidth k must be at least 4.4.

However, experimental work by Fouhy [56] indicates that we can expect a performance ratio of 3 for First-Fit on randomly generated graphs of bounded pathwidth.

We consider the work presented in this section to be a first step in a program to investigate the ramifications of online width metrics. Of course, there are many graph width metrics, along with many online problems, that are candidates for study. However, it does seem apparent that pathwidth, or metrics that are pathwidth-like, are a natural fit in this context.

The rest of this chapter looks at a natural restriction to pathwidth arising from the work presented here.

5.6 Bounded persistence pathwidth

In Section 5.3 we introduced a quite natural online presentation scheme that gives rise to graphs having *bounded persistence pathwidth*.

Recall, we first fix two arbitrary constants (constituting the parameter) (k, l). At each timestep we present one new *active* vertex that may be incident with at most k active vertices previously presented. Once a vertex has been presented we may render some of the current set of active vertices *inactive* in preparation for the introduction of the next new vertex. At no point do we allow more than $k+1$ active vertices, and we do not allow any vertex to *remain active* for more than l timesteps.

We are, in effect, presenting the graph as a path decomposition, one node per timestep, where every vertex of the underlying graph belongs to at most l nodes of the path.

We call such a path decomposition, in which every vertex of the underlying graph belongs to at most l nodes of the path, a path decomposition of *persistence* l, and say that a graph that admits such a decomposition has *bounded persistence pathwidth*. Thus, any graph presented in this fashion must have bounded persistence pathwidth (pathwidth k, persistence l).

A related notion is *domino treewidth* introduced by Bodlaender and Engelfriet [18]. A *domino tree decomposition* is a tree decomposition in which every vertex of the underlying graph belongs to at most two nodes of the tree. *Domino pathwidth* is a special case of bounded persistence pathwidth, where $l = 2$.

Note that bounded persistence pathwidth gives us a natural characterization of graphs having both bounded pathwidth and bounded degree. If a graph G admits a path decomposition of width k and persistence l then it must be the case that all vertices in G have degree $\leq k \cdot l$. On the other hand, if G has pathwidth k and maximum degree d then the persistence that can be achieved in any path decomposition of G must be "traded off" against the resulting width.

In [45] an interesting related result is presented: graphs with bounded degree and bounded treewidth have bounded domino treewidth. It is shown that for every k, and d, there exists a constant $c_{k,d}$ such that every graph with treewidth at most k and maximum degree at most d has domino treewidth at most $c_{k,d}$. A substantially better bound for $c_{k,d}$ is presented in [17]. It seems that the techniques used in both [45] and [17] do not translate to domino pathwidth.

In the case of online coloring using First-Fit neither bounded persistence pathwidth, nor domino pathwidth, give us any real gain over our original pathwidth metric. However, the notion of persistence seems like it may be interesting, in a broader setting, in its own right.

It seems that graphs that have *high* persistence are, in some sense, "unnatural" or pathological. In real-world examples, a single data point would be unlikely to have an influence throughout an entire data structure or model. This observation has been made by Fouhy in [56].

Consider the graph G presented in figure 5.6 below. G is not really "path-like", but still has a path decomposition of width only 2. The reason for this is reflected in the presence of vertex a in *every* node of the path decomposition.

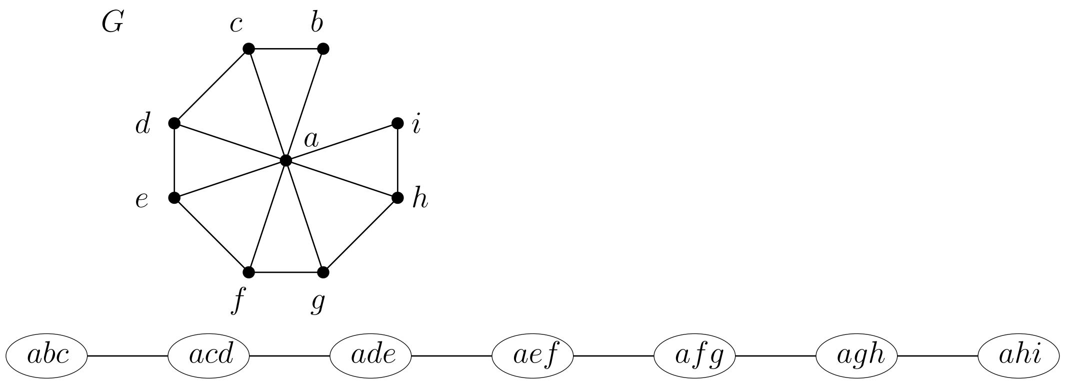


Figure 5.6: Graph G having high persistence.

5.7 Complexity of bounded persistence pathwidth

Persistence appears to be an interesting parameter in relation to graph width metrics and associated graph decompositions or layouts. However, deciding whether or not a given graph has a path decomposition of bounded persistence and bounded width appears to be a hard problem.

In this section we give some strong evidence for the likely parameterized intractability of both the bounded persistence pathwidth problem and the domino pathwidth problem. We show that the bounded persistence pathwidth problem is $W[t]$-hard, for all $t \in \mathbb{N}$ and we show that the domino pathwidth problem is $W[2]$-hard. We will not give a definition of $W[t]$-hardness ($t \in \mathbb{N}$) here, since the structure of the W-hierarchy and the notion of $W[t]$-hardness ($t \in \mathbb{N}$) is explained in some detail in Section 7. However, note that these results mean that is likely to be impossible to find FPT algorithms for either of these problems, at least in the general case.

A related result from [18] is that finding the domino treewidth of a general graph is $W[t]$-hard for all $t \in \mathbb{N}$. Our first result relies on the following theorem from [19].

Theorem 5.1. *k-BANDWIDTH is $W[t]$-hard, for all $t \in \mathbb{N}$.*

k-BANDWIDTH is defined as follows:

Instance:	A graph $G = (V, E)$.
Parameter:	A positive integer k.
Question:	Is there a bijective *linear layout* of V, $f : V \rightarrow \{1, 2, \ldots, \lvert V\rvert\}$, such that, for all $(u, v) \in E$, $\lvert f(u) - f(v)\rvert \leq k$?

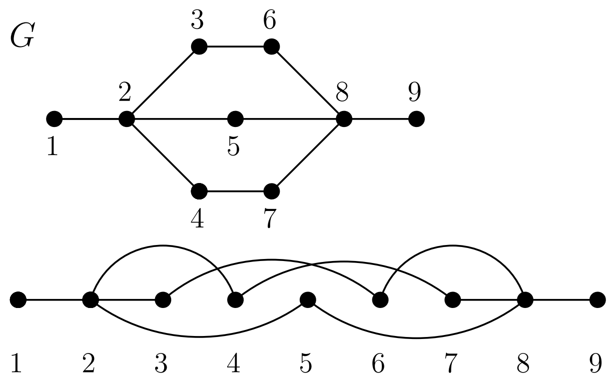


Figure 5.7: Graph G of bandwidth 3, and a layout of bandwidth 3 of G.

BOUNDED PERSISTENCE PATHWIDTH is defined as follows:

Instance:	A graph $G = (V, E)$.
Parameter:	A pair of positive integers (k, l).
Question:	Is there a path decomposition of G of width at most k, and persistence at most l?

Theorem 5.2. *BOUNDED PERSISTENCE PATHWIDTH is $W[t]$-hard, for all $t \in \mathbb{N}$*

Proof: We transform from k-BANDWIDTH.
Let $G = (V, E)$ be a graph and k the parameter. We produce $G' = (V', E')$ such that G' has a width $2(k+1)^2) - 1$, persistence $k + 1$, path decomposition iff G has bandwidth at most k.

To build G' we begin with the original graph G, and alter as follows:

1. for each vertex v in G, we introduce new vertices and form a big clique of size $(k+1)^2 + 1$ containing these vertices and v, call this C_v.

2. for each neighbour u of v (we can assume at most $2k$ of these, otherwise G cannot have bandwidth at most k), choose a unique vertex, c_{v_u}, from C_v (not v) and attach this vertex to all the vertices in C_u.

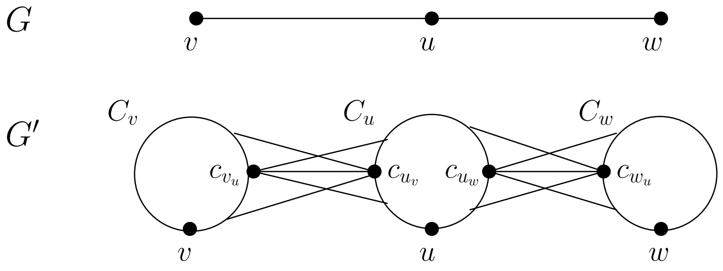


Figure 5.8: Bounded persistence pathwidth, transformation from G to G'.

$\Leftarrow$ If G has bandwidth k, then the required decomposition for G' exists.

Let $\{v_0, \ldots, v_n\}$ be a layout of bandwidth k for G. To build the decomposition $(P, \mathcal{X})$ for G' we let the ith node of the decomposition, X_i, contain $\{v_i, \ldots, v_{i+k}\}$ plus C_i, plus each c_{j_r} connected to C_r with $j \leq i$ and $r \geq i$ or $r \leq i$ and $j \geq i$.

This fulfills the requirements for a path decomposition.

1. $\bigcup_{i \in I} X_i = V'$,
2. for every edge $\{v, w\} \in E'$, there is an $i \in I$ with $v \in X_i$ and $w \in X_i$, and
3. for all $i, j, k \in I$, if j is on the path from i to k in P, then $X_i \cap X_k \subseteq X_j$.

Each node contains at most $(k+1) + k(k+1) + (k+1)^2 = 2(k+1)^2$ vertices. Thus, the decomposition has width $2(k+1)^2) - 1$.

Any v_i from G appears in at most $k+1$ nodes, being nodes X_{i-k} up to X_i. Any c_{j_r} appears in at most $k+1$ nodes, being nodes X_j up to X_r, or X_r up to X_j, where $|j - r| \leq k$. Thus, the decomposition has persistence $k+1$.

$\Rightarrow$ If the required decomposition of G' exists, then G has bandwidth k.

At some point in the decomposition, a node $X_{v'}$, containing C_v, will appear for the first time. No other C_u, $u \neq v$ can appear in this node, as there is not room.

Pick a neighbour u of v for which C_u has already appeared. C_u must have appeared for the first time in some node $X_{u'}$, where $v' - u' \leq k$, since some c_{u_v} was present in this node which must appear with C_v at some point, and c_{u_v} cannot appear in more than $k+1$ nodes.

Pick a neighbour u of v for which C_u has not yet appeared. C_u must appear for the first time some node $X_{u'}$ where $u' - v' \leq k$, since there is some c_{v_u} in $X_{v'}$ which must be present with C_u at some point and c_{v_u} cannot appear in more than $k+1$ nodes.

If we lay out the vertices of G in the order in which the corresponding cliques first appear in the decomposition, then we have a layout of G with bandwidth k. □

DOMINO PATHWIDTH is defined as follows:

Instance:	A graph $G = (V, E)$.
Parameter:	A positive integer k.
Question:	Is there a path decomposition of G having width at most k, and persistence 2?

Theorem 5.3. *DOMINO PATHWIDTH is $W[2]$-hard.*

Proof: We transform from *DOMINATING SET*, a fundamental $W[2]$-complete problem.

Let G be a graph and k the parameter. We produce G' such that G' has a width $K-1$ domino path decomposition, where $K = k^2(k+4) + k(k+3)$, if and only if G has a dominating set of size k.

$G' = (V, E)$ consists of the following components:

- **Two anchors.** Take two cliques, each with K vertices, with vertex sets $A_1 = \{a_i^1 | 1 \leq i \leq K\}$ and $A_2 = \{a_i^2 | 1 \leq i \leq K\}$.
- **The graph thread.** Let $n = |V|$. Take $P = 2n + n^2 + (n+1)$ cliques, each with $Q = k^2(k+3)$ vertices, with vertex sets $C^i = \{c_r^i | 1 \leq r \leq Q\}$, for $1 \leq i \leq P$. Join them into a "thread" by choosing separate vertices in each clique, c_{start}^i and c_{end}^i, and identifying c_{end}^i with c_{start}^{i+1}. Identify c_{start}^1 with a_1^1, and identify c_{end}^P with a_1^2. Now, for each $1 \leq i \leq (P-1)$ cliques C^i and C^{i+1} have a vertex in common, C^1 has a vertex in common with A_1, and C^P has a vertex in common with A_2.
- **The vertex cliques.** For each i of the form $i = 2n + j.n + 1$ for $0 \leq j \leq n-1$ take the clique C^i from the graph thread and add another vertex to each such C^i, to make a clique with $Q+1$ vertices. Each of these n cliques, C_j^i, represents a vertex, v_j, of G.
- **The selector threads.** Take $2n$ cliques of size 2; n^2 cliques of size $k+3$ with vertex sets $S^i = \{s_r^i | 1 \leq r \leq k+3\}$, for $1 \leq i \leq n^2$; and another $2n$ cliques of size 2. Join them into a thread as for the graph thread, so that $2n$ size 2 cliques form the first portion of the thread, and $2n$ size 2 cliques form the last portion of the thread. Now, the first $2n$ size 2 cliques form a simple path having $2n+1$ vertices and $2n$ edges, where the last vertex in this path is also an element of S^1. For each $1 \leq i \leq (n^2-1)$ cliques S^i and S^{i+1} have a vertex in common. The last $2n$ size 2 cliques form a simple path having $2n+1$ vertices and $2n$ edges, where the first vertex in this path is also an element of S^{n^2}.

  For $1 \leq i \leq n$, let $\mathcal{S}_i$ denote the ith consecutive set of n cliques of size $k+3$, $\{S^{(i-1)n+1}, \ldots, S^{(i-1)n+n}\}$.

Remove one (interior) vertex from the ith clique of $\mathcal{S}_i$, $S^{(i-1)n+i}$. If v_i is connected to v_j in G, remove one (interior) vertex from the jth clique of $\mathcal{S}_i$, $S^{(i-1)n+j}$.

Make k copies of the selector thread component described here, and identify the first vertex of the ith thread with a^1_{i+1}, the last vertex of the ith thread with a^2_{i+1}.

Each of these threads is used to select a vertex in a dominating set of G, if one exists.

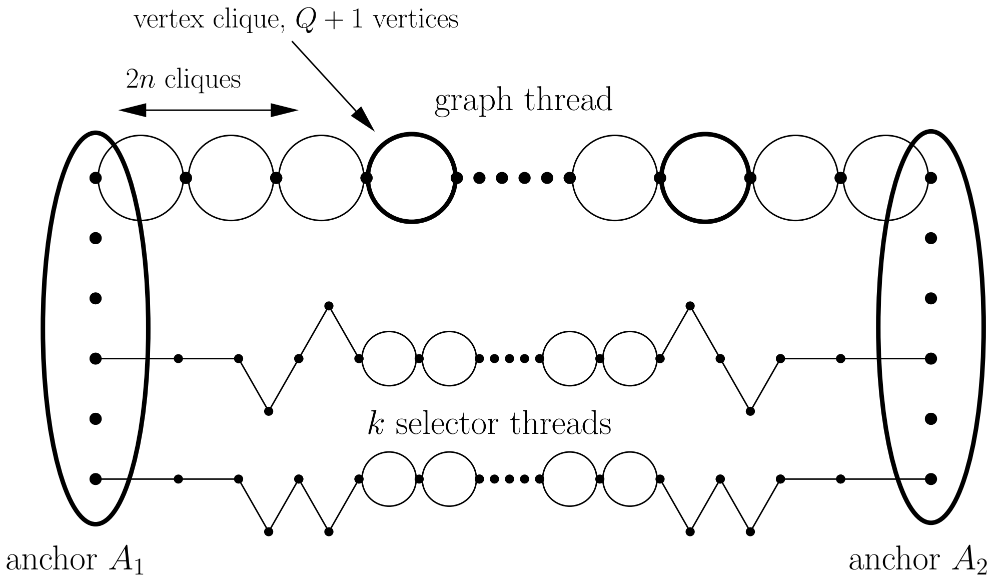


Figure 5.9: Gadget for domino pathwidth transformation.

$\Leftarrow$ Suppose G has a dominating set of size k. Then the required domino path decomposition of G', $PD(G')$, exists.

There are $P+2$ nodes in the decomposition, $\{X_0, \ldots, X_{P+1}\}$. We let A_1 be contained in X_0. For $1 \leq i \leq P$, we let X_i contain C^i from the graph thread, and we let the X_{P+1} contain A_2.

Note that X_1 must contain the first (size 2) clique of each of the selector threads, and X_P must contain the last (size 2) clique of each of the selector threads, by domino-ness. Each of these cliques has a vertex in common with the anchors, and this vertex can appear in only two nodes. It must appear in some node with its clique, and this cannot be the same node as the one where it appears with the anchor.

Suppose the dominating set of G is $\{v_{d_1}, v_{d_2}, \ldots, v_{d_k}\}$. We will align the first selector thread so that the v_{d_1}th clique of $\mathcal{S}_1$ in this thread appears in X_{2n+1}, the same node in which C^{2n+1}, the first vertex clique in the graph thread, appears. We will align the second selector thread so that the v_{d_2}th clique of $\mathcal{S}_1$ in this thread appears in X_{2n+1}, and so on.

For each selector thread, we place each of the S^i cliques, $1 \leq i \leq n^2$, one per node consecutively, but we "fold" the size 2 cliques at the start of each selector thread, by placing 2 of these at at time into a node (i.e. 3 vertices per node) as many times as necessary, so as to ensure that the v_{d_i}th clique of $\mathcal{S}_1$ in ith thread occurs in the same node as C^{2n+1} from the graph thread. The size 2 cliques at the end of each selector thread are also folded to fit into the nodes remaining before A_2 is reached in X_{P+1}.

Exactly $(n-1)$ folds will be required altogether, for each selector thread. The folding of the size 2 cliques will not breach the width bound, since each fold contributes 3 to the bag size, over at most k threads, and at most $k(k+3)$ is permitted i.e $k+3$ per thread. Allowing $(n-1)$ folds will ensure that any of $\{v_1, \ldots, v_n\}$ can be positioned correctly.

If we align the selector threads this way, then each node containing a vertex clique from the graph thread will also contain a clique from at least one selector thread that is of size only $(k+2)$. Let $C^{2n+j.n+1}$ be a vertex clique representing vertex v_j from G. If v_j is in the dominating set then one of the selector threads will be aligned so that the jth clique of $\mathcal{S}_j$ from that thread appears in $X_{2n+j.n+1}$, and this clique has size only $(k+2)$. If v_j is a neighbour of some vertex v_i in the dominating set then one of the selector threads will be aligned so that the ith clique of $\mathcal{S}_j$ from that thread appears in $X_{2n+j.n+1}$, and this clique has size only $(k+2)$.

The decomposition described here preserves domino-ness. X_0 and X_{P+1} each contain K vertices. Each interior node in the decomposition contains either a non-vertex clique in the graph thread along with at most $k(k+3)$ other vertices, or a vertex clique in the graph thread along with k cliques of size $(k+3)$ or $(k+2)$, where at least one of these cliques must have size $(k+2)$. Hence, each interior node contains at most K vertices

Thus, we have a domino path decomposition of width at most $K-1$, as required.

$\Rightarrow$ Suppose a domino path decomposition of G' with width K, $PD(G')$, exists, then G must have a dominating set of size k.

- Each of A_1 and A_2 must be the contained in an end node of $PD(G')$, since no threads can pass over these large cliques, and all threads have vertices in common with both of them. Let us assume that A_1 is contained in the first node, X_0.

- Only one clique from the graph thread can be contained in any node of $PD(G')$, since there is not enough room to admit more than one. Each clique of the graph thread must be contained in some node of $PD(G')$. By domino-ness, and a simple induction argument, we must have the same situation described in the first part of this proof. The decomposition $PD(G')$ must consist of $P+2$ nodes, A_1 is contained in the first node, X_0, A_2 is contained in the last node, X_{P+1}, and for $1 \leq i \leq P$, node X_i contains C^i from the graph thread.

- The first vertex clique in the graph thread must appear in a node containing a clique from $\mathcal{S}_1$ for each of the selector threads. The last vertex clique in the graph thread must appear in a node containing a clique from $\mathcal{S}_n$ for each of the the selector threads.

  The first vertex clique in the graph thread appears in node X_{2n+1}. The last vertex clique in the graph thread appears in node $X_{2n+n(n-1)+1}$. There are $2n+1$ nodes that occur before X_{2n+1} in the decomposition and $2n+1$ nodes that occur after $X_{2n+n(n-1)+1}$ in the decomposition. There are only $2n$ vertices occurring in a selector thread before the first clique of $\mathcal{S}_1$ is encountered. These are connected in a path, and by domino-ness, this path cannot stretch over more than $2n$ nodes. Similarly, the path at the end of the selector thread cannot stretch over more than $2n$ nodes.

- Each node in the decomposition, apart from the first and the last, contains a clique from the graph thread and so can contain at most k distinct S^i cliques from the selector threads.

Each clique from the graph thread contains $k^2(k+4)$ vertices and each S^i clique contains at least $(k+2)$ vertices. In any node there is room for only $k(k+3)$ more vertices apart from the graph thread clique. $(k+1)$ distinct S^i cliques will consist of $(k+1)(k+2) = k(k+3)+2$ vertices.

- The arguments given here, along with domino-ness, force the following situation.

  Every node in the decomposition from X_{2n+1}, which contains the first vertex clique of the graph thread, to $X_{2n+n(n-1)+1}$, which contains the last vertex clique of the graph thread, must contain exactly k S^i cliques, one from each of the selector threads. These must appear in the order in which they occur in the threads.

- For the width bound to be maintained, each node containing a vertex clique $C^{2n+j.n+1}$ from the graph thread must contain at least one S^i clique of size $(k+2)$. Thus, the $\mathcal{S}_1$ cliques that occur in node X_{2n+1} must correspond to k vertices that form a dominating set in G.

$\square$

Part II

Parameterized Counting Problems

CHAPTER 6
INTRODUCTION

Classical complexity is not only concerned with decision problems, but also with search, enumeration and counting problems. Parameterized complexity has, so far, been largely confined to consideration of computational problems as decision or search problems. However, it is becoming evident that the parameterized point of view can lead to new insight into counting problems. The aim of this part of the thesis is to introduce a formal framework in which to address issues of parameterized *counting complexity.*

Counting complexity is a very important branch of complexity, with many applications. It is also very hard. The consensus is that while decision problems can often have good approximation algorithms, it is generally thought that counting problems are very hard indeed (see, for example, [113]). Counting problems tend not to have approximation algorithms, randomized algorithms, PTAS's, or the like.

We think that parameterized complexity has a lot to say about the enumeration of small structures.

Arvind and Raman [9, 86] have modified the method of bounded search trees to show that the problem of counting all size k vertex covers of a graph $G = (V, E)$ is solvable in time $O(2^{k^2+k} + 2^k|V|)$. The approach used in [9] appears promising as a method for demonstrating parameterized tractability of many parameterized counting problems corresponding to well-studied decision problems.

Courcelle, Makowsky and Rotics [42] have considered counting and evaluation problems on graphs where the range of counting is definable in monadic second order logic. They show that these problems are fixed-parameter tractable, where the parameter is the treewidth of the graph. Andrzejak [6] , and Noble [82] have shown that for graphs $G = (V, E)$ of treewidth at most k, the Tutte polynomial can be computed in polynomial time and evaluated in time $O(|V|)$, despite the fact that the general problem is $\#P$-complete. Makowsky [74] has shown that the same bounds hold for the colored Tutte polynomial on coloured graphs $G = (V, E, c)$ of treewidth at most k. Grohe and Frick have introduced the notion of locally tree-decomposable classes of structures (see Section 2.3), and have shown that counting problems definable in first order logic can be solved in fixed-parameter linear time on such structures [54, 55].

Thus, it is clear that parameters can greatly alter the overall complexity of a problem, even in the realm of counting problems. All of the above have been ad hoc observations. In this part of the thesis we introduce a general framework for parameterized counting complexity, extending the framework introduced by Downey and Fellows for decision problems.

In Chapter 7 we review the structural framework developed by Downey and Fellows for parameterized decision problems.

In Chapter 8 we introduce basic definitions for tractability and the notion of a parameterized counting reduction, and we look at a basic hardness class. We define $\#W[1]$, the parameterized analog of Valiant's class $\#P$. Our core problem here is #SHORT TURING MACHINE ACCEPTANCE, where the input is a non-deterministic Turing machine M and a string x, the parameter is a positive integer k, and the output is the number of k-step accepting computations of M on x. We show that #SHORT TURING MACHINE ACCEPTANCE is complete for $\#W[1]$. We also determine $\#W[1]$-completeness, or $\#W[1]$-hardness, for several other parameterized counting problems. The material that we present has been published in [78]. We note that some of the results in Chapter 8 have been independently obtained by Grohe and Frick. They have recently shown that the problem of counting small cycles in a graph is $\#W[1]$-hard [58].

In Chapter 9 we present a normalization theorem, reworked from the framework developed by Downey and Fellows for decision problems. characterizing the $\#W[t]$, ($t \in \mathbb{N}$), parameterized counting classes.

It is important to note that parameterized hardness classes, and hardness results, are of more than just academic interest, they have real ramifications for classical complexity as well.

For example, Bazgan [12], (and independently Cesati and Trevisan [35]) showed that associated with every optimization problem is a parameterized decision problem. Moreover, if the optimization problem has an efficient PTAS then the parameterized decision problem is fixed-parameter tractable. Thus, an optimization problem has no efficient PTAS if the associated parameterized decision problem is $W[1]$-hard (unless the W-hierarchy collapses). Razborov and Alekhnovich [87] have shown that various axiom systems have no resolution proofs unless the W-hierarchy collapses.

We anticipate that the framework presented here will have similar applications in counting complexity. For example, if one could show that the (parameterized) problem of counting Euler cycles in graphs of bounded degree was $\#W[1]$-hard then this would show that there is likely no polynomial time method for solving the classical problem; even without considering the open question of whether this problem is $\#P$ complete.

CHAPTER 7
STRUCTURAL PARAMETERIZED COMPLEXITY

7.1 Introduction

As illustrated by the classical P vs NP scenario, there are two main ingredients necessary and sufficient for a framework in which we might hope to stratify computational problems into distinct but comparable degrees of complexity. The first is the identification of *classes* of problems that are, in some sensible sense, “equally hard”. The second is a notion of *reducibility* that allows us to demonstrate “hardness equality” or a “hardness ordering” between problems. The formulation of these two ingredients is completely intertwined, which comes first is very much a chicken and egg situation.

A class of problems can be defined by specifying some uniform manner in which the members of the class may be described. For such a definition to be useful at all, it must be the case that the class corresponds to a reasonably wide collection of natural problems, and that it relates to the reducibility notion employed in a meaningful way.

In the case of the classical complexity class NP, there are thousands of natural NP-complete problems, all of which can be couched in the same manner. We can construct uniformly structured transformations between these problems demonstrating that, if any of them were solvable in polynomial time, then all of them would be. We don’t have a proof that $P \neq NP$, but we have plenty of “sociological evidence” to suggest that $P = NP$ is unlikely in the extreme. The power of the theory comes from the interaction between the class definition and the notion of reducibility employed.

This is the main idea underpinning the development of structural parameterized complexity theory. In this chapter we review the important defintions and motivations of structural parameterized complexity, introduced by Downey and Fellows [48]. We begin with a notion of *good* complexity behaviour for a parameterized problem (language), *fixed-parameter tractability* (FPT).

Recall our standard definition, originally presented in Chapter 2.

Definition 7.1 (Fixed Parameter Tractability). *A parameterized language $L \subseteq \Sigma^* \times \Sigma^*$ is fixed-parameter tractable if there is an algorithm that correctly decides, for input $\langle \sigma, k \rangle \in \Sigma^* \times \Sigma^*$, whether $\langle \sigma, k \rangle \in L$ in time $f(k)n^\alpha$, where n is the size of the main part of the input σ, $|\sigma| = n$, k is the parameter, α is a constant (independent of k), and f is an arbitrary function.*

We need to define a notion of reducibility that expresses the fact that two parameterized problems have the same parameterized complexity. That is, if problem (language) A reduces to problem (language) B, and problem B is fixed-parameter tractable, then so too is problem A. We will need reductions that work in FPT time and that take parameters to parameters.

Here is our standard definition, again, originally presented in Chapter 2.

Definition 7.2 (Parameterized Transformation). *A parameterized transformation from a parameterized language L to a parameterized language L' is an algorithm that computes, from input consisting of a pair $\langle \sigma, k \rangle$, a pair $\langle \sigma', k' \rangle$ such that:*

1. *$\langle \sigma, k \rangle \in L$ if and only if $\langle \sigma', k' \rangle \in L'$,*
2. *$k' = g(k)$ is a function only of k, and*
3. *the computation is accomplished in time $f(k)n^\alpha$, where $n = |\sigma|$, α is a constant independent of both n and k, and f is an arbitrary function.*

As noted in Chapter 2, there are (at least) three different possible definitions of fixed-parameter tractability, depending on the level of uniformity desired. These relativize to different definitions of reducibility between problems. In this part of the thesis, we continue to use the standard working definitions given above, where the functions f and g, used in these definitions, are simple, computable functions. If we have a parameterized transformation from problem A to problem B, then we say that A and B are fpt-eqivalent.

Armed with these notions of good complexity behaviour and problem reducibility we now need to define what we mean by parameterized *intractability.* We need to determine a method of characterizing problems which (we believe) are *not* FPT. As for classical complexity theory, the best that parameterized complexity can offer is a completeness theory. It is important to note, however, that the theory is far more *fine-grained* than that of classical complexity theory. Herein lies a fundamental contribution of parameterized complexity to the structural complexity community at large.

7.2 The W-hierarchy

Our aim in this section is to explain the origins of the main sequence of parameterized complexity classes, commonly termed the W-hierarchy.

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[t] \cdots \subseteq W[P] \subseteq AW[P] \subseteq XP$$

We first review the original definition for $W[1]$, presented in [48]. We will need some preliminary definitions.

We consider a 3CNF formula as a circuit consisting of one input (of unbounded fanout) for each variable, possibly inverters below the variable, and structurally a large *and* of small *or*'s (of size 3) with a single output line. We can similarly consider a 4CNF formula to be a large *and* of small *or*'s where "small" is defined to be 4. More generally, it is convenient to consider the model of a *decision circuit*. This is a circuit consisting of large and small gates with a single output line, and no restriction on the fanout of gates. For such a circuit, the *depth* is the maximum number of gates on any path from the input variables to the output line, and the $weft$ is the "large gate depth." Formally, we define the weft of a circuit as follows:

Definition 7.3 (Weft). *Let C be a decision circuit. The $weft$ of C is defined to be the maximum number of large gates on any path from the input variables to the output line. A gate is called large if it's fanin exceeds some pre-determined bound.*

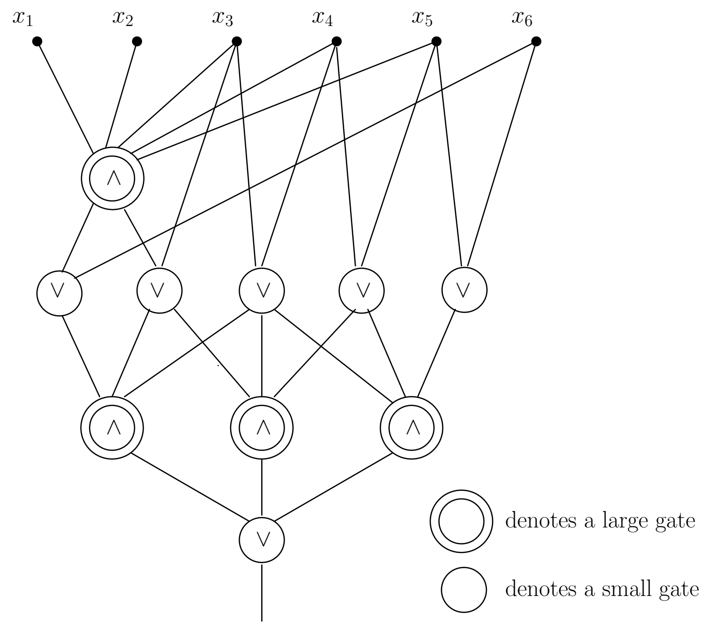


Figure 7.1: A weft 2, depth 4 decision circuit.

The *weight* of an assignment to the input variables of a decision circuit is the number of variables made true by the assignment. Let $\mathcal{F} = \{C_1, ..., C_n, ...\}$ be a family of decision circuits. Associated with $\mathcal{F}$ is a basic parameterized language

$$L_{\mathcal{F}} = \{\langle C_i, k\rangle : C_i \text{ has a weight } k \text{ satisfying assignment}\} .$$

Notation: We will denote by $L_{\mathcal{F}(t,h)}$ the parameterized language associated with the family of weft t depth h decision circuits.

Definition 7.4 ($W[1]$). *We define a language L to be in the class $W[1]$ iff there is a parametric transformation from L to $L_{\mathcal{F}(1,h)}$ for some h.* □

We now have a uniform manner in which to define the members of the class $W[1]$, but what makes us think that this is a good definition? Why should we believe that any $W[1]$-complete problem is not FPT?

Consider the basic generic NP-complete problem, identified by Cook [40].

NONDETERMINISTIC TURING MACHINE ACCEPTANCE

Instance: A nondeterministic Turing machine M, a string x, and a number n.

Question: Does M have a computation path accepting x in $\leq |M|^n$ steps?

A nondeterministic Turing machine is an opaque and generic object. It doesn't seem reasonable that we should be able to decide, for any given Turing machine, on any given input, whether the machine accepts, without essentially trying all paths. Hence, it doesn't seem reasonable that we should be able to solve this problem, in general, in polynomial time. Thus, Cook's theorem [40] provides us with powerful evidence that $P \neq NP$.

If we accept this idea, then we should also accept that the following parameterized problem is not solvable in time $O(|M|^c)$ for any fixed c, that is, not FPT, since our intuition would again be that all paths would need to be tried.

SHORT NDTM ACCEPTANCE

Instance:	A nondeterministic Turing machine M and a string x.
Parameter:	A positive integer k.
Question:	Does M have a computation path accepting x in $\leq k$ steps?

Downey and Fellows [48] have proved the following analog of Cook's theorem. The proof is quite technical and involves a number of steps. In Chapter 8 we prove a counting analog of this theorem.

Theorem 7.1 (Parameterized analog of Cook's theorem). *SHORT TURING MACHINE ACCEPTANCE is complete for* $W[1]$.

This theorem provides us with powerful evidence that $W[1] \neq FPT$.

Now consider the following parameterized problem:

WEIGHTED CNF SAT

Instance:	A CNF formula X.
Parameter:	A positive integer k.
Question:	Does X have a satisfying assignment of weight k?

Similarly, we can define WEIGHTED nCNF SAT, where the clauses have only n variables and n is some number fixed in advance. As a consequence of the proof of Theorem 7.1 we get the following theorem.

Theorem 7.2. *WEIGHTED nCNF SAT, for any fixed $n \geq 2$, is complete for* $W[1]$.

Classically, using a padding argument, we know that CNF SAT $\equiv_m^P$ 3 CNF SAT. However, the classical reduction *doesn't* define a parameterized transformation from

WEIGHTED CNF SAT to WEIGHTED 3CNF SAT, it is not structure-preserving enough to ensure that parameters map to parameters. In fact, Downey and Fellows conjecture that there is *no* parameterized transformation at all from WEIGHTED CNF SAT to WEIGHTED 3CNF SAT. If the conjecture is correct, then WEIGHTED CNF SAT is *not* in the class $W[1]$.

We can view the input formula, X, for WEIGHTED CNF SAT as a product of sums. Extending this reasoning, we can define WEIGHTED t-NORMALIZED SAT as the weighted satisfiability problem for a formula X where X is a product of sums of products of sums $\ldots$ with t alternations. We can define WEIGHTED SAT to be the weighted satisfiability problem for a formula X that is unrestricted.

By Theorem 7.2, $W[1]$ is the collection of parameterized languages fpt-equivalent to WEIGHTED 3CNF SAT.

Recall that $L_{\mathcal{F}(t,h)}$ is the parameterized language associated with the family of weft t depth h decision circuits.

Definition 7.5 ($W[t]$). *We define a language L to be in the class $W[t]$ iff there is a parameterized transformation from L to $L_{\mathcal{F}(t,h)}$ for some h.*

In [48] Downey and Fellows present the following theorem.

Theorem 7.3 (Normalization theorem). *For all $t \geq 1$, WEIGHTED t-NORMALIZED SAT is complete for $W[t]$.*

Thus, $W[2]$ is the collection of parameterized languages fpt-equivalent to WEIGHTED CNF SAT, and for each $t \geq 2$, $W[t]$ is the collection of parameterized languages fpt-equivalent to WEIGHTED t-NORMALIZED SAT. The proof of the theorem is again quite technical and involves a number of steps. In Chapter 9 we prove a counting analog of this theorem.

The class $W[SAT]$ is the collection of parameterized languages fpt-equivalent to WEIGHTED SAT. The class $W[P]$ is the collection of parameterized languages fpt-equivalent to WEIGHTED CIRCUIT SAT, the weighted satisfiability problem for a decision circuit C that is unrestricted.

A standard translation of Turing machines into circuits shows that k-WEIGHTED CIRCUIT SAT is the same as the problem of deciding whether or not a deterministic Turing machine accepts an input of weight k. Downey and Fellows conjecture that the containment $W[SAT] \subset W[P]$ is proper.

$AW[P]$ captures the notion of *alternation.* $AW[P]$ is the collection of parameterized languages fpt-equivalent to PARAMETERIZED QUANTIFIED CIRCUIT SATISFIABILITY, the weighted satisfiability problem for an unrestricted decision circuit that applies *alternating quantifiers* to the inputs, defined here.

PARAMETERIZED QC SAT

Instance:	A decison circuit C whose inputs correspond to a sequence $s_1, \ldots s_r$ of pairwise disjoint sets of variables.
Parameter:	$r, k_1, \ldots, k_n$.
Question:	Is it the case that there exists a size k_1 subset t_1 of s_1, such that for every size k_2 subset t_2 of s_2, there exists a size k_3 subset t_3 of s_3, such that ... (alternating quantifiers) such that, when $t_1 \cup t_2 \cup \ldots \cup t_r$ are set to true, and all other variables are set to false, C is satisfied?

XP is the collection of parameterized languages L such that the kth slice of L (the instances of L having parameter k) is complete for $DTIME(n^k)$. XP is provably distinct from FPT and seems to be the parameterized class corresponding to the classical class EXP (exponential time).

Each of these classes contains concrete natural problems. The collection of $W[1]$-complete problems includes CLIQUE, INDEPENDENT SET, SHORT POST CORRESPONDENCE, SQUARE TILING, SHORT DERIVATION for CONTEXT SENSITIVE GRAMMARS, as well as WEIGHTED n-CNF SATISFIABILITY and SHORT NONDETERMINISTIC TURING MACHINE ACCEPTANCE.

The collection of $W[2]$-complete problems includes DOMINATING SET (see Section 5.7) and TOURNAMENT DOMINATING SET. Concrete problems hard for $W[2]$ include UNIT LENGTH PRECEDENCE CONSTRAINED SCHEDULING, SHORTEST COMMON SUPERSEQUENCE, MAXIMUM LIKELIHOOD DECODING, WEIGHT DISTRIBUTION in LINEAR CODES, NEAREST VEC-

TOR in INTEGER LATTICES, and SHORT PERMUTATION GROUP FACTORIZATION.

Problems that are $W[t]$-hard, for all $t \geq 1$, include FEASIBLE REGISTER ASSIGNMENT, TRIANGULATING COLORED GRAPHS, BANDWIDTH (see Section 5.7), PROPER INTERVAL GRAPH COMPLETION [19], DOMINO TREEWIDTH [18], and BOUNDED PERSISTENCE PATHWIDTH (see Section 5.7).

The 3-DIMENSIONAL EUCLIDEAN GENERALIZED MOVER'S PROBLEM is $W[SAT]$-hard.

$W[P]$-complete problems include LINEAR INEQUALITIES, SHORT SATISFIABILITY, SHORT CIRCUIT SATISFIABILITY, and THRESHOLD STARTING SET.

COMPACT NONDETERMINISTIC TURING MACHINE COMPUTATION is $AW[P]$-hard.

The collection of XP-complete problems includes k-CAT AND MOUSE GAME, k-PEG GAME, and k-PEBBLE GAME.

For detailed descriptions of these problems, and related results, apart from those introduced in this thesis, see [48].

CHAPTER 8
$\#W[1]$ - A PARAMETERIZED COUNTING CLASS

8.1 Classical counting problems and $\#P$

The fundamental class of *counting problems* in classical complexity theory, $\#P$, was proposed by Valiant [112] in the late 1970's. His definition uses the notion of a witness function:

Definition 8.1 (Witness function). *Let $w : \Sigma^* \to \mathcal{P}(\Gamma^*)$, and let $x \in \Sigma^*$. We refer to the elements of $w(x)$ as witnesses for x. We associate a decision problem $A_w \subseteq \Sigma^*$ with w:*

$$A_w = \{ \, x \in \Sigma^* \mid w(x) \neq \emptyset \, \}.$$

In other words, A_w is the set of strings that have witnesses.

Definition 8.2 ($\#P$). *The class $\#P$ is the class of witness functions w such that:*

(i). there is a polynomial-time algorithm to determine, for given x and y, whether $y \in w(x)$;

(ii). there exists a constant $k \in \mathcal{N}$ such that for all $y \in w(x)$, $|y| \leq |x|^k$. (The constant k can depend on w).

$\#P$ is the class of witness functions naturally associated with decision problems in the class NP.

The counting problem associated with a particular computational problem is to determine the *number* of solutions to the problem for any given instance. That is, given the witness function associated with the decision problem, what we are really interested in is $|w(x)|$ for any input x.

This leads us to other definitions for $\#P$, where $\#P$ is treated as a class of functions of the form $f : \Sigma^* \to \mathcal{N}$. For instance, see [113], [83].

We now need to establish how counting problems v and w are related under the process of reduction. For this purpose we introduce the notions of *counting reduction* and *parsimonious reduction.*

Definition 8.3 (Counting reduction). *Let*

$$w : \Sigma^* \to \mathcal{P}(\Gamma^*)$$
$$v : \Pi^* \to \mathcal{P}(\Delta^*)$$

be counting problems, in the sense of [112]. A polynomial-time many-one counting reduction from w to v consists of a pair of polynomial-time computable functions

$$\sigma : \Sigma^* \to \Pi^*$$
$$\tau : \mathcal{N} \to \mathcal{N}$$

such that

$$|w(x)| = \tau(|v(\sigma(x))|).$$

When such a reduction exists we say that w reduces to v.

Intuitively, if one can easily count the number of witnesses of $v(y)$, then one can easily count the number of witnesses of $w(x)$.

There is a particularly convenient kind of reduction that preserves the number of solutions to a problem exactly. We call such a reduction *parsimonious.*

Definition 8.4 (Parsimonious reduction). *A counting reduction σ, τ is parsimonious if τ is the identity function.*

Armed with the notion of a counting reduction, we can define the class of $\#P$-*complete* problems.

One famous result is the following:

Theorem 8.1 (Valiant). *The problem of counting the number of perfect matchings in a bipartite graph is $\#P$-complete.*

Despite the fact that a perfect matching can be found in polynomial time, counting the number of them is as hard as counting the number of satisfying assignments to a Boolean formula. Details of the proof of this theorem may be found in [69] or [83].

8.2 Definitions for parameterized counting complexity

In order build a framework in which to consider parameterized counting problems, we first need to establish a some basic definitions.

As in the classical case, we use the notion of a witness function to formalize the association between parameterized counting problems and their corresponding decision problems.

Definition 8.5 (Parameterized witness function). *Let $w : \Sigma^* \times \mathcal{N} \to \mathcal{P}(\Gamma^*)$, and let $\langle \sigma, k \rangle \in \Sigma^* \times \mathcal{N}$. The elements of $w(\langle \sigma, k \rangle)$ are witnesses for $\langle \sigma, k \rangle$. We associate a parameterized language*

$$L_w \subseteq \Sigma^* \times \mathcal{N} \text{ with} w$$

:

$L_w = \{ \langle \sigma, k \rangle \in \Sigma^* \times \mathcal{N} \mid w(\langle \sigma, k \rangle) \neq \emptyset \}$.
L_w is the set of problem instances that have witnesses.

Definition 8.6 (Parameterized counting problem). *Let $w : \Sigma^* \times \mathcal{N} \to \mathcal{P}(\Gamma^*)$ be a parameterized witness function. The corresponding parameterized counting problem can be considered as a function $f_w : \Sigma^* \times \mathcal{N} \to \mathcal{N}$ that, on input $\langle \sigma, k \rangle$, outputs $|w(\langle \sigma, k \rangle)|$.*

We note here that "easy" parameterized counting problems can be considered to be those in the class that we might call "FFPT", the class of functions of the form $f : \Sigma^* \times \mathcal{N} \to \mathcal{N}$ where $f(\langle \sigma, k \rangle)$ is computable in time $g(k)|\sigma|^\alpha$, where g is an arbitrary function and α is a constant not depending on k.

To consider "hard" parameterized counting problems, we need some more definitions:

Definition 8.7 (Parameterized counting reduction). *Let*

$$w : \Sigma^* \times \mathcal{N} \to \mathcal{P}(\Gamma^*)$$
$$v : \Pi^* \times \mathcal{N} \to \mathcal{P}(\Delta^*)$$

be (witness functions for) parameterized counting problems. A parameterized counting reduction from w to v consists of a parameterized transformation

$$\rho \ : \ \Sigma^* \times \mathcal{N} \to \Pi^* \times \mathcal{N}$$

and a function

$$\tau \ : \ \mathcal{N} \to \mathcal{N}$$

running in time $f(k)n^\alpha$ (where $n = |\sigma|$, α is a constant independent of both f and k, and f is an arbitrary function) such that

$$|w(\langle\sigma, k\rangle)| = \tau(|v(\rho(\langle\sigma, k\rangle))|).$$

When such a reduction exists we say that w reduces to v.

As in the classical case, if one can easily count the number of witnesses of $v(\langle\sigma', k'\rangle)$, then one can easily count the number of witnesses of $w(\langle\sigma, k\rangle)$.

Definition 8.8 (Parsimonious parameterized counting reduction). *A parameterized counting reduction ρ, τ is parsimonious if τ is the identity function.*

8.2.1 A definition for $\#W[1]$

Consider the following generalized parameterized counting problem:

#WEIGHTED WEFT t DEPTH h CIRCUIT SATISFIABILITY (WCS(t, h))

Input:	A weft t depth h decision circuit C.
Parameter:	A positive integer k.
Output:	The number of weight k satisfying assignments for C.

Let $w_{\mathcal{F}(t,h)} : \Sigma^* \times \mathcal{N} \to \mathcal{P}(\Gamma^*)$ be the standard parameterized witness function associated with this counting problem:

$$w_{\mathcal{F}(t,h)}(\langle C, k\rangle) = \{ \text{ weight } k \text{ satisfying assignments for } C \ \} \ .$$

Definition 8.9 ($\#W[1]$). *We define a parametric counting problem, f_v, to be in $\#W[1]$ iff there is a parameterized counting reduction from v, the parameterized witness function for f_v, to $w_{\mathcal{F}(1,h)}$.*

8.3 A fundamental complete problem for $\#W[1]$

We have defined what it means for a parameterized counting problem to be "easy" or tractable, via the class "FFPT" of FPT-time computable functions. We have made a start on establishing a completeness program to exhibit (likely) intractability of "hard" parameterized counting problems, by introducing the notion of a parameterized counting reduction and a definition for the class $\#W[1]$. Now, we need to show that our definition for $\#W[1]$ is a sensible one.

Consider the following fundamental parameterized counting problem.

#SHORT TURING MACHINE ACCEPTANCE

Input:	A nondeterministic Turing machine M, and a string x.
Parameter:	A positive integer k.
Output:	$acc_M(x, k)$, the number of $\leq k$-step accepting computations of machine M on input x.

In this section we prove the following theorem:

Theorem 8.2. *#SHORT TURING MACHINE ACCEPTANCE is complete for $\#W[1]$.*

The point here is that we are proving a parameterized counting analog of Cook's theorem. For example, the $W[1]$-completeness of INDEPENDENT SET means that deciding whether a graph has an independent set of size k is as hard as deciding whether a non-deterministic Turing machine has an accepting path of length k on some input. We aim to show that for $\#W[1]$-complete problems *counting* the number of solutions is as hard as counting the number of "short" accepting paths of a non-deterministic Turing machine on some input.

The proof makes use of a series of parameterized transformations described in [48], with alterations where required to ensure that each of these may be considered as a parsimonious parameterized counting reduction. The most important alteration that we make is the reworking of the final step in the proof of Lemma 8.1. Most of the transformations described in [48] are, in fact, parsimonious, we just need to argue that this is the case.

We first outline the structure of the proof, then present full proofs for each of the key steps separately.

We begin with some preliminary *normalization* results about weft 1 circuits.

We turn our attention to circuits having depth 2 and a particularly simple form, consisting of a single output *and* gate which receives arguments from *or* gates having fanin bounded by a constant s. Each such circuit is isomorphically represented by a boolean expression in conjunctive normal form having clauses with at most s literals. We will say that a circuit having this form is *s-normalized.*

Let $F(s)$ denote the family of s-normalized circuits. Note that an s-normalized circuit is a weft 1, depth 2 decision circuit, with the *or* gates on level 1 having fanin bounded by s.

We show that there is a parsimonious parameterized counting reduction from $w_{\mathcal{F}(1,h)}$, the standard parameterized witness function for $\mathcal{F}(1,h)$, to $w_{\mathcal{F}(s)}$, the standard parameterized witness function for $\mathcal{F}(s)$, where $s = 2^h + 1$. Thus, any parametric counting problem $f_v \in \#W[1]$ can, in fact, be reduced to the following problem (where s is fixed in advance and depends on f_v):

#WEIGHTED s-NORMALIZED CIRCUIT SATISFIABILITY

Input:	An s-normalized decision circuit C.
Parameter:	A positive integer k.
Output:	The number of weight k satisfying assignments for C.

Lemma 8.1. *$w_{\mathcal{F}(1,h)}$ reduces to $w_{\mathcal{F}(s)}$, where $s = 2^h + 1$, via a parsimonious parametric counting reduction.*

Proof outline:

Let $C \in \mathcal{F}(1,h)$ and let k be a positive integer. We describe a parameterized transformation that, on input $\langle C, k \rangle$, produces a circuit $C' \in \mathcal{F}(s)$ and an integer

k' such that for every weight k input accepted by C there exists a unique weight k' input accepted by C'. The transformation proceeds in four stages, and follows that given in [48], with substantial alterations to the last step in order to ensure parsimony.

The first three steps culminate in the production a tree circuit, C', of depth 4, that corresponds to a Boolean expression, E, in the following form. (We use product notation to denote logical $\wedge$ and sum notation to denote logical $\vee$.)

$$E = \prod_{i=1}^{m} \sum_{j=1}^{m_i} E_{ij}$$

where:
(1) m is bounded by a function of h,
(2) for all i, m_i is bounded by a function of h,
(3) for all i, j, E_{ij} is either:

$$E_{ij} = \prod_{k=1}^{m_{ij}} \sum_{l=1}^{m_{ijk}} x[i,j,k,l]$$

or

$$E_{ij} = \sum_{k=1}^{m_{ij}} \prod_{l=1}^{m_{ijk}} x[i,j,k,l],$$

where the $x[i,j,k,l]$ are literals (i.e., input Boolean variables or their negations) and for all i, j, k, m_{ijk} is bounded by a function of h. The family of circuits corresponding to these expressions has weft 1, with the large gates corresponding to the E_{ij}. (In particular, the m_{ij} are *not* bounded by a function of h.)

Note that any weight k input accepted by C will also be accepted by C', any weight k input rejected by C will also be rejected by C'. Thus, the witnesses for C' are exactly the witnesses for our original circuit C.

In the fourth step, we employ additional nondeterminism.

Let C denote the normalized depth 4 circuit received from the previous step, corresponding to the Boolean expression E described above.

We produce an expression E' in product-of-sums form, with the size of the sums bounded by $2^h + 1$, that has a unique satisfying truth assignment of weight

$$k' = k + (k+1)(1+2^h)2^{2^h} + m + \sum_{i=1}^{m} m_i$$

corresponding to each satisfying truth assignment of weight k for C.

The idea is to employ extra variables in E' so that τ', a weight k' satisfying truth assignment for E', encodes both τ, a weight k satisfying truth assignment for E and a "proof" that τ satisfies E. Thus, τ' in effect guesses τ and also checks that τ satisfies E.

We build E' so that the *only* weight k' truth assignments that satisfy E' are the τ''s that correspond to τ's satisfying E.

Details of the construction of E', and the correspondence of a weight k satisfying assignment for E to a unique weight k' satisfying assignment for E', can be found in Section 8.3.1. □

Lemma 8.1 allows us to now state the following theorem:

Theorem 8.3.

$$\#W[1] = \bigcup_{s=1}^{\infty} \#W[1,s]$$

where $\#W[1,s]$ is the class of parametric counting problems whose associated witness functions reduce to $w_{\mathcal{F}(s)}$, the standard parameterized witness function for $\mathcal{F}(s)$, the family of s-normalized decision circuits.

We now want to show that $\#W[1]$ collapses to $\#W[1,2]$.

We will need some more definitions:

A circuit C is termed *monotone* if it does not have any *not* gates. Equivalently, C corresponds to a boolean expression having only positive literals.

We define a circuit C to be *antimonotone* if all the input variables are negated and the circuit has no other inverters. Thus, in an antimonotone circuit, each fanout line from an input node goes to a *not* gate (and in the remainder of the circuit there are *no* other *not* gates). The restriction to families of antimonotone circuits yields the class of parametric counting problems #ANTIMONOTONE $W[1,s]$.

Theorem 8.4. $\#W[1,s] =$ #ANTIMONOTONE $W[1,s]$ *for all* $s \geq 2$.

Using theorem 8.4 we can prove the following:

Theorem 8.5. $\#W[1] = \#W[1,2]$.

Theorem 8.6. *The following are complete for* $\#W[1]$*:*

(i) #INDEPENDENT SET

(ii) #CLIQUE

Proofs of these theorems can be found in Sections 8.3.2, 8.3.3, and 8.3.4.

We are now in a position to establish Theorem 8.2, which we restate here.

Theorem 8.2. #SHORT TURING MACHINE ACCEPTANCE is complete for $\#W[1]$.

Proof outline:

To show hardness for $\#W[1]$ we provide a parsimonious parametric counting reduction from #CLIQUE. Again, this follows [48] with a slight alteration. $\#W[1]$-hardness will then follow by Theorem 8.6 (ii).

To show membership in $\#W[1]$ we describe a parsimonious parametric counting reduction from #SHORT TURING MACHINE ACCEPTANCE to #WEIGHTED WEFT 1 DEPTH h CIRCUIT SATISFIABILITY (#WCS$(1,h)$). Details can be found in Section 8.3.5. □

We can now state an alternative definition for $\#W[1]$.

Definition 8.10 ($\#W[1]$, Turing machine characterization.). *Let* $acc_{[M,k]}(x)$ *be the number of* k*-step accepting computations of machine* M *on input* $x \in \Sigma^*$*. Then*

$$\#W[1] = \{f : \Sigma^* \to \mathcal{N} \mid f = acc_{[M,k]} \textit{ for some NP machine } M\}\ .$$

8.3.1 Proof of Lemma 8.1

Lemma. $w_{\mathcal{F}(1,h)}$ reduces to $w_{\mathcal{F}(s)}$, where $s = 2^h + 1$, via a parsimonious parametric counting reduction.

Proof:

Let $C \in \mathcal{F}(1, h)$ and let k be a positive integer. We describe a parametric transformation that, on input $\langle C, k \rangle$, produces a circuit $C' \in \mathcal{F}(s)$ and an integer k' such that for every weight k input accepted by C there exists a unique weight k' input accepted by C'. The transformation follows that given in [48], with alterations to **Step 4** to ensure parsimony.

Step 1. The reduction to tree circuits.

The first step is to transform C into an equivalent weft 1 *tree circuit*, C', of depth at most h. In a tree circuit every logic gate has fanout one, thus the circuit can be viewed as equivalent to a Boolean formula. The transformation can be accomplished by replicating the portion of the circuit above a gate as many times as the fanout of the gate, beginning with the top level of logic gates, and proceeding downward level by level. The creation of C' from C may require time $O(|C|^{O(h)})$ and involve a similar blow-up in the size of the circuit. This is permitted under our rules for parametric transformations since the circuit depth, h, is a fixed constant independent of k and $|C|$.

Note that any weight k input accepted by C will also be accepted by C', any weight k input rejected by C will also be rejected by C'. Thus, the witnesses for C are exactly the witnesses for C'.

Step 2. Moving the *not* gates to the top of the circuit.

Let C denote the circuit we receive from the previous step. Transform C into an equivalent circuit C' by commuting the *not* gates to the top, using DeMorgan's laws. This may increase the size of the circuit by at most a constant factor. The new tree circuit C' thus consists (at the top) of the input nodes with *not* gates on some of the lines fanning out from the inputs. In counting levels we consider all of this as level 0.

Note that, as was the case for step 1, any weight k input accepted by C will also be accepted by C', any weight k input rejected by C will also be rejected by C'. Thus, the witnesses for C are exactly the witnesses for C'.

Step 3. A preliminary depth 4 normalization.

The goal of this step is to produce a tree circuit, C', of depth 4, that corresponds to a Boolean expression, E, in the following form. (We use product notation to denote logical $\wedge$ and sum notation to denote logical $\vee$.)

$$E = \prod_{i=1}^{m} \sum_{j=1}^{m_i} E_{ij}$$

where:

(1) m is bounded by a function of h,

(2) for all i, m_i is bounded by a function of h,

(3) for all i, j, E_{ij} is either:

$$E_{ij} = \prod_{k=1}^{m_{ij}} \sum_{l=1}^{m_{ijk}} x[i,j,k,l]$$

or

$$E_{ij} = \sum_{k=1}^{m_{ij}} \prod_{l=1}^{m_{ijk}} x[i,j,k,l],$$

where the $x[i,j,k,l]$ are literals (i.e., input Boolean variables or their negations) and

(4) for all i, j, k, m_{ijk} is bounded by a function of h.

The family of circuits corresponding to these expressions has weft 1, with the large gates corresponding to the E_{ij}. (In particular, the m_{ij} are *not* bounded by a function of h.)

Let C denote the tree circuit received from Step 2, with all *not* gates at level 0. Let g denote a large gate in C. An input to g is computed by a subcircuit with depth bounded by h consisting only of small gates, and so is a function of at most 2^h literals. This subcircuit can be replaced, at constant cost, by either a product-of-sums expression (if g is a large $\wedge$ gate), or a sum-of-products expression (if g is a large $\vee$ gate). In the first case, the product of these replacements over all inputs to g yields the subexpression E_{ij} corresponding to g. In the second case, the sum of these replacements yields the corresponding E_{ij}.

The output of C is a function of the outputs of at most 2^h large gates, since the tree of small gates lying below all large gates in C has depth bounded by h. This function can be expressed as a product-of-sums expression of size at most 2^{2^h}. At the cost of possibly duplicating some of the large gate subcircuits at most 2^{2^h} times, we can achieve the desired normal form with the bounds: $m \leq 2^{2^h}$, $m_i \leq 2^h$ and

$m_{ijk} \leq 2^h$.

Note that any weight k input accepted by C will also be accepted by C', any weight k input rejected by C will also be rejected by C'. Thus, the witnesses for C are exactly the witnesses for C'.

Step 4. Employing additional nondeterminism.

Let C denote the normalized depth 4 circuit received from Step 3 and corresponding to the Boolean expression E described above.

In this step we produce an expression E' in product-of-sums form, with the size of the sums bounded by $2^h + 1$, that has a unique satisfying truth assignment of weight

$$k' = k + (k+1)(1+2^h)2^{2^h} + m + \sum_{i=1}^{m} m_i$$

corresponding to each satisfying truth assignment of weight k for C.

The idea is to employ extra variables in E' so that τ', a weight k' satisfying truth assignment for E', encodes both τ, a weight k satisfying truth assignment for E and a "proof" that τ satisfies E. Thus, τ' in effect guesses τ and also checks that τ satisfies E.

For convenience, assume that the E_{ij} for $j = 1, ..., m'_i$ are sums-of-products and the E_{ij} for $j = m'_i + 1, ..., m_i$ are products-of-sums. Let $V_0 = \{x_1, ..., x_n\}$ denote the variables of E.

The set V of variables of E' is $V = V_0 \cup V_1 \cup V_2 \cup V_3 \cup V_4$, where

$$\begin{aligned}
V_0 = &\ \{x_1, ..., x_n\} \text{ the variables of } E.\\
V_1 = &\ \{\, x[i,j] :\ 1 \leq i \leq n,\ 0 \leq j \leq (1+2^h)2^{2^h}\}\\
V_2 = &\ \{\, u[i,j] :\ 1 \leq i \leq m,\ 1 \leq j \leq m_i\}\\
V_3 = &\ \{\, w[i,j,k] :\ 1 \leq i \leq m,\ 1 \leq j \leq m'_i,\ 0 \leq k \leq m_{ij}\}\\
V_4 = &\ \{\, v[i,j,k] :\ 1 \leq i \leq m,\ m'_i + 1 \leq j \leq m_i,\ 0 \leq k \leq m_{ij}\}
\end{aligned}$$

We first describe how we intend to extend a weight k truth assignment for V_0 that satisfies E, into a unique weight k' truth assignment for V.

Suppose τ is a weight k truth assignment for V_0 that satisfies E, we build τ' as follows:

1. For each i such that $\tau(x_i) = 1$ and for each j, $0 \leq j \leq (1 + 2^h)2^{2^h}$ set $\tau'(x[i, j]) = 1$.

2. For each i, $1 \leq i \leq m$, choose the lexicographically least index j_i such that E_{i,j_i} evaluates to 1 (this is possible, since τ satisfies E) and set $\tau'(u[i, j_i]) = 1$.

   (a) If, in (2), E_{i,j_i} is a sum-of-products, then choose the lexicographically least index k_i such that

   $$\prod_{l=1}^{m_{i,j_i,k_i}} x[i, j_i, k_i, l]$$

   evaluates to 1, and set $\tau'(w[i, j_i, k_i]) = 1$.

   Here, we are choosing the first input line to the sum E_{i,j_i} that makes it true.

   (b) If, in (2), E_{i,j_i} is a product-of-sums, then for each product-of-sums $E_{i,j'}$ with $j' < j_i$, choose the lexicographically least index k_i such that

   $$\prod_{l=1}^{m_{i,j',k_i}} \neg x[i, j', k_i, l]$$

   evaluates to 1, and set $\tau'(v[i, j', k_i]) = 1$.

   Here, for each product $E_{i,j'}$ that precedes E_{i,j_i}, we are choosing the first input line that makes $E_{i,j'}$ false.

3. For $i = 1, ..., m$ and $j = 1, ..., m'_i$ such that $j \neq j_i$, set $\tau'(w[i, j, 0]) = 1$.

   For $i = 1, ..., m$ and $j = m'_i + 1, ..., m_i$ such that $j \geq j_i$, set $\tau'(v[i, j, 0]) = 1$.

4. For all other variables, v, of V, set $\tau'(v) = 0$.

Note that τ' is a truth assignment to the variables of V having weight

$$k' = k + (k + 1)(1 + 2^h)2^{2^h} + m + \sum_{i=1}^{m} m_i$$

and that there is exactly one τ' for V corresponding to each τ for V_0.

We now build E' so that the *only* weight k' truth assignments that satisfy E' are the τ''s that correspond to τ's satisfying E.

The expression E' is a product of subexpressions: $E' = E_1 \wedge \cdots \wedge E_{9c}$.

We first want to associate with each x_i, a variable from V_0, the block of variables $x[i,j] : 0 \leq j \leq (1+2^h)2^{2^h}$ from V_1. We want to ensure that if x_i is set to 1 then all of the associated $x[i,j]$ are also set to 1, and, that if x_i is set to 0 then all of the associated $x[i,j]$ are set to 0. The following two subexpressions accomplish this task.

$$E_1 = \prod_{i=1}^{n} (\neg x_i + x[i,0])(\neg x[i,0] + x_i)$$

$$E_2 = \prod_{i=1}^{n} \prod_{j=0}^{2^{2^h}+1} (\neg x[i,j] + x[i,j+1 \pmod{r}]) \qquad r = 1 + (1+2^h)2^{2^h}$$

For each i, $1 \leq i \leq m$, j, $1 \leq j \leq m_i$, if we set $u[i,j] = 1$ then this represents the fact that E_{ij} is satisfied. We need to ensure that the collection of $u[i,j]$'s set to 1 represents the satisfaction of E, our original expression over V_0.

$$E_3 = \prod_{i=1}^{m} \sum_{j=1}^{m_i} u[i,j]$$

For each i, $1 \leq i \leq m$, we must ensure that we set exactly one variable $u[i,j_i]$ to 1, and that j_i is the lexicographically least index such that E_{i,j_i} evaluates to 1.

Exactly one j_i for each i:

$$E_4 = \prod_{i=1}^{m} \prod_{j=1}^{m_i-1} \prod_{j'=j+1}^{m_i} (\neg u[i,j] + \neg u[i,j'])$$

If E_{ij} is a sum-of-products and $u[i,j] = 1$ then we do not set the default, $w[i,j,0]$. Thus, it must be the case that some $w[i,j,k], k \neq 0$ is able to be chosen, and E_{ij} must be satisfied by τ:

$$E_5 = \prod_{i=1}^{m} \prod_{j=1}^{m'_i} (\neg u[i,j] + \neg w[i,j,0])$$

To make the reduction parsimonious we must also ensure that if E_{ij} is a sum-of-products and $u[i,j] = 0$ then we *do* set the default, $w[i,j,0]$:

$$E_{5a} = \prod_{i=1}^{m} \prod_{j=1}^{m'_i} (u[i,j] + w[i,j,0])$$

If E_{ij} is a product-of-sums and $u[i,j] = 1$ then E_{ij} is satisfied by τ:

$$E_6 = \prod_{i=1}^{m} \prod_{j=m'_i+1}^{m_i} \prod_{k=1}^{m_{ij}} (\neg u[i,j] + \sum_{l=1}^{m_{ijk}} x[i,j,k,l)$$

To make the reduction parsimonious, for each i and j where E_{ij} is a product-of-sums, we must restrict the $v[i,j,k]$ chosen. If we set $u[i,j] = 1$, then for all $E_{ij'}$, $j' \geq j$, that are product-of-sums we set the default $v[i,j,0]$, for all $E_{ij'}$, $j' < j$, that are product-of-sums we do not set the default:

$$E_{6a} = \prod_{i=1}^{m} \prod_{j=1}^{m'_i} \prod_{j'=m'_i+1}^{m_i} (\neg u[i,j] + v[i,j',0])$$

$$E_{6b} = \prod_{i=1}^{m} \prod_{j=m'_i+1}^{m_i} \prod_{j'=j}^{m_i} (\neg u[i,j] + v[i,j',0])$$

If $u[i,j] = 1$ then all other choices $u[i,j']$ with $j' < j$ must have Eij' not satisfied by τ. In the case of sum-of-products, we ensure directly that none of the input lines can evaluate to 1, in the case of product-of-sums, we ensure that the default variable $v[i,j',0] \neq 1$, therefore forcing some $v[i,j',k]$ with $k \neq 0$ to be set to 1:

$$E_{7a} = \prod_{i=1}^{m} \prod_{j=1}^{m'_i} \prod_{j'=1}^{j-1} \prod_{k=1}^{m_{ij'}} (\neg u[i,j] + \sum_{l=1}^{m_i j' k} \neg x[i,j',k,l])$$

$$E_{7b} = \prod_{i=1}^{m} \prod_{j=m'_i+1}^{m_i} \prod_{j'=1}^{m'_i} (\neg u[i,j] + \sum_{l=1}^{m_i j' k} \neg x[i,j',k,l])$$

$$E_{7c} = \prod_{i=1}^{m} \prod_{j=m'_i+1}^{m_i} \prod_{j'=m'_i+1}^{j-1} (\neg u[i,j] + \neg v[i,j',0])$$

For each $E_{i,j}$ that is a sum-of-products, we must set exactly one variable $w[i,j,k]$ to 1. If we set $w[i,j,k_i] = 1$, where $k_i \neq 0$, then we need to ensure that k_i is the lexicographically least index such that

$$\prod_{l=1}^{m_{i,j,k_i}} x[i,j,k,l]$$

evaluates to 1.

$$E_{8a} = \prod_{i=1}^{m} \prod_{j=1}^{m'_i} \prod_{k=0}^{m_{ij}-1} \prod_{k'=k+1}^{m_{ij}} (\neg w[i,j,k] + \neg w[i,j,k'])$$

$$E_{8b} = \prod_{i=1}^{m} \prod_{j=1}^{m'_i} \prod_{k=1}^{m_{ij}} \prod_{l=1}^{m_{ijk}} (\neg w[i,j,k] + x[i,j,k,l])$$

$$E_{8c} = \prod_{i=1}^{m} \prod_{j=1}^{m'_i} \prod_{k=1}^{m_{ij}} \prod_{k'=1}^{k-1} (\neg w[i,j,k] + \sum_{l=1}^{m_{ijk'}} \neg x[i,j,k',l])$$

For each $E_{i,j}$ that is a product-of-sums, we must set exactly one variable $v[i,j,k]$ to 1. If we set $v[i,j,k_i] = 1$, where $k_i \neq 0$, then we need to ensure that k_i is the lexicographically least index such that

$$\prod_{l=1}^{m_{i,j,k_i}} \neg x[i,j,k,l]$$

evaluates to 1.

$$E_{9a} = \prod_{i=1}^{m} \prod_{j=m'_i+1}^{m_i} \prod_{k=0}^{m_{ij}-1} \prod_{k'=k+1}^{m_{ij}} (\neg v[i,j,k] + \neg v[i,j,k'])$$

$$E_{9b} = \prod_{i=1}^{m} \prod_{j=m'_i+1}^{m_i} \prod_{k=1}^{m_{ij}} \prod_{l=1}^{m_{ijk}} (\neg v[i,j,k] + \neg x[i,j,k,l])$$

$$E_{9c} = \prod_{i=1}^{m} \prod_{j=m'_i+1}^{m_i} \prod_{k=1}^{m_{ij}} \prod_{k'=1}^{k-1} (\neg v[i,j,k] + \sum_{l=1}^{m_{ijk'}} x[i,j,k',l])$$

Let C be the weft 1, depth 4, circuit corresponding to E. Let C' be the s-normalized circuit, with $s = 2^h + 1$, corresponding to E'.

We now argue that the transformation described here, $\langle C, k\rangle \rightarrow \langle C', k'\rangle$, is, in fact, a parsimonious parametric counting reduction.

If τ is a witness for $\langle C, k\rangle$ (that is, τ is a weight k truth assignment for V_0 that satisfies E), then τ' is a witness for $\langle C', k'\rangle$ (that is, τ' is a weight k' truth assignment for V that satisfies E'). This is evident from the description of E' and τ'.

In addition, we argue that τ' is the only extension of τ that is a witness for $\langle C', k'\rangle$.

Suppose τ' is an extension of τ, and that τ is a witness for $\langle C, k\rangle$:

1. The clauses of E_1 and E_2 ensure that for each i such that $\tau(x_i) = 1$ and for each j, $0 \leq j \leq (1 + 2^h)2^{2^h}$ we must have $\tau'(x[i,j]) = 1$.

2. The clause E_4 ensures that, for each i, $1 \leq i \leq m$, we must choose exactly one index j_i such that E_{i,j_i} evaluates to 1 and set $\tau'(u[i,j_i]) = 1$, and we must set $\tau'(u[i,j']) = 0$ for all $j' \neq j_i$. The clauses E_{7a}, E_{7b}, and E_{7c} ensure that j_i is the lexicographically least possible index.

   (a) If, in (2), E_{i,j_i} is a sum-of-products, then the clauses E_5 and $E_{8a}, \ldots, E_{8c}$ ensure that we must choose the lexicographically least index $k_i \neq 0$ such that

   $$\prod_{l=1}^{m_{i,j_i,k_i}} x[i,j_i,k_i,l]$$

   evaluates to 1, and set $\tau'(w[i,j_i,k_i]) = 1$, and we must set $\tau'(w[i,j_i,k']) = 0$ for all $k' \neq k_i$.

Clauses E_{5a} and E_{8a} ensure that for $j = 1, ..., m'_i$ such that $j \neq j_i$, we must set $\tau'(w[i,j,0]) = 1$, and $\tau'(w[i,j,k]) = 0$ for all $k \neq 0$.

Clauses E_{6a} and E_{9a} ensure that for $j = m'_i + 1, ..., m_i$ we must set $\tau'(v[i,j,0]) = 1$, and $\tau'(v[i,j,k]) = 0$ for all $k \neq 0$.

(b) If, in (2), E_{i,j_i} is a product-of-sums, then clauses E_{7c} and $E_{9a}, \ldots, E_{9c}$ ensure that for each product-of-sums $E_{i,j'}$ with $j' < j_i$, we must choose the lexicographically least index $k_i \neq 0$ such that

$$\prod_{l=1}^{m_{i,j',k_i}} \neg x[i,j',k_i,l]$$

evaluates to 1, and set $\tau'(v[i,j',k_i]) = 1$, and that for all $k' \neq k_i$ we must set $\tau'(v[i,j',k']) = 0$.

Clauses E_{6b} and E_{9a} ensure that for all $j = m'_i+1, ..., m_i$ such that $j \geq j_i$, we must set $\tau'(v[i,j,0]) = 1$, and $\tau'(v[i,j,k]) = 0$ for all $k \neq 0$.

Clauses E_{5a} and E_{8a} ensure that for $j = 1, ..., m'_i$ we must set $\tau'(w[i,j,0]) = 1$, and $\tau'(w[i,j,k]) = 0$ for all $k \neq 0$.

Now suppose v' is witness for $\langle C', k' \rangle$. We argue that the restriction v of v' to V_0 is a witness for $\langle C, k \rangle$.

Claim 1. v sets at most k variables of V_0 to 1.

If this were not so, then the clauses in E_1 and E_2 would together force at least $(k+1)+(k+2)(1+2^h)2^{2^h})$ variables to be 1 in order for v' to satisfy E', a contradiction as this is more than k'.

Claim 2. v sets at least k variables of V_0 to 1.

The clauses of E_{7a} ensure that v' sets at most m variables of V_2 to 1. The clauses of E_{8a} ensure that v' sets at most $\sum_{i=1}^{m} m'_i$ variables of V_3 to 1. The clauses of E_{9a} ensure that v' sets at most $\sum_{i=1}^{m} m_i - m'_i$ variables of V_4 to 1.

If Claim 2 were false then for v' to have weight k' there must be more than k indices j for which some variable $x[i,j]$ of V_1 has the value 1, a contradiction in consideration of E_1 and E_2.

The clauses of E_3 and the arguments above show that v' necessarily has the following restricted form:

(1) Exactly k variables of V_0 are set to 1.

(2) For each of the k in (1) the corresponding $(1+2^h)2^{2^h}+1$ variables of V_1 are set to 1.

(3) For each $i = 1, ..., m$ there is exactly one j_i for which $u[i, j_i] \in V_2$ is set to 1.

(4) For each $i = 1, ..., m$ and $j = 1, ..., m'_i$ there is exactly one k_i for which $w[i, j, k_i] \in V_3$ is set to 1.

(5) For each $i = 1, ..., m$ and $j = m'_i + 1, ..., m_i$ there is exactly one k_i for which $v[i, j, k_i] \in V_4$ is set to 1.

To argue that υ satisfies E it suffices to argue that υ satisfies every E_{i,j_i} for $i = 1, ..., m$.

If E_{i,j_i} is a sum-of-products then the clauses of E_5 ensure that if $\upsilon'(u[i,j]) = 1$ then $k_i \neq 0$. This being the case, the clauses of E_{8b} force the literals in the k_ith subexpression of E_{i,j_i} to evaluate in a way that shows E_{i,j_i} to evaluate to 1.

If E_{i,j_i} is a product-of-sums then the clauses of E_6 ensure that if $\upsilon'(u[i,j]) = 1$ then E_{i,j_i} evaluates to 1. □

8.3.2 Proof of Theorem 8.4

Theorem. $\#W[1, s] = \#$ANTIMONOTONE $W[1, s]$ for all $s \geq 2$.

Proof:

Clearly, $\#W[1, s] \supseteq \#$ANTIMONOTONE $W[1, s]$.

To show that $\#W[1, s] \subseteq \#$ANTIMONOTONE $W[1, s]$ we will identify a parameterized counting problem, #s-RED/BLUE NONBLOCKER, that belongs to #ANTIMONOTONE $W[1, s]$, and then show that this problem is hard for $\#W[1, s]$.

#RED/BLUE NONBLOCKER

Input:	A graph $G = (V, E)$, where V is partitioned into two color classes $V = V_{\text{red}} \cup V_{\text{blue}}$.
Parameter:	A positive integer k.
Output:	The number of sets of red vertices $V' \subseteq V_{\text{red}}$, of cardinality k, such that every blue vertex has at least one neighbour that does not belong to V'.

#s-RED/BLUE NONBLOCKER is the restriction of RED/BLUE NONBLOCKER to graphs G of maximum degree s.

To see that #s-RED/BLUE NONBLOCKER belongs to #ANTIMONOTONE $W[1, s]$:

Consider a graph $G = (V_{\text{red}} \cup V_{\text{blue}}, E)$, of maximum degree s. The *closed neighbourhood* of a vertex $u \in V_{\text{blue}}$ is the set of vertices

$$N[u] = \{v : v \in V_{\text{red}} \cup V_{\text{blue}} \text{ and } v = u \text{ or } vu \in E\} \ .$$

We define a set of variables

$$V = \{u_i \ : \ 1 \leq i \leq |V_{\text{blue}}|\} \cup \{x_i \ : \ 1 \leq i \leq |V_{\text{red}}|\}$$

corresponding to the vertices of G.

Now consider the product-of-sums boolean expression over V

$$\prod_{u \in V_{\text{blue}}} \ \sum_{x_i \in N[u] \cap V_{\text{red}}} \neg x_i \ .$$

This expression corresponds directly to a circuit meeting the defining conditions for #ANTIMONOTONE $W[1, s]$, that is, a weft 1, depth 2 decision circuit, with the *or* gates on level 1 having fanin bounded by s, and with all the input variables negated (and having no other inverter).

Each weight k truth assignment that satisfies this expression corresponds to a unique size k nonblocking set for G. Thus, we have a parsimonious parametric counting reduction from #s-RED/BLUE NONBLOCKER to #ANTIMONOTONE $W[1, s]$.

We next argue that #s-RED/BLUE NONBLOCKER is complete for #$W[1, s]$.

Suppose that we are given an s-normalized circuit C, then C is isomorphic to X, a boolean expression in conjunctive normal form with clauses of size bounded by s. Suppose X consists of m clauses $c_1, ..., c_m$ over the set of n variables $x_0, ..., x_{n-1}$. We show how to produce, in time $O(k.n^c)$, c a constant, a graph $G = (V_{\text{red}} \cup V_{\text{blue}}, E)$ such that, for each nonblocking set of size $2k$ for G, there is a unique weight k truth assignment satisfying X.

The construction we use is taken directly from [48], we just need to argue that the reduction is parsimonious.

We first give an overview of the construction. There are $2k$ 'red' components arranged in a line. These are alternatively grouped as blocks from $V_{\text{red}} = V_1 \cup V_2$, $(V_1 \cap V_2 = \emptyset)$. Each block of vertices from V_1, and then V_2, will be precisely described below. The idea is that V_1 blocks should represent a positive choice (corresponding to a literal being true) and that the V_2 blocks should correspond to the 'gap' until the next positive choice. We think of the V_1 blocks as $A(0), ..., A(k-1)$, with a V_2 block between successive $A(i)$, $A(i+1)$ blocks, the last group following $A(k)$.

We will ensure that for each pair in a block there will be a blue vertex connected to the pair and nowhere else (these are the sets V_3 and V_5). This device ensures that at most one red vertex from each block can be chosen and since we must choose $2k$ this ensures that we choose *exactly* one red vertex from each block.

We think of the V_2 blocks as arranged in k columns. Now if i is chosen from a V_1 block we will ensure that column i gets to select the next gap. To ensure this we connect a blue degree 2 vertex to i and each vertex not in the i-th column of the next V_2 block. This means that, if i is chosen, since these blue vertices must have an unchosen red neighbour, we must choose from the i-th column.

The final part of the component design is to enforce consistency in the next V_1 block. That is, if we choose i and have a gap choice in the next V_2 block, column i, of j, then the next chosen variable should be $i+j+1$. Again we can enforce this by using many degree 2 blue vertices to block any other choice. (These are the V_6 vertices.) Also, we keep the k choices for V_1 blocks in ascending order $1 \leq c_1 \leq ... \leq c_k \leq n$ with c_i in $A(i)$ by the use of extra blue enforcers, the blue vertices V_8. We ensure that for each j in $A(i)$ and each $j' \leq j$ in $A(q)$ with $(q > i)$ there is a blue vertex v in V_8 adjacent to both j and j'. This ensures that if j is chosen in block $A(i)$ then we cannot choose any $j' \leq j$ in any subsequent V_1 block.

The last part of the construction is to force consistency with the clauses. We do this as follows. For each way a nonblocking set can correspond to making a clause false, we make a blue vertex and join it up to the s relevant vertices. This ensures that they can't *all* be chosen. (This is the point of the V_7 vertices.) We now turn to the formal details.

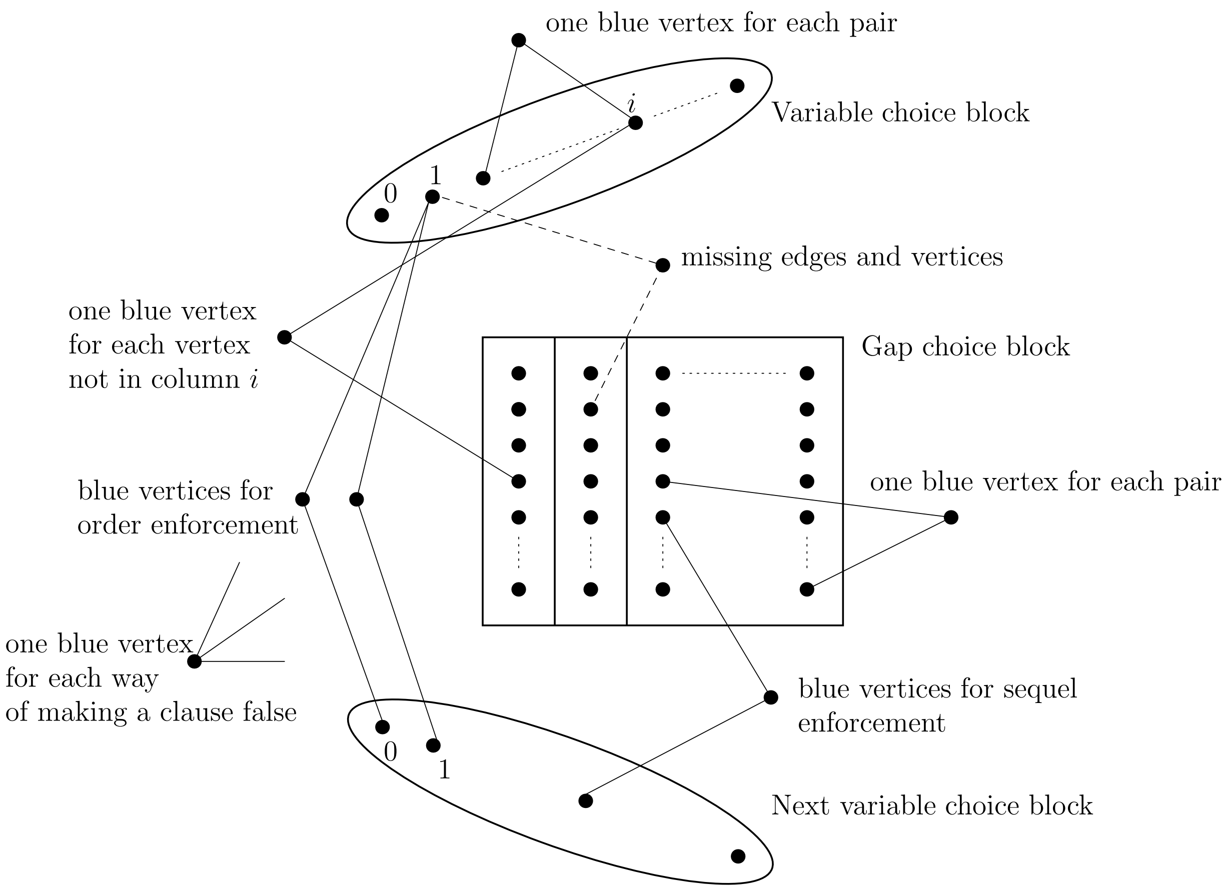

one blue vertex for each pair
Variable choice block
i
0
1
missing edges and vertices
one blue vertex for each vertex not in column i
Gap choice block
blue vertices for order enforcement
one blue vertex for each pair
one blue vertex for each way of making a clause false
blue vertices for sequel enforcement
0
1
Next variable choice block

Figure 8.1: Gadget for RED/BLUE NONBLOCKER.

The red vertex set V_{red} of G is the union of the following sets of vertices:

$V_1 = \{a[r_1, r_2] : 0 \le r_1 \le k-1, 0 \le r_2 \le n-1\}$,

$V_2 = \{b[r_1, r_2, r_3] : 0 \le r_1 \le k-1, 0 \le r_2 \le n-1, 1 \le r_3 \le n-k+1\}$.

The blue vertex set V_{blue} of G is the union of the following sets of vertices:

$V_3 = \{c[r_1, r_2, r_2'] : 0 \le r_1 \le k-1, 0 \le r_2 < r_2' \le n-1\}$,

$V_4 = \{d[r_1, r_2, r_2', r_3, r_3'] : 0 \le r_1 \le k-1, 0 \le r_2, r_2' \le n-1, 0 \le r_3,$
$r_3' \le n-1 \text{ and either } r_2 \ne r_2' \text{ or } r_3 \ne r_3'\}$,

$V_5 = \{e[r_1, r_2, r_2', r_3] : 0 \le r_1 \le k-1, 0 \le r_2, r_2' \le n-1, r_2 \ne r_2', 1 \le r_3 \le n-k+1\}$,

$V_6 = \{f[r_1, r_1', r_2, r_3] : 0 \le r_1,$
$r_1' \le k-1, 0 \le r_2 \le n-1, 1 \le r_3 \le n-k+1, r_1' \ne r_2 + r_3\}$,

$V_7 = \{g[j, j'] : 1 \le j \le m, 1 \le j' \le m_j\}$.

$V_8 = \{h[r_1, r_1', j, j'] : 0 \le r_1 < r_1' \le k-1 \text{ and } j \ge j'\}$.

In the description of V_7, the integers m_j are bounded by a polynomial in n and k whose degree is a function of s which will be described below. Note that since s is a fixed constant independent of k, this is allowed by our definition of for parameterized transformations.

For convenience we distinguish the following sets of vertices.

$$\begin{aligned} A(r_1) = \quad & \{a[r_1, r_2] : 0 \leq r_2 \leq n-1\}, \\ B(r_1) = \quad & \{b[r_1, r_2, r_3] : 0 \leq r_2 \leq n-1, 1 \leq r_3 \leq n-k+1\}, \\ B(r_1, r_2) = \quad & \{b[r_1, r_2, r_3] : 1 \leq r_3 \leq n-k+1\}. \end{aligned}$$

The edge set E of G is the union of the following sets of edges. In these descriptions we implicitly quantify over all possible indices for the vertex sets $V_1, ..., V_8$.

$$\begin{aligned} E_1 = \quad & \{a[r_1, q]c[r_1, r_2, r_2'] : q = r_2 \text{ or } q = r_2'\}, \\ E_2 = \quad & \{b[r_1, q_2, q_3]d[r_1, r_2, r_2', r_3, r_3'] : \text{ either } (q_2 = r_2 \text{ and } q_3 = r_3) \\ & \quad \text{or } (q_2 = r_2' \text{ and } q_3 = r_3')\}, \\ E_3 = \quad & \{a[r_1, r_2]e[r_1, r_2, q, q']\}, \\ E_4 = \quad & \{b[r_1, q, q']e[r_1, r_2, q, q']\}, \\ E_5 = \quad & \{b[r_1, r_2, r_3]f[r_1, r_1', r_2, r_3]\}, \\ E_6 = \quad & \{a[r_1 + 1 \bmod n, r_1']f[r_1, r_1', r_2, r_3]\}, \\ E_7 = \quad & \{a[r_1, j]h[r_1, r_1', j, j']\}, \\ E_8 = \quad & \{a[r_1', j']h[r_1, r_1', j, j']\}. \end{aligned}$$

We say that a red vertex $a[r_1, r_2]$ *represents the possibility* that the boolean variable x_{r_2} may evaluate to *true* (corresponding to the possibility that $a[r_1, r_2]$ may belong to a $2k$-element nonblocking set V' in G). Because of the vertices V_8 and the edge sets E_7 and E_8, for any k nonblocking elements we must have one from each V_1 block, $\{a[r_j, r_{i_j}] : j = 0, ..., k-1\}$, $i_0 < i_1 ... < i_k \leq k-1$.

We say that a red vertex $b[r_1, r_2, r_3]$ *represents the possibility* that the boolean variables $x_{r_2+1}, ..., x_{r_2+r_3-1}$ (with indices reduced mod n) may evaluate to *false*.

Suppose c is a clause of X having s literals. There are $O(n^{2s})$ distinct ways of choosing, for each literal $l \in c$, a single vertex representative of the possibility that $l = x_i$ may evaluate to *false*, in the case that l is a positive literal, or in the case that l is a negative literal $l = \neg x_i$, a representative of the possibility that x_i may evaluate to *true*. For each clause c_j of X, $j = 1, ..., m$, let $R(j,1), R(j,2), ..., R(j, m_j)$ be an enumeration of the distinct possibilities for such a set of representatives. We have the additional sets of edges for the clause components of G:

$$\begin{aligned} E_9 = &\quad \{a[r_1, r_2]g[j, j'] : a[r_1, r_2] \in R(j, j')\}, \\ E_{10} = &\quad \{b[r_1, r_2, r_3]g[j, j'] : b[r_1, r_2, r_3] \in R(j, j')\}. \end{aligned}$$

Suppose X has a satisfying truth assignment τ of weight k, with variables $x_{i_0}, x_{i_1}, ..., x_{i_{k-1}}$ assigned the value *true*. Suppose $i_0 < i_1 < ... < i_{k-1}$. Let $d_r = i_{r+1(\text{mod}k)} - i_r$ (mod n) for $r = 0, ..., k-1$. It is straightforward to verify that the set of $2k$ vertices

$$N = \{a[r, i_r] : 0 \leq r \leq k-1\} \cup \{b[r, i_r, d_r] : 0 \leq r \leq k-1\}$$

is a nonblocking set in G.

Conversely, suppose N is a $2k$-element nonblocking set in G. It is straightforward to check that a truth assignment for X of weight k is described by setting those variables *true* for which a vertex representative of this possibility belongs to N, and by setting all other variables to *false*.

We need to argue that the transformation may be viewed as a parsimonious parametric counting reduction:

Note that the edges of the sets E_1 (E_2) which connect pairs of distinct vertices of $A(r_1)$ ($B(r_1)$) to blue vertices of degree two, enforce that any $2k$-element nonblocking set must contain exactly one vertex from each of the sets $A(0), B(0), A(1), B(1), ...,$ $A(k-1), B(k-1)$.

The edges of E_3 and E_4 enforce (again by connections to blue vertices of degree two) that if a representative of the possibility that x_i evaluates to *true* is selected for a nonblocking set from $A(r_1)$, then a vertex in the i^{th} row of $B(r_1)$ must be selected as well, representing (consistently) the interval of variables set false (by increasing index because of the E_7 and E_8 edges) until the "next" variable selected to be *true*.

The edges of E_5 and E_6 ensure consistency between the selection in $A(r_1)$ and the selection in $A(r_1 + 1 \text{ mod } n)$. The edges of E_9 and E_{10} ensure that a consistent selection can be nonblocking if and only if it does not happen that there is a set of representatives for a clause witnessing that every literal in the clause evaluates to *false*. (There is a blue vertex for every such possible set of representatives.)

Thus, each τ, a nonblocking set of red vertices in G of size $2k$, corresponds to τ', a unique weight k truth assignment satisfying X, where the k variables set to true in τ' correspond to the k vertices in in τ chosen from the $A(0), \ldots, A(k-1)$ sets. Each τ', a weight k truth assignment satisfying X, corresponds to τ, a unique nonblocking set of red vertices in G of size $2k$, where the k vertices in τ chosen from the $A(0), \ldots, A(k-1)$ sets correspond to the variables set to true in τ', and the k vertices in τ chosen from the $B(0), \ldots, B(k-1)$ sets correspond to the intervals between these. □

8.3.3 Proof of Theorem 8.5

Theorem. $\#W[1] = \#W[1,2]$.

Proof:

Clearly, $\#W[1] \supseteq \#W[1,2]$. By theorems 8.3 and 8.4, it suffices to argue that for all $s \geq 2$, #ANTIMONOTONE $W[1,s] \subseteq \#W[1,2]$.

Let C be an antimonotone s-normalized circuit for which we wish to determine the number of weight k input vectors accepted. We show how to produce a circuit C' corresponding to an expression in conjunctive normal form with clause size two, that accepts a unique input vector of weight

$$k' = k2^k + \sum_{i=2}^{s} \binom{k}{i}$$

for each input vector of weight k accepted by C. Note that C' is not, necessarily, antimonotone, but, by Theorem 8.4, this is immaterial.

The construction we use is again taken directly from [48], we just need to show that the transformation described in [48] is parsimonious.

Let $x[j]$ for $j = 1, ...n$ be the input variables to C. The idea is to create new variables representing all possible sets of at most s and at least 2 of the variables $x[j]$. Let $A_1, ..., A_p$ be an enumeration of all such subsets of the input variables $x[j]$ to C.

The inputs to each *or* gate g in C (all negated, since C is antimonotone) are precisely the elements of some A_i. The new input corresponding to A_i represents that all of the variables whose negations are inputs to the gate g have the value *true*. Thus, in the construction of C', the *or* gate g is replaced by the negation of the corresponding new "collective" input variable. The idea being that if it is NOT the case that all the variables whose negations are inputs to the gate g have the value *true*, then at least one of these variables would force the output of g to be true.

We introduce new input variables of the following kinds:

(1) One new input variable $v[i]$ for each set A_i for $i = 1, ..., p$, to be used as above.

(2) For each $x[j]$ we introduce 2^k copies $x[j, 0], x[j, 1], x[j, 2], ..., x[j, 2^k - 1]$.

In addition to replacing each *or* gate of C by the negation of some A_i, we add to the circuit additional *or* gates of fanin 2 that provide an enforcement mechanism for the change of variables. The necessary requirements can be easily expressed in conjunctive normal form with clause size two, and thus can be incorporated into a 2-normalized circuit.

We require the following implications concerning the new variables:

(1) The $n \cdot 2^k$ implications, for $j = 1, ..., n$ and $r = 0, ..., 2^k - 1$,

$$x[j, r] \Rightarrow x[j, r + 1 \pmod{2^k}].$$

(2) For each containment $A_i \subseteq A_{i'}$, the implication

$$v[i'] \Rightarrow v[i].$$

(3) For each membership $x[j] \in A_i$, the implication

$$v[i] \Rightarrow x[j, 0].$$

It may be seen that this transformation may increase the size of the circuit by a linear factor exponential in k. We now show that the transformation may be viewed as a parsimonious parameterized counting reduction.

If C accepts a weight k input vector τ, then we build τ' for C' by setting the corresponding copies $x[i, j]$ among the new input variables accordingly, together with appropriate settings for the new "collective" variables $v[i]$. This yields a vector of weight k' that is accepted by C'.

Note that, if τ' is a weight k' input vector that is accepted by C' then τ' must have at most k of the $x[i, 0]$'s set to 1, otherwise the implications in (1) above would force τ' to have weight $> k'$.

Suppose that τ' sets $< k$ of the $x[i, 0]$'s to 1, then we need to have more than 2^k of the $v[i]$'s set to 1, but the implications in (3) above would now force $> k$ of the $x[i, 0]$'s to be set to 1, so we have a contradiction. We can't force more than 2^k distinct subsets to be produced from fewer than k elements.

Thus, exactly k sets of copies of inputs to C must have value 1 in τ'.

Because of the implications described in (2) and (3) above, the $\sum_{i=2}^{s} \binom{k}{i}$ $v[i]$'s that we need to set to 1 in τ' can only be those compatible with the k sets of copies.

Thus, for each weight k input vector accepted by C, there is exactly one weight k' input vector accepted by C'.

For the other direction, suppose C' accepts a vector of weight k'. As explained above, exactly k sets of copies of inputs to C must have value 1 in the accepted input vector, and the $\sum_{i=2}^{s} \binom{k}{i}$ $v[i]$'s set to 1 must have values in the accepted input vector compatible with the values of the sets of copies. By the construction of C', this implies there is a unique weight k input vector accepted by C corresponding to the weight k' input vector accepted by C'. □

8.3.4 Proof of Theorem 8.6

Theorem. The following are complete for $\#W[1]$:

(i) #INDEPENDENT SET

(ii) #CLIQUE

Proof:

We first demonstrate that #INDEPENDENT SET belongs to $\#W[1]$.

Suppose we are given a graph $G = (V, E)$. Consider the following product-of-sums boolean expression:

$$\prod_{uv \in E} \sum (\neg u + \neg v).$$

This expression is isomorphic to a 2-normalized circuit C, where each weight k truth assignment that satisfies C corresponds to a unique weight k independent set for G.

To show that #INDEPENDENT SET is hard for $\#W[1]$, by theorems 8.4 and 8.5 it is enough to argue that #INDEPENDENT SET is hard for #ANTIMONOTONE $W[1, 2]$.

Suppose we are given a boolean expression X in conjunctive normal form (product-of-sums) with clause size two and all literals negated, we may form a graph G_X with one vertex for each variable of X, and having an edge between each pair of vertices corresponding to variables in a clause. Each independent set of size k in G corresponds to a unique weight k truth assignment satisfying X.

Suppose τ is weight k truth assignment satisfying X, let τ' be the set of k vertices in G_X corresponding to variables set to true in τ.

No clause in X may have both of its variables set to true by τ. Thus, for each edge in G_X, one of its endpoints cannot be included in the k vertices of τ', and so τ' forms an independent set in G_X.

Suppose τ' is an independent set of size k in G_X, then for each edge in G_X, at least one of its endpoints must not be included in τ'. Thus, τ, the corresponding weight k truth assignment to variables of X, must satisfy all clauses in X.

(ii) This follows immediately by considering the complement of a given graph. The complement has an independent set of size k corresponding to each clique of size k in the given graph. □

8.3.5 Proof of Theorem 8.2

Theorem. #SHORT TURING MACHINE ACCEPTANCE is complete for $\#W[1]$.

Proof:

To show hardness for $\#W[1]$ we reduce from #CLIQUE, $\#W[1]$-hardness will then follow by Theorem 8.6 (ii).

Suppose we wish to determine the number of k-cliques in a given graph $G = (V, E)$.

We first order the vertices of G, $\{x_1, \ldots, x_n\}$. We then construct a nondeterministic Turing machine, M_G, that has a unique, accepting, $k' = f(k)$ move, computation for each different k-clique contained in G. M_G is designed so that any accepting computation consists of two phases.

In the first phase, M_G nondeterministically writes k symbols representing vertices of G in the first k tape squares. (There are enough symbols so that each vertex of G is represented by a symbol.)

The second phase consists of two parts. First, M_G makes a scan of the k tape squares to ensure that the k vertices chosen are in ascending order. This can be accomplished by employing $O(|V|^2)$ states in M_G. Second, M_G makes $\binom{k}{2}$ scans of the k tape squares, each scan devoted to checking, for a pair of positions (i, j), that the vertices represented by the symbols in these positions are adjacent in G. Each such pass can be accomplished by employing $O(|V|)$ states in M_G dedicated to the ij^{th} scan.

Each set of k vertices that form a clique in G corresponds to a unique accepting computation of M_G . Thus, we have a parsimonious parameterized counting reduction from #CLIQUE to #SHORT TURING MACHINE ACCEPTANCE.

To show membership in $\#W[1]$ we describe a parsimonious parameterized counting reduction from #SHORT TURING MACHINE ACCEPTANCE to #WEIGHTED WEFT 1 DEPTH h CIRCUIT SATISFIABILITY (#WCS$(1, h)$). Recall that we originally defined $\#W[1]$ to be the class of counting problems whose witness functions could be reduced to $w_{\mathcal{F}(1,h)}$, the standard parameterized witness function for the family $F(1, h)$ of weft 1 circuits of depth bounded by h, for some h.

Given a Turing machine $M = (\Sigma, Q, q_0, \delta, F)$ and positive integer k, we build a circuit C that accepts a unique weight k' input vector corresponding to each accepting k-step computation for M.

C has depth bounded by some h (independent of k and the Turing machine M), and has only a single large (output) *and* gate, with all other gates small.

We arrange the circuit C so that the k' inputs to be chosen to be set to 1 in a weight k' input vector, represent the various data: (1) the i^{th} transition of M, for $i = 1, ..., k$, (2) the head position at time i, (3) the state of M at time i, and (4) the symbol in square j at time i for $1 \leq i, j \leq k$. Thus we may take $k' = k^2 + 3k$.

In order to force exactly one input to be set equal to 1 among a pool of input variables (for representing one of the above choices), we add to the circuit, for each such pool of input variables, and for each pair of variables x and y in the pool, a small "not both" circuit representing $(\neg x \vee \neg y)$. It might seem that we must also enforce (e.g. with a large *or* gate) the condition, "at least one variable in each such pool is set true" — but this is actually unnecessary, since in the presence of the "not both" conditions on each pair of input variables in each pool, an accepted weight k' input vector *must have* exactly one variable set true in each of the k' pools.

Let n denote the total number of input variables in this construction. It will be the case that $n = O(k|\delta| + k^2 + k|Q| + k^2|\Sigma|)$.

The remainder of the circuit encodes various checks on the consistency of the above choices. These consistency checks conjunctively determine whether the choices represent an accepting k-step computation by M, much as in the proof of Cook's theorem. These consistency checks can be implemented so that each involves only a bounded number b of the input variables. For example, we will want to enforce that if five variables are set true indicating particular values of: the tape head position at time $i + 1$, and the head position, state, scanned symbol and machine transition at time i, then the values are consistent with δ. Thus $O(n^5)$ small "checking" circuits of bounded depth are sufficient to make these consistency checks; in general we will have $O(n^b)$ "checking" circuits for consistency checks involving b values. All of the small "not both" and "checking" circuits feed into the single large output *and* gate of C.

The range of input variables for C allows us to describe exactly any k-step computation for M via a unique weight k' truth assignment for these variables. The small "not both" and "checking circuits" ensure that all and only weight k' truth assignments that correspond to k-step accepting computations for M will satisfy C. Thus, each k-step accepting computation for M corresponds to a unique weight k' truth assignment satisfying C. □

8.4 Populating $\#W[1]$

In this section, we note the complexity of some other parameterized counting problems relative to $\#W[1]$. The first of these is #PERFECT CODE, which we show to be $\#W[1]$-complete.

8.4.1 PERFECT CODE

The decision version of this problem was shown to be $W[1]$-hard in [48], and it was conjectured that the problem could be of intermediate difficulty between $W[1]$ and $W[2]$. Until recently, there was no evidence that the problem belonged to $W[1]$, although it can easily be shown to belong to $W[2]$. In [33] Cesati has shown that PERFECT CODE is, in fact, $W[1]$-complete.

A consequence of our argument proving $\#W[1]$-completeness for #PERFECT CODE is an alternative proof of membership in $W[1]$ for PERFECT CODE.

We define #PERFECT CODE as follows:

Input:	A graph $G = (V, E)$.
Parameter:	A positive integer k.
Output:	The number of distinct k-element perfect codes in G.
	A *perfect code* is a set $V' \subseteq V$ such that
	for each $v \in V$, there is exactly *one* vertex in $N[v] \cap V'$.

Theorem 8.7. *#PERFECT CODE is* $\#W[1]$*-hard.*

Proof: To show that #PERFECT CODE is $\#W[1]$-hard, we transform from #INDEPENDENT SET.

Let $G = (V, E)$ be a graph. We show how to produce a graph $H = (V', E')$ that has a unique perfect code of size $k' = \binom{k}{2} + 2k + 1$ corresponding to each independent set of size k in G. The construction that we use is based on a construction given in [48] with some modifications to ensure parsimony.

We will produce H in two phases. In the first phase we produce a graph $H'' = (V'', E'')$ such that H'' has a perfect code of size $k'' = \binom{k}{2} + k + 1$ if and only if G has an independent set of size k. We then modify H'' to produce a graph $H = (V', E')$ that has a *unique* perfect code of size $k' = \binom{k}{2} + 2k + 1$ corresponding to each independent set of size k of G.

The vertex set V'' of H'' is the union of the following sets of vertices:

$$\begin{aligned}
V_1 = &\ \{a[s] \,:\, 0 \leq s \leq 2\},\\
V_2 = &\ \{b[i] \,:\, 1 \leq i \leq k\},\\
V_3 = &\ \{c[i] \,:\, 1 \leq i \leq k\},\\
V_4 = &\ \{d[i,u] \,:\, 1 \leq i \leq k, u \in V\},\\
V_5 = &\ \{e[i,j,u] \,:\, 1 \leq i < j \leq k, u \in V\},\\
V_6 = &\ \{f[i,j,u,v] \,:\, 1 \leq i < j \leq k, u, v, \in V\}.
\end{aligned}$$

The edge set E'' of H'' is the union of the following sets of edges:

$$\begin{aligned}
E_1 = &\ \{\, (a[0], a[i]) \,:\, i = 1, 2\},\\
E_2 = &\ \{\, (a[0], b[i]) \,:\, 1 \leq i \leq k\},\\
E_3 = &\ \{\, (b[i], c[i]) \,:\, 1 \leq i \leq k\},\\
E_4 = &\ \{\, (c[i], d[i,u]) \,:\, 1 \leq i \leq k, u \in V\},\\
E_5 = &\ \{\, (d[i,u], d[i,v]) \,:\, 1 \leq i \leq k, u, v \in V\},\\
E_6 = &\ \{\, (d[i,u], e[i,j,u]) \,:\, 1 \leq i < j \leq k, u \in V\},\\
E_7 = &\ \{\, (d[j,v], e[i,j,x]) \,:\, 1 \leq i < j \leq k, x \in N[v]\},\\
E_8 = &\ \{\, (e[i,j,x], f[i,j,u,v]) \,:\, 1 \leq i < j \leq k, x \neq u, x \notin N[v]\},\\
E_9 = &\ \{\, (f[i,j,x,y], f[i,j,u,v]) \,:\, 1 \leq i < j \leq k, x \neq u \vee y \neq v\}.
\end{aligned}$$

Suppose C is a perfect code of size k'' in H''. C must contain $a[0]$, since otherwise C must contain both $a[1]$ and $a[2]$ and in this case, C would contain two neighbours of $a[0]$.

Since $a[0]$ is in C, none of the vertices of V_2 or V_3 can be in C. To cover V_3, it must be the case that exactly one vertex in each of the cliques formed by V_4 and the edges of E_5 is in C. Note that each of these cliques contains $|V|$ vertices, indexed by V. We can think of these cliques as a set of k *vertex selection* components. Let I be the set of vertices of G corresponding to the elements of C in these cliques. We argue that I is an independent set of size k in G.

Choose any i, j such that $1 \leq i < j \leq k$. Suppose that $d[i,u] \in C$ and $d[j,v] \in C$, with $u = v$ or $(u,v) \in E$. In H'' it must be the case that each of $d[i,u]$ and $d[j,v]$ is adjacent to $e[i,j,u]$, which contradicts that C is perfect code. Thus, $u \neq v$ and $(u,v) \notin E$, so I is an independent set in G.

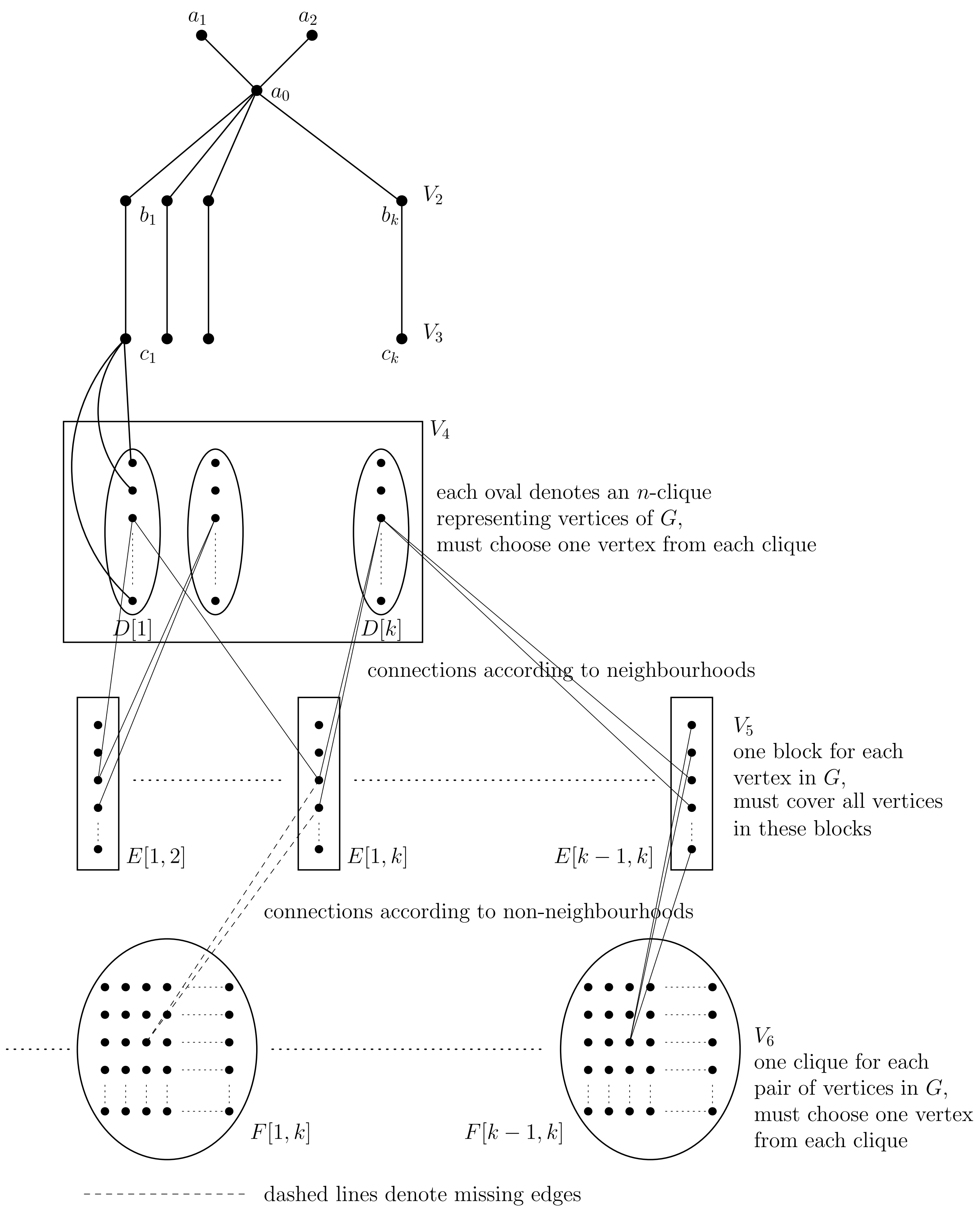

a_1
a_2
a_0
b_1
b_k
V_2
c_1
c_k
V_3
V_4
each oval denotes an n-clique representing vertices of G, must choose one vertex from each clique
$D[1]$
$D[k]$
connections according to neighbourhoods
V_5
one block for each vertex in G, must cover all vertices in these blocks
$E[1,2]$
$E[1,k]$
$E[k-1,k]$
connections according to non-neighbourhoods
V_6
one clique for each pair of vertices in G, must choose one vertex from each clique
$F[1,k]$
$F[k-1,k]$
dashed lines denote missing edges

Figure 8.2: Overview of the graph H'' constructed from $G = (V, E)$.

Now suppose that $J = \{u_1, u_2, \ldots, u_k\}$ is a size k independent set in G. Let C_J be the following set of vertices:

$$C_J = \{\ a[0]\ \} \cup \{\ d[i, u_i]\ :\ 1 \leq i \leq k\ \} \cup \{\ f[i, j, u_i, u_j]\ :\ 1 \leq i < j \leq k\ \}$$

C_J is a perfect code of size k'' in H'', we can verify this directly from the definition of H''. However, C_J is not the only perfect code of size k'' in H'' that selects J. For example, for some i and j, $1 \leq i, j \leq k$, we could have chosen $d[j, u_i]$ and $d[i, u_j]$ and then $f[i, j, u_j, u_i]$. In order to ensure that the transformation is parsimonious we need to modify H''.

Let $\tau\ :\ V \to \{0, 2, \ldots, n-1\}$ be an ordering of the vertices in V. We add the following vertices and edges to H'' to produce H:

$$\begin{aligned}
V_7 = &\ \{g[i, u, t]\ :\ 1 \leq i \leq k, u \in V, 0 \leq t \leq n-k\},\\
V_8 = &\ \{h[i, u]\ :\ 1 \leq i \leq k, u \in V\},\\
V_9 = &\ \{z[i]\ :\ 1 \leq i \leq k\}.
\end{aligned}$$

$$\begin{aligned}
V_{10} = &\ \{n[i, j, u]\ :\ 1 \leq i < j \leq k, u \in V\},\\
V_{11} = &\ \{y[i, j]\ :\ 1 \leq i < j \leq k\}.
\end{aligned}$$

$$\begin{aligned}
E_{10} = &\ \{\ (d[i, u], g[i, v, t])\ :\ 1 \leq i \leq k, u \neq v, 0 \leq t \leq n-k\},\\
E_{11} = &\ \{\ (g[i, u, t], g[i, u, t'])\ :\ 1 \leq i \leq k, u \in V, t \neq t'\},\\
E_{12} = &\ \{\ (g[i, u, t], h[i, v])\ :\ 1 \leq i \leq k-1, \tau\{u\} + t + 1 \leq n-1,\\
&\ \tau\{u\} + t + 1 \neq \tau\{v\}\ \} \cup \{\ (g[k, u, t], h[k, v])\ :\ \tau\{u\} + t + 1 \bmod n \neq \tau\{v\}\ \},\\
E_{13} = &\ \{\ (h[i, u], d[i+1, u])\ :\ 1 \leq i \leq k-1, u \in V\} \cup \{\ (h[k, u], d[1, u])\ :\ u \in V\},\\
E_{14} = &\ \{\ (b[i], z[i])\ :\ 1 \leq i \leq k\},\\
E_{15} = &\ \{\ (z[i], g[i, u, t])\ :\ 1 \leq i \leq k, u \in V, 0 \leq t \leq n-k\},\\
E_{16} = &\ \{\ (b[i], h[i, u])\ :\ 1 \leq i \leq k, u \in V\}.
\end{aligned}$$

$$\begin{aligned}
E_{17} = &\ \{\ (d[j, v], n[i, j, v])\ :\ 1 \leq i < j \leq k, v \in V\},\\
E_{19} = &\ \{\ (f[i, j, u, v], n[i, j, x])\ :\ 1 \leq i < j \leq k, v \neq x\},\\
E_{20} = &\ \{\ (f[i, j, u, v], y[i, j])\ :\ 1 \leq i < j \leq k, u, v \in V\},\\
E_{21} = &\ \{\ (b[i], y[i, j])\ :\ 1 \leq i < j \leq k\}.
\end{aligned}$$

Now suppose that $J = \{u_1, u_2, \ldots, u_k\}$ is a size k independent set in G, with $\tau\{u_1\} < \tau\{u_2\} < \cdots < \tau\{u_k\}$. Let t_i be the 'gap' between u_i and u_{i+1}.

That is, $t_i = \tau\{u_{i+1}\} - \tau\{u_i\} - 1$ for $1 \leq i \leq k-1$, $t_k = \tau\{u_1\} - \tau\{u_k\} - 1 \bmod n$.

Let C_J be the following set of vertices:

$$C_J = \{a[0]\} \cup \{d[i, u_i] : 1 \leq i \leq k\} \cup \{g[i, u_i, t_i] : 1 \leq i \leq k\} \cup \{f[i, j, u_i, u_j] : 1 \leq i < j \leq k\}$$

We argue that C_J is the unique perfect code of size $k' = \binom{k}{2} + 2k + 1$ in H corresponding to J.

C_J is a perfect code in H, it is straightforward but tedious to verify this directly from the definition of H. We will show that C_J is the *only* perfect code in H that selects J.

Suppose C is a perfect code in H. As for H'', we must choose $a[0]$ to be in C, and we must choose exactly one vertex from each of the cliques formed by V_4 and E_5.

We will denote by $D[i]$ the ith clique formed by V_4 and E_5, and we will call this the ith *vertex selection* component. Between $D[i]$ and $D[i+1]$ lie the ith *gap selection* component, and the ith *order enforcement* component.

We denote by $G[i]$ the ith gap selection component, formed by the i-indexed vertices in V_7 and edges in E_{11}. $G[i]$ consists of n cliques, each of size $n - k + 1$, which we will call columns. We denote by $H[i]$ the ith order enforcement component, formed by the i-indexed vertices of V_8. Note that we must choose exactly one vertex from each $G[i]$, $1 \leq i \leq k$, in order to cover the vertices of V_9. Also note that we can't choose any vertex of $H[i]$, $1 \leq i \leq k$, since these vertices are connected to the vertices of V_2.

Claim 1.

If C is a perfect code of H containing $d[1, u_1], d[2, u_2], \ldots, d[k, u_k]$ then it must be the case that $\tau\{u_1\} < \tau\{u_2\} < \cdots < \tau\{u_k\}$. That is, there is only one order in which the vertices $\{u_1, u_2, \ldots, u_k\}$ may be selected.

Proof of Claim 1:

Suppose we choose $d[1, u_i]$ in $D[1]$, then we must choose some vertex in the u_ith column of $G[1]$, since all other columns of $G[1]$ are covered by $d[1, u_i]$. If we choose $g[1, u_i, t]$, then it must be the case that in $D[2]$ we choose $d[2, u_j]$ where $\tau\{u_i\} + t + 1 = \tau\{u_j\}$, in order to cover $h[1, u_j]$ in $H[1]$. Thus, $\tau\{u_i\} < \tau\{u_j\}$.

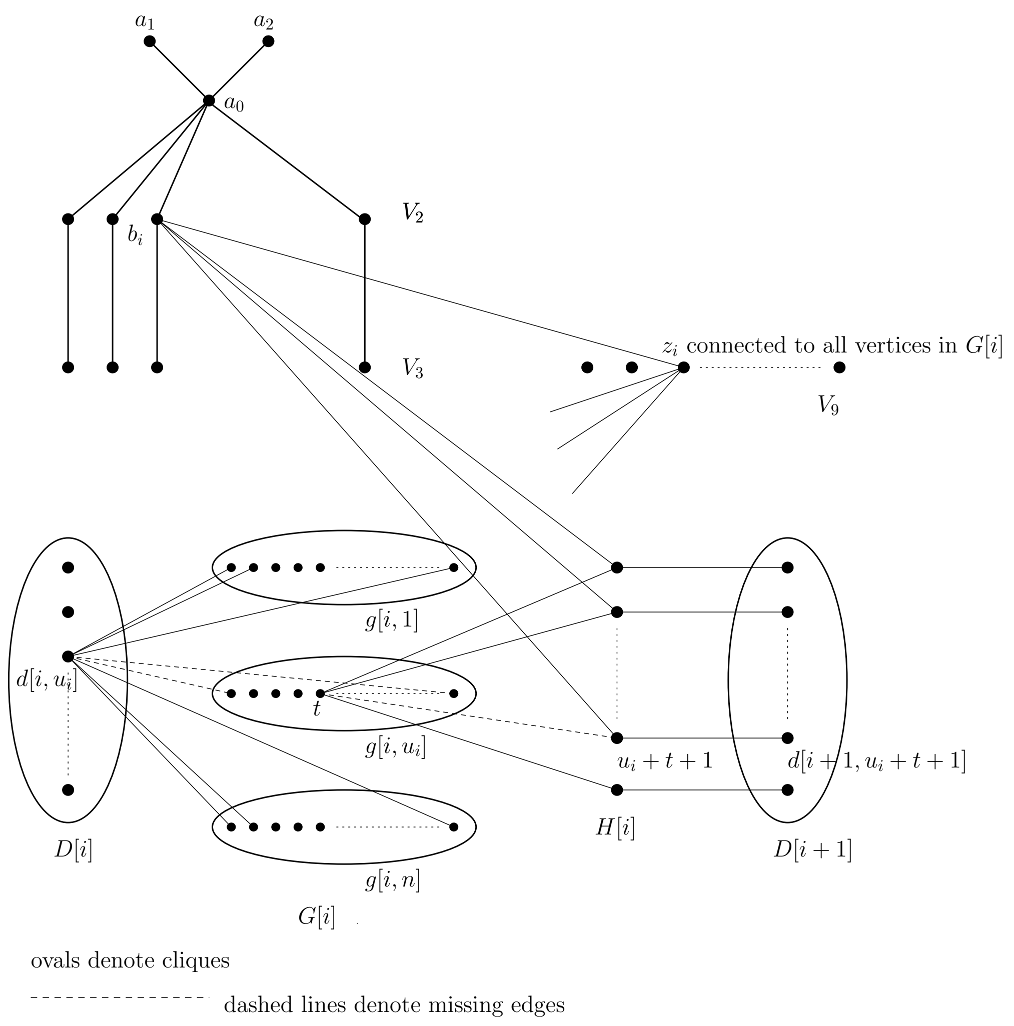


Figure 8.3: Overview of gap selection and order enforcement components of H.

Note also that we must choose $g[1, u_i, t]$ so that $\tau\{u_i\} + t + 1 \leq n - 1$, otherwise none of $H[1]$ will be covered by $g[1, u_i, t]$. Since only one vertex of $H[1]$ can be covered by the vertex chosen in $D[2]$, choosing $g[1, u_i, t]$ with $\tau\{u_i\} + t + 1 > n - 1$ could not produce a perfect code in H.

It is straightforward to see how this argument extends to each consecutive pair $i, i + 1$ for $1 \leq i \leq k - 1$. For $i = k$ we must choose $g[k, u_k, t_k]$ with $\tau\{u_k\} + t_k + 1 \bmod n = \tau\{u_1\}$. Thus, $0 \leq \tau\{u_1\} < \tau\{u_2\} < \cdots < \tau\{u_k\} \leq n - 1$. □

Claim 2.

If C is a perfect code of H containing $d[1, u_1], d[2, u_2], \ldots, d[k, u_k]$ then it must be the case that C contains $\{\, g[i, u_i, t_i] \,:\, 1 \leq i \leq k \,\}$ where $t_i = \tau\{u_{i+1}\} - \tau\{u_i\} - 1$ for $1 \leq i \leq k - 1$, $t_k = \tau\{u_1\} - \tau\{u_k\} - 1 \bmod n$.

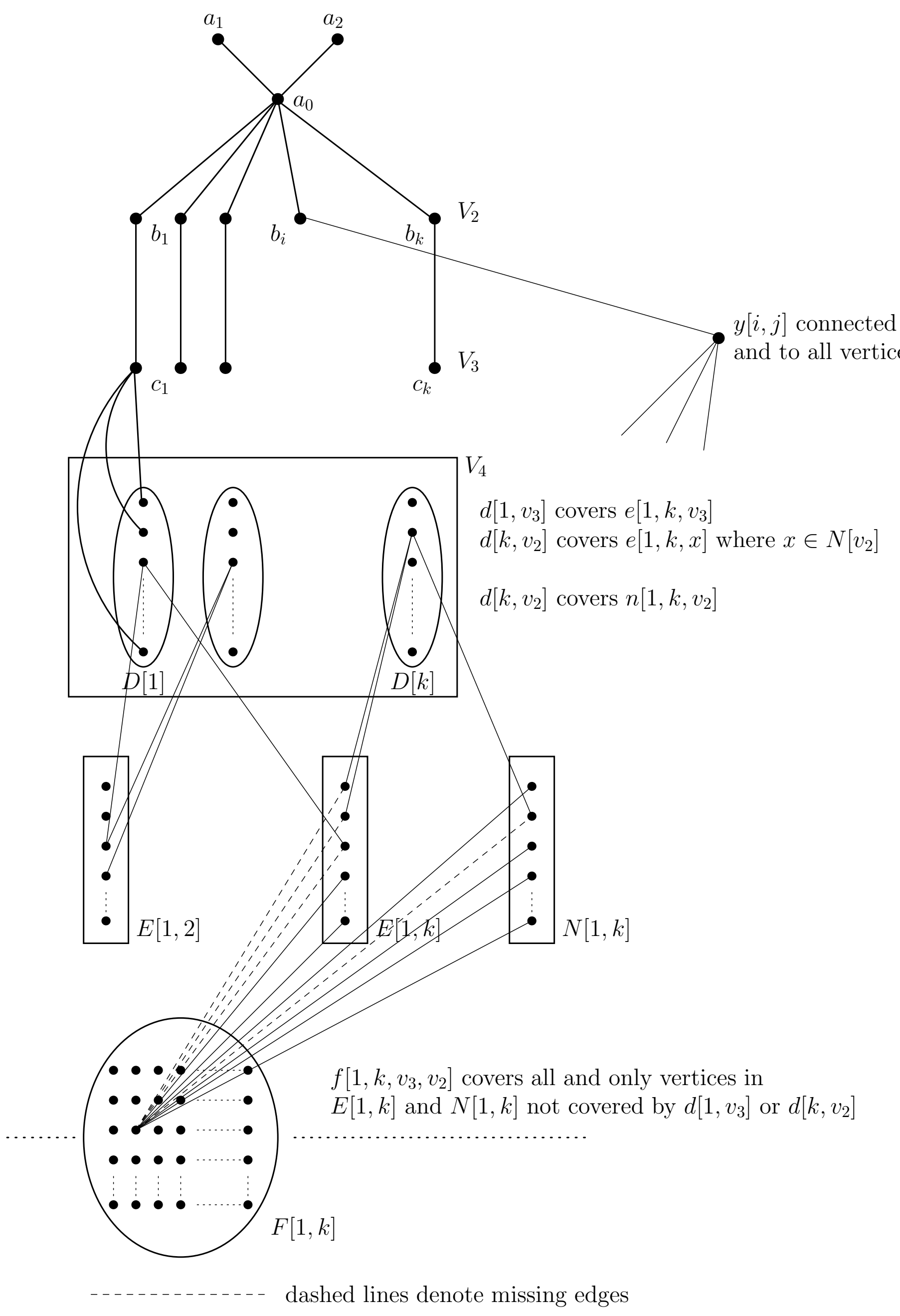

a_1
a_2
a_0
b_1
b_i
b_k
V_2
c_1
c_k
V_3
$y[i,j]$ connected to b_i
and to all vertices in $F[i,j]$
V_4
$d[1,v_3]$ covers $e[1,k,v_3]$
$d[k,v_2]$ covers $e[1,k,x]$ where $x \in N[v_2]$
$d[k,v_2]$ covers $n[1,k,v_2]$
$D[1]$
$D[k]$
$E[1,2]$
$E[1,k]$
$N[1,k]$
$f[1,k,v_3,v_2]$ covers all and only vertices in
$E[1,k]$ and $N[1,k]$ not covered by $d[1,v_3]$ or $d[k,v_2]$
$F[1,k]$
dashed lines denote missing edges

Figure 8.4: Overview of connections to $F[i,j]$ in H.

Proof of Claim 2:

Suppose we have chosen $d[i,u_i]$ in $D[i]$, $d[i{+}1,u_{i+1}]$ in $D[i{+}1]$, for any i, $1 \leq i \leq k{-}1$. Then we must choose $g[i,u_i,t_i]$ with $\tau\{u_i\} + t_i + 1 = \tau\{u_{i+1}\}$, by the definition of H. That is, we must choose $t_i = \tau\{u_{i+1}\} - \tau\{u_i\} - 1$.

Suppose we have chosen $d[k,u_k]$ in $D[k]$, $d[1,u_1]$ in $D[1]$. Then we must choose $g[k,u_k,t_k]$ with $\tau\{u_k\} + t_k + 1 \bmod n = \tau\{u_1\}$, by the definition of H. That is, we must choose $t_k = \tau\{u_1\} - \tau\{u_k\} - 1 \bmod n$. □

Claim 3.

If C is a perfect code of H containing $d[1, u_1], d[2, u_2], \ldots, d[k, u_k]$ then it must be the case that C contains $\{ f[i, j, u_i, u_j] \; : \; 1 \leq i < j \leq k \}$.

Proof of Claim 3:

We denote by $F[i, j]$ the set of vertices $\{ f[i, j, u, v] \; : \; 1 \leq i < j \leq k, u, v \in V \}$. Note that we must choose exactly one vertex from each $F[i, j]$, $1 \leq i < j \leq k$, in order to cover the vertices of V_{11}.

Suppose we have chosen $d[i, u_i]$ in $D[i]$, $d[j, u_j]$ in $D[j]$, for any i, j, $1 \leq i < j \leq k$. Then $e[i, j, u_i]$ is already covered and $n[i, j, u_j]$ is already covered. Thus, any perfect code in H may only contain $f[i, j, u_i, u_j]$ from $F[i, j]$. □

Together, these three claims establish that C_J is the *only* perfect code in H that selects J. Thus, H contains a unique perfect code of size $k' = \binom{k}{2} + 2k + 1$ corresponding to each independent set of size k in G. This completes the proof of Theorem 8.7. □

We note here that the following problems can also be shown to be $\#W[1]$-hard:

#SIZED SUBSET SUM

Input:	A set $X = \{x_1, x_2, \ldots, x_n\}$ of integers and a positive integer S.
Parameter:	A positive integer k.
Output:	The number of distinct k-element subsets of X that sum to S.

We can reduce #PERFECT CODE to #SIZED SUBSET SUM via a parsimonious parameterized counting reduction.

#EXACT CHEAP TOUR

Input:	A weighted graph $G = (V, E, W_E)$ and a positive integer S.
Parameter:	A positive integer k.
Output:	The number of distinct k-edge tours through k vertices of G whose edge weights sum to exactly S.

We can reduce #SIZED SUBSET SUM to #EXACT CHEAP TOUR via a parsimonious parameterized counting reduction.

The following variation on #EXACT CHEAP TOUR can be shown to be $\#W[1]$-hard, if we relax our notion of reducibility from the standard one that we have used so far to a Turing-type reduction.

#SHORT CHEAP TOUR

Input:	A weighted graph $G = (V, E, W_E)$ and a positive integer S.
Parameter:	A positive integer k.
Output:	The number of distinct k-edge tours through k vertices of G whose edge weights sum to *at most* S.

Here, we assume an oracle function Φ that returns the number of witnesses for any instance of #EXACT CHEAP TOUR, and we allow a fixed number of questions to Φ. In order to determine the number of witnesses for #SHORT CHEAP TOUR on input (G, S, k) we need to know the number of witnesses for #EXACT CHEAP TOUR on inputs $(G, 1, k), (G, 2, k), \ldots, (G, S-1, k), (G, S, k)$.

We now turn to the $\#W[1]$-membership question for #PERFECT CODE.

Theorem 8.8. *#PERFECT CODE* $\in \#W[1]$.

Proof: To show that #PERFECT CODE is in $\#W[1]$, we show that there is a parsimonious parameterized counting reduction from #PERFECT CODE to #WEIGHTED WEFT 1, DEPTH h CIRCUIT SATISFIABILITY.

Suppose we are given a graph $G = (V, E)$. We build a circuit, C, that has weft 1 and depth 2, such that any satisfying assignment of weight $2k$ to the input variables of C corresponds uniquely to a perfect code of size k in G.

The construction of C will ensure that any weight $2k$ satisfying assignment to the input variables will correspond to a choice of k vertices in G, chosen in ascending order, and a choice of k *partial sums* of neighbourhood sizes, that correspond correctly to the chosen vertices.

The construction of C will, in effect, mimic the action of the Turing machine described in [33], which, on input $(G = (V, E), k)$, does the following:

- guesses k vertices in V,
- checks that these are all different,
- checks that no two are adjacent,

- checks that no two have a common neighbour,
- checks that the sum of neighbourhood sizes is exactly n.

These actions are sufficient to ensure that, for any accepting computation, the chosen vertices form a perfect code in G. We will need to add some further machinery in the construction of C to ensure that the vertices are chosen in ascending order so that the reduction is parsimonious.

Let us assume for convenience that the vertex set of G is $V = \{1, 2, \ldots, n\}$. C has $(2k \cdot n)$ input variables, arranged in $2k$ blocks, each of size n.

$\{\ x[i, u]\ :\ 1 \leq i \leq k, 1 \leq u \leq n\ \}$
$\{\ s[i, t]\ :\ 1 \leq i \leq k, 1 \leq t \leq n\ \}$

The $x[i, u]$'s represent vertex choices. The $s[i, t]$'s represent partial sums of neighbourhood sizes. We want to ensure that any weight $2k$ satisfying assignment to the input variables of C consists of k different $x[i.u]$'s, chosen in ascending order, representing the vertices of a perfect code in G, and a k-sequence of $s[i, t]$'s, consistent with these vertex choices, so that the last element in the sequence is $s[k.n]$.

We denote the input variables of C as level 0. Directly below these, on level 1, we introduce several sets of small *or* gates, described here:

$X = \{\ \neg x[i, u] + \neg x[i, v]\ :\ 1 \leq i \leq k,\ 1 \leq u < v \leq n\ \}$
$S = \{\ \neg s[i, t] + \neg s[i, t']\ :\ 1 \leq i \leq k,\ 1 \leq t < t' \leq n\ \}$

Any weight $2k$ truth assignment to the input variables of C that forces all of X and S to have positive output must make true exactly one variable from each of the $2k$ input blocks.

$X1 = \{\ \neg x[i, u] + \neg x[j, v]\ :\ 1 \leq i < j \leq k,\ 1 \leq v < u \leq n\ \}$

Any truth assignment to the input variables of C that forces all of $X1$ to have positive output may only choose variables from each of the k vertex selection blocks in ascending order. That is, $x[1, u_1], x[2, u_2], \ldots, x[k, u_k]$ with $u_1 < u_2, \cdots < u_k$.

$M = \{\ \neg x[i, u] + \neg x[j, v]\ :\ 1 \leq i < j \leq k,\ u, v \in V,\ u \in N[v]\ \}$. $N[v]$ is the closed neighbourhood of v, $u \in N[v]$ iff $u = v$ or u is adjacent to v.

Any truth assignment to the input variables of C that forces all of M to have positive output may only choose variables from each of the k vertex selection blocks that are distinct and non-adjacent.

$N = \{\neg x[i,u] + \neg x[j,v] : 1 \leq i < j \leq k, u, v \in V, \exists w \in V$ with $u \in N[w] \wedge v \in N[w]\}$.

Any truth assignment to the input variables of C that forces all of N to have positive output may only choose variables from each of the k vertex selection blocks that do not have any common neighbours in G.

$P1 = \{\neg x[1,u] + s[1,t] : u \in V, |N[u]| = t\}$. t is the size of the closed neighbourhood of u in G.

$P2 = \{\ \neg s[i,t] + \neg x[i+1,u] + s[i+1,t']\ :\ 1 \leq i \leq k-1,\ u \in V,\ t + |N[u]| = t'\ \}$. t' is the sum of the size of the closed neighbourhood of u in G and the index of the variable chosen in the previous neighbourhood-size selection block.

Any truth assignment to the input variables of C that forces all of $P1$ and $P2$ to have positive output may only choose variables from each of the k neighbourhood-size selection blocks consistent with the vertex choices from the corresponding vertex selection blocks.

The output gate of C, on level 2, is a large *and* gate, with inputs from each of the *or* gates on level 1, and the input variable $s[k,n]$.

$$C_{\text{out}} = X \wedge S \wedge X1 \wedge M \wedge N \wedge P1 \wedge P2 \wedge s[k,n]$$

is the large *and* output gate of C.

Suppose $J = \{u_1 < u_2, \ldots, < u_k\}$ is a k-element perfect code in G. It is evident from the construction of C that $C_j = \{x[1,u_1], x[2,u_2], \ldots, x[k,u_k], s[1,|N[u_1]|], s[2,|N[u_1]| + |N[u_2]|], \ldots, s[k,|N[u_1]| + \cdots |N[u_k]|]\ \}$ is a weight $2k$ satisfying assignment for C.

Suppose $C_J = \{x[1,u_1], x[2,u_2], \ldots, x[k,u_k], s[1,t_1], s[2,t_2], \ldots, s[k,t_k]\}$ is a weight $2k$ satisfying assignment for C. Again, it is evident from the construction of C that $\{\ u_1 < u_2, \ldots, < u_k\ \}$ must be a perfect code in G.

It remains to show that, given G, say in the form of an adjacency matrix, we can construct C in time $O(f(k) \cdot n^\alpha)$ for some fixed α, independent of k and n.

- We can construct X and S in time $O(k \cdot n^2)$; for each of the $2k$ input blocks in C we need to construct $O(n^2)$ small *or* gates.

- We can construct $X1$ in time $O(k \cdot n^2)$; for each pair (i, j) of the k vertex selection blocks in C with $i < j$, and each vertex u in G, we need to construct a gate for each vertex v in G with $v < u$.

- We can construct M in time $O(k \cdot n^2)$; for each pair (i, j) of the k vertex selection blocks in C with $i < j$, and each pair of vertices (u, v) in G, we need to consider whether u is a neighbour of v and, if so, construct a gate.

- We can construct N in time $O(k \cdot n^3)$; for each pair (i, j) of the k vertex selection blocks in C with $i < j$, and each pair of vertices (u, v) in G, we need to consider whether each vertex $w \neq u, w \neq v$ in G is a neighbour of u and of v. If such a w is found we need to construct a gate.

- We can construct $P1$ in time $O(k \cdot n^2)$; for each vertex u in G, we need to consider whether each vertex w in G is a neighbour of u to determine $t = |N[u]|$ and construct the corresponding gate.

- We can construct $P2$ in time $O(k \cdot n^2)$, given $|N[u]|$ for each u in G; for each of $k - 1$ vertex selection blocks in C, each vertex u in G, and each t, $1 \leq t \leq n - |N[u]|$, we need to construct a gate corresponding to u, t and $t' = t + |N[u]|$.

Thus, the transformation can be executed in FPT time, as required, and so we have a parsimonious parameterized counting reduction from #PERFECT CODE to #WEIGHTED WEFT 1, DEPTH h CIRCUIT SATISFIABILITY. $\square$

CHAPTER 9
A COUNTING ANALOG OF THE NORMALIZATION THEOREM

A propositional formula X is *t-normalized* if X is of the form products-of-sums-of-products... of literals with t-alternations. We can consider t-normalized formulas as special examples of weft t circuits, namely the weft t boolean circuits.

Consider the following fundamental parameterized counting problem:

#WEIGHTED t-NORMALIZED SATISFIABILITY

Input:	A t-normalized propositional formula X.
Parameter:	A positive integer k.
Output:	The number of weight k satisfying assignments for X.

In this chapter, we prove the following theorem, which is the counting analog of the NORMALIZATION THEOREM given in [48].

Theorem 9.1 (Normalization theorem, counting version). *For all $t \geq 1$, #WEIGHTED t-NORMALIZED SATISFIABILITY is complete for $\#W[t]$.*

Proof:

Our proof closely follows the proof of the NORMALIZATION THEOREM, making use of a series of parameterized transformations described in [48], with alterations where required to ensure that each of these may be considered as a parsimonious parameterized counting reduction.

We first recall the definition of $\#W[t]$.

Consider the following parameterized counting problem:

#WEIGHTED WEFT t DEPTH h CIRCUIT SATISFIABILITY (#WCS(t, h))

Input:	A weft t depth h decision circuit C.
Parameter:	A positive integer k.
Output:	The number of weight k satisfying assignments for C.

Let $w_{\mathcal{F}(t,h)} : \Sigma^* \times \mathcal{N} \to \mathcal{P}(\Gamma^*)$ be the standard parameterized witness function associated with this counting problem:

$$w_{\mathcal{F}(t,h)}(\langle C, k \rangle) = \{ \text{ weight } k \text{ satisfying assignments for } C \ \} \ .$$

Definition 9.1 (#W[1]). *We define a parameterized counting problem, f_v, to be in #$W[t]$ iff there is a parameterized counting reduction from v, the parameterized witness function for f_v, to $w_{\mathcal{F}(t,h)}$.*

Let $C \in \mathcal{F}(t,h)$ and let k be a positive integer. We want to describe a parameterized transformation that, on input $\langle C, k \rangle$, produces an instance $\langle X, k' \rangle$ of WEIGHTED t-NORMALIZED SATISFIABILITY such that for every weight k input accepted by C there exists a unique weight k' input satisfying X.

Step 1. The reduction to tree circuits.

The first step is to transform C into a *tree circuit*, C', of depth and weft bounded by h and t. Recall, in a tree circuit the input nodes may have large fanout, but every logic gate has fanout one, thus the circuit can be viewed as equivalent to a Boolean formula.

The transformation can be accomplished by replicating the portion of the circuit above a gate as many times as the fanout of the gate, beginning with the top level of logic gates, and proceeding downward level by level. The creation of C' from C may require time $O(|C|^{O(h)})$ and involve a similar blow-up in the size of the circuit. This is permitted under our rules for parameterized transformations since the circuit depth, h, is a fixed constant independent of k and $|C|$.

Note that any weight k input accepted by C will also be accepted by C', any weight k input rejected by C will also be rejected by C'. Thus, the witnesses for C are exactly the witnesses for C'.

Step 2. Moving the *not* gates to the top of the circuit.

Let C denote the circuit we receive from the previous step. Transform C into an equivalent circuit C' by commuting the *not* gates to the top, using DeMorgan's laws. This may increase the size of the circuit by at most a constant factor. The new tree circuit C' thus consists (at the top) of the input nodes with *not* gates on some of the lines fanning out from the inputs. In counting levels we consider all of this as level 0 and may refer to negated fanout lines as negated inputs. Next, there are consisting of only large and small *and* and *or* gates, with a single output gate.

Note that, as was the case for Step 1, any weight k input accepted by C will also be accepted by C', any weight k input rejected by C will also be rejected by C'. Thus, the witnesses for C are exactly the witnesses for C'.

Step 3. Homogenizing the layers.

The goal of this step is to reduce to the situation where all of the large gates are at the bottom of the circuit, in alternating layers of large *And* and *Or* gates. To achieve this, we work from the bottom up, with the first task being to arrange for the output gate to be large.

Let C denote the circuit received from the previous step. Suppose the output gate z is small. Let $C[z]$ denote the connected component of C including z that is induced by the set of small gates. Without loss of generality, we will assume *small* here to be fixed at 2. All the gates providing input to $C[z]$ are either large gates, or input gates, of C. Because of the bound h on the depth of C, there are at most 2^h inputs to $C[z]$. The function of these inputs computed by $C[z]$ is equivalent to a product-of-sums expression E_z having at most 2^{2^h} sums, with each sum a product of at most 2^h inputs.

Let C' denote the circuit equivalent to C obtained by replacing the small gate output component $C[z]$ with the depth 2 circuit representing E_z, duplicating sub-circuits of C as necessary to provide the inputs to this circuit. The *and* gate representing the product of E_z is now the output gate of C'.

This entails a blowup in size by a factor bounded by 2^{2^h}. Since h is an absolutely fixed constant, this blowup is permitted. Note that E_z, and therefore C', are easily computed in a similar amount of time to this size blowup.

Let p denote the output *and* gate of C' (corresponding to the product in E_z). Let $s_1, ..., s_m$ denote the *or* gates of C' corresponding to the sums of E_z. We consider all of these gates to be *small*, since the number of inputs to them does not depend on $|C|$ or k.

Each *or* gate s_i of C' has three kinds of input lines: those coming from large *Or* gates, those coming from large *And* gates, and those coming from input gates of C'. Let these three groups of inputs be denoted

$$S_{i,\vee} = \{s[i,j] \ : \ j = 1, ..., m_{i,\vee}\}$$
$$S_{i,\wedge} = \{s[i,j] \ : \ j = m_{i,\vee}+1, ..., m_{i,\wedge}\}$$
$$S_{i,\top} = \{s[i,j] \ : \ j = m_{i,\wedge}+1, ..., m_{i,\top}\}$$

and define

$$S_i = S_{i,\vee} \cup S_{i,\wedge} \cup S_{i,\top}$$

For convenience, we assume that the inputs are ordered so that those coming from large *Or* gates precede those coming from large *And* gates which, in turn, precede and those coming from input gates of C'.

For each line $s[i,j]$ of C' coming from a large *Or* gate u, let

$$S_{i,\vee,j} = \{s_\vee[i,j,k] \ : \ k = 1, ..., m_{i,\vee,j}\}$$

denote the set of input lines to u in C'.

For each line $s[i,j]$ of C' coming from a large *And* gate u', let

$$S_{i,\wedge,j} = \{s_\wedge[i,j,k] \ : \ k = 1, ..., m_{i,\wedge,j}\}$$

denote the set of input lines to u' in C'.

Let $k' = \sum_{i=1}^{m}(1 + m_{i,\wedge})$.

The integer k' is the number of *or* gates in C' corresponding to the sums of E_z, plus the number of *large* gates that directly supply input to these *or* gates. Note that k' is bounded above by $2^h \cdot 2^{2^h}$.

We now describe how to transform C' into a weft t tree circuit, C'', that accepts a unique input vector of weight $k'' = k + k'$ corresponding to each input vector of weight k accepted by C' (and therefore C). The tree circuit C'' will have a large *And* gate giving the output.

Let $V_0 = \{x_1, ..., x_n\}$ denote the inputs to C'. The circuit C'' has additional input variables that correspond to the input lines to the *or* gates in C' corresponding to the sums of E_z, and the input lines to the *large* gates that directly supply input to these *or* gates.

The set V of input variables for C'' is the union of the following sets of variables:

$$\begin{aligned}
V_0 = &\ \{x_1, ..., x_n\} \text{ the input variables of } C'.\\
V_1 = &\ \{\, u[i,j] \ :\ 1 \le i \le m,\ 1 \le j \le m_{i,\vee}\}\\
V_2 = &\ \{\, u[i,j] \ :\ 1 \le i \le m,\ m_{i,\vee} < j \le m_{i,\wedge}\}\\
V_3 = &\ \{\, u[i,j] \ :\ 1 \le i \le m,\ m_{i,\wedge} < j \le m_{i,\top}\}\\
V_4 = &\ \{\, w[i,j,k] :\ 1 \le i \le m,\ 1 \le j \le m_{i,\vee},\ 0 \le k \le m_{i,\vee,j}\}\\
V_5 = &\ \{\, v[i,j,k] :\ 1 \le i \le m,\ m_{i,\vee} < j \le m_{i,\wedge},\ 0 \le k \le m_{i,\wedge,j}\}
\end{aligned}$$

We first describe how we intend to extend a weight k input vector (on V_0) that is accepted by C', into a unique weight k' input vector (on V) for C''.

Suppose τ is a weight k input vector (on V_0) that is accepted by C', we build τ'' as follows:

1. For each i such that $\tau(x_i) = 1$, set $\tau''(x(i)) = 1$.

2. For each *or* gate s_i, $1 \le i \le m$, corresponding to a sum of E_z in C', choose the lexicographically least index j_i such that the input line $s[i, j_i]$ evaluates to 1 under τ (this is possible, since τ is accepted by C') and set $\tau''(u[i, j_i]) = 1$.

   (a) If, in (2), $s[i, j_i]$ is in $S_{i,\vee}$, i.e. $s[i, j_i]$ is an input line coming from a large *Or* gate, then choose the lexicographically least index k_i such that $s_\vee[i, j_i, k_i]$ evaluates to 1, and set $\tau''(w[i, j_i, k_i]) = 1$.

   Here, we are choosing the first input line to the sum $s[i, j_i]$ that makes it true.

   (b) If, in (2), $s[i, j_i]$ is in $S_{i,\wedge}$ or $S_{i,\top}$, i.e. $s[i, j_i]$ is an input line coming from a large *And* gate or from an input gate of C', then for each $s[i, j']$ in $S_{i,\wedge}$ with $j' < j_i$, choose the lexicographically least index k_i such that $s_\wedge[i, j', k_i]$ evaluates to 0, and set $\tau''(v[i, j', k_i]) = 1$.

   Here, for each product $s[i, j']$ that precedes $s[i, j_i]$, we are choosing the first input line that makes $s[i, j']$ false.

3. For $i = 1, ..., m$ and $j = 1, ..., m_{i,\vee}$ such that $j \neq j_i$, set $\tau''(w[i,j,0]) = 1$.

   For $i = 1, ..., m$ and $j = m_{i,\vee}+1, ..., m_{i,\wedge}$ such that $j \geq j_i$, set $\tau''(v[i,j,0]) = 1$.

4. For all other variables, v, of V, set $\tau''(v) = 0$.

Note that τ'' is a truth assignment to the variables of V having weight $k'' = k + k'$ and that there is exactly one τ'' for V corresponding to each τ of weight k for V_0.

We now build C'' so that the *only* weight k'' input vectors that can be accepted by C'' are the τ'''s that correspond to weight k τ's accepted by C'.

The circuit C'' is represented by a boolean expression.

$$C'' = E_1 \wedge \cdots \wedge E_{8c}$$

For each i, $1 \leq i \leq m$, we must ensure that we set exactly one variable $u[i, j_i]$ to 1, and that j_i is the lexicographically least index such that $s[i, j_i]$ evaluates to 1.

Exactly one j_i for each i:

$$E_1 = \prod_{i=1}^{m} \left(\sum_{j=1}^{m_{i,\top}} u[i,j] \right)$$

$$E_2 = \prod_{i=1}^{m} \prod_{j=1}^{m_{i,\top}-1} \prod_{j'=j+1}^{m_{i,\top}} (\neg u[i,j] + \neg u[i,j'])$$

If $s[i,j]$ is in $S_{i,\vee}$ and $u[i,j] = 1$ then we do not set the default, $w[i,j,0]$. Thus, it must be the case that some $w[i,j,k], k \neq 0$ is able to be chosen, and $s[i,j]$ must be satisfied by τ:

$$E_3 = \prod_{i=1}^{m} \prod_{j=1}^{m_{i,\vee}} (\neg u[i,j] + \neg w[i,j,0])$$

To make the reduction parsimonious we must also ensure that if $s[i,j]$ is in $S_{i,\vee}$ and $u[i,j] = 0$ then we *do* set the default, $w[i,j,0]$:

$$E_{3a} = \prod_{i=1}^{m} \prod_{j=1}^{m_{i,\vee}} (u[i,j] + w[i,j,0])$$

If $s[i,j]$ is in $S_{i,\wedge}$ and $u[i,j] = 1$ then $s[i,j]$ is satisfied by τ:

$$E_4 = \prod_{i=1}^{m} \prod_{j=m_{i,\vee}+1}^{m_{i,\wedge}} \prod_{k=1}^{m_{i,\wedge,j}} (\neg u[i,j] + s_{\wedge}[i,j,k])$$

To make the reduction parsimonious, for each i and j where $s[i,j]$ is in $S_{i,\wedge}$, we must restrict the $v[i,j,k]$ chosen. If we set $u[i,j] = 1$, then for all $s[i,j']$ in $S_{i,\wedge}$ where $j' \geq j$, we set the default $v[i,j,0]$, for all $s[i,j']$ in $S_{i,\wedge}$ where $j' < j$, we do not set the default:

$$E_{4a} = \prod_{i=1}^{m} \prod_{j=1}^{m_{i,\vee}} \prod_{j'=m_{i,\vee}+1}^{m_{i,\wedge}} (\neg u[i,j] + v[i,j',0])$$
$$E_{4b} = \prod_{i=1}^{m} \prod_{j=m_{i,\vee}+1}^{m_{i,\wedge}} \prod_{j'=j}^{m_{i,\wedge}} (\neg u[i,j] + v[i,j',0])$$

If $s[i,j]$ is in $S_{i,\top}$ and $u[i,j] = 1$ then $s[i,j]$ is satisfied by τ:

$$E_5 = \prod_{i=1}^{m} \prod_{j=m_{i,\wedge}+1}^{m_{i,\top}} (\neg u[i,j] + s[i,j])$$

If $u[i,j] = 1$ then all other choices $u[i,j']$ with $j' < j$ must have $s[i,j']$ not satisfied by τ. In the case where $s[i,j'] \in S_{i,\vee}$, we ensure directly that none of the input lines in $S_{i,\vee,j'}$ evaluate to 1. In the case where $s[i,j'] \in S_{i,\wedge}$, we ensure that the default variable $v[i,j',0] \neq 1$, therefore forcing some $v[i,j',k]$ with $k \neq 0$ to be set to 1. In the case where $s[i,j'] \in S_{i,\top}$, we ensure directly that $s[i,j']$ doesn't evaluate to 1:

$$E_{6a} = \prod_{i=1}^{m} \prod_{j=1}^{m_{i,\vee}} \prod_{j'=1}^{j-1} \prod_{k=1}^{m_{i,\vee,j'}} (\neg u[i,j] + \neg s_{\vee}[i,j',k])$$
$$E_{6b} = \prod_{i=1}^{m} \prod_{j=m_{i,\vee}+1}^{m_{i,\top}} \prod_{j'=1}^{m_{i,\vee}} \prod_{k=1}^{m_{i,\vee,j'}} (\neg u[i,j] + \neg s_{\vee}[i,j',k])$$
$$E_{6c} = \prod_{i=1}^{m} \prod_{j=m_{i,\vee}+1}^{m_{i,\wedge}} \prod_{j'=m_{i,\vee}+1}^{j-1} (\neg u[i,j] + \neg v[i,j',0])$$
$$E_{6d} = \prod_{i=1}^{m} \prod_{j=m_{i,\wedge}+1}^{m_{i,\top}} \prod_{j'=m_{i,\vee}+1}^{m_{i,\wedge}} (\neg u[i,j] + \neg v[i,j',0])$$
$$E_{6e} = \prod_{i=1}^{m} \prod_{j=m_{i,\wedge}+1}^{m_{i,\top}} \prod_{j'=m_{i,\wedge}+1}^{j-1} (\neg u[i,j] + \neg s[i,j'])$$

For each $s[i,j]$ that is in $S_{i,\vee}$, we must set exactly one variable $w[i,j,k]$ to 1. If we set $w[i,j,k_i] = 1$, where $k_i \neq 0$, then we need to ensure that k_i is the lexicographically least index such that $s_{\vee}[i,j,k_i]$ evaluates to 1.

$$E_{7a} = \prod_{i=1}^{m} \prod_{j=1}^{m_{i,\vee}} \left(\sum_{k=0}^{m_{i,\vee,j}} w[i,j,k] \right)$$
$$E_{7b} = \prod_{i=1}^{m} \prod_{j=1}^{m_{i,\vee}} \prod_{k=0}^{m_{i,\vee,j}-1} \prod_{k'=k+1}^{m_{i,\vee,j}} (\neg w[i,j,k] + \neg w[i,j,k'])$$
$$E_{7c} = \prod_{i=1}^{m} \prod_{j=1}^{m_{i,\vee}} \prod_{k=1}^{m_{i,\vee,j}} (\neg w[i,j,k] + s_{\vee}[i,j,k])$$
$$E_{7d} = \prod_{i=1}^{m} \prod_{j=1}^{m_{i,\vee}} \prod_{k=1}^{m_{i,\vee,j}} \prod_{k'=1}^{k-1} (\neg w[i,j,k] + \neg s_{\vee}[i,j,k'])$$

For each $s[i,j]$ that is in $S_{i,\wedge}$, we must set exactly one variable $v[i,j,k]$ to 1. If we set $v[i,j,k_i] = 1$, where $k_i \neq 0$, then we need to ensure that k_i is the lexicographically least index such that $s_{\wedge}[i,j,k_i]$ evaluates to 0.

$$E_{8a} = \prod_{i=1}^{m} \prod_{j=m_{i,\vee}+1}^{m_{i,\wedge}} \left(\sum_{k=0}^{m_{i,\wedge,j}} v[i,j,k] \right)$$

$$E_{8b} = \prod_{i=1}^{m} \prod_{j=m_{i,\vee}+1}^{m_{i,\wedge}} \prod_{k=0}^{m_{i,\wedge,j}-1} \prod_{k'=k+1}^{m_{i,\wedge,j}} (\neg v[i,j,k] + \neg v[i,j,k'])$$

$$E_{8c} = \prod_{i=1}^{m} \prod_{j=m_{i,\vee}+1}^{m_{i,\wedge}} \prod_{k=1}^{m_{i,\wedge,j}} (\neg v[i,j,k] + \neg s_{\wedge}[i,j,k])$$

$$E_{8d} = \prod_{i=1}^{m} \prod_{j=m_{i,\vee}+1}^{m_{i,\wedge}} \prod_{k=1}^{m_{i,\wedge,j}} \prod_{k'=1}^{k-1} (\neg v[i,j,k] + s_{\wedge}[i,j,k'])$$

We now argue that the transformation described here, $\langle C', k\rangle \rightarrow \langle C'', k''\rangle$, has the properties required:

Claim 1. It is evident that the size of C'' is bounded by $|C'|^2$.

Claim 2. The circuit C'' has weft t.

Firstly, recall that we are assuming here $t \geq 2$. Any input-output path in C'' beginning from a new input variable v in V (i.e. $v \in V_1, \ldots, V_5$) has at most two large gates, since the expression for C'' is essentially a product-of-sums.

Some of the subexpressions of C'' involve subcircuits of C' These subcircuits of C' are either of the form $s_{\vee}[i,j,k], s_{\wedge}[i,j,k]$, i.e. inputs to large gates in C', or of the form $s[i,j]$, $m_{i,\wedge} < j \leq m_{i,\top}$, i.e. original input variables of C'. Note that, all of these subcircuits have weft at most $t-1$, since the weft of C' is bounded by t.

Any input-output path in C'' beginning from an original input variable v in V (i.e. $v \in V_0$) passes through one of these subcircuits and on to a small *or* gate, and then further encounters only the bottommost large *And* gate. Thus, any input-output path in C'' beginning from an original input variable has at most t large gates.

Claim 3. The circuit C'' accepts a unique input vector of weight k'' corresponding to each input vector of weight k accepted by C'.

If τ is a witness for $\langle C', k\rangle$ (that is, τ is a weight k input vector on V_0 that is accepted by C'), then τ'' is a witness for $\langle C'', k''\rangle$ (that is, τ'' is a weight k'' input vector on V that satisfies C''). This is evident from the description of C'' and τ'' given above.

In addition, we argue that τ' is the only extension of τ that is a witness for $\langle C'', k''\rangle$. Suppose τ'' is an extension of τ, and that τ is accepted by C':

1. The clauses E_1 and E_2 ensure that, for each i, $1 \le i \le m$, we must choose exactly one index j_i and set $\tau''(u[i, j_i]) = 1$, and we must set $\tau''(u[i, j']) = 0$ for all $j' \neq j_i$. The clauses E_3, E_4, and E_5 ensure that if $u[i, j_i]$ is chosen, then the input line $s[i, j_i]$ evaluates to 1 in C'. The clauses $E_{6a}, ..., E_{6e}$ ensure that j_i must be the lexicographically least possible index.

   (a) If, in (1), $s[i, j_i]$ is an input line coming from a large *Or* gate, then the clauses E_3 and $E_{7a}, \ldots, E_{7d}$ ensure that we must choose the lexicographically least index $k_i \neq 0$ such that $s_\vee[i, j_i, k_i]$ evaluates to 1, and set $\tau''(w[i, j_i, k_i]) = 1$, and we must set $\tau''(w[i, j_i, k']) = 0$ for all $k' \neq k_i$.

   Clauses E_{3a} and E_{7a}, E_{7b} ensure that for $j = 1, ..., m_{i,\vee}$ such that $j \neq j_i$, we must set $\tau''(w[i, j, 0]) = 1$, and $\tau''(w[i, j, k]) = 0$ for all $k \neq 0$.

   Clauses E_{4a}, E_{4b} and E_{8a}, E_{8b} ensure that for $j = m_{i,\vee} + 1, ..., m_{i,\wedge}$ we must set $\tau''(v[i, j, 0]) = 1$, and $\tau''(v[i, j, k]) = 0$ for all $k \neq 0$.

   (b) If, in (1), $s[i, j_i]$ is an input line coming from a large *And* gate or an input gate of C', then clauses E_{6c} and $E_{8a}, \ldots, E_{8d}$ ensure that for each input line $s[i, j']$ in $S_{i,\wedge}$ with $j' < j_i$, we must choose the lexicographically least index $k_i \neq 0$ such that $s_\wedge[i, j', k_i]$ evaluates to 0, and set $\tau''(v[i, j', k_i]) = 1$, and that for all $k' \neq k_i$ we must set $\tau''(v[i, j', k']) = 0$.

   Clauses E_{4a}, E_{4b} and E_{8a}, E_{8b} ensure that for all $j = m_{i,\vee} + 1, ..., m_{i,\wedge}$ such that $j \ge j_i$, we must set $\tau''(v[i, j, 0]) = 1$, and $\tau''(v[i, j, k]) = 0$ for all $k \neq 0$.

   Clauses E_{3a} and E_{7a} ensure that for $j = 1, ..., m_{i,\vee}$ we must set $\tau''(w[i, j, 0]) = 1$, and $\tau''(w[i, j, k]) = 0$ for all $k \neq 0$.

Now suppose v' is witness for $\langle C'', k''\rangle$. We argue that the restriction v of v' to V_0 is a witness for $\langle C', k\rangle$.

1. v' sets exactly k variables of V_0 to 1.

   Let

$$\begin{aligned} V_i = &\quad \{\, u[i,j] \;:\; 1 \le j \le m_{i,\top}\} \\ V_{ij} = &\quad \{\, w[i,j,k] :\; 0 \le k \le m_{i,\vee,j}\} \text{ if } 1 \le j \le m_{i,\vee} \\ V_{ij} = &\quad \{\, v[i,j,k] :\; 0 \le k \le m_{i,\wedge,j}\} \text{ if } m_{i,\vee}+1 \le j \le m_{i,\wedge} \end{aligned}$$

Any input vector of weight k'' accepted by C'' must set exactly one variable from each of the sets of variables V_i (for $i = 1, \ldots, m$) and V_{ij} (for $i = 1, \ldots, m$ and $j = 1, \ldots, m_{i,\wedge}$) to 1. Any such accepted input must set exactly k variables of V_0 to 1, by definition of k''.

2. The restriction v of v' to V_0 is accepted by C'.

   It is enough to show that for each $i = 1, \ldots, m$, some input line of s_i in C' is set to 1 by v.

   v' must set exactly one variable from each of the sets of variables V_i (for $i = 1, \ldots, m$) to 1. If $u[i,j] \in V_1$ from V_i is set to 1, then the clauses of E_3 ensure that $w[i,j,0]$ is not set to 1. This being the case, the clauses of E_{7a} and E_{7b} ensure that exactly one variable, $w[i,j,k]$, with $k \neq 0$, from V_{ij} is set to 1, and, in turn, the clauses of E_{7c} ensure that, in this case, the kth input line to $s[i,j]$ must evaluate to 1. Since $s[i,j]$ is a large *Or* gate, this will force the jth input line of s_i to evaluate to 1. If $u[i,j] \in \{V_2 \cup V_3\}$ from V_i is set to 1 then the clauses of E_4, or E_5 will ensure that the jth input line of s_i evaluates to 1.

This completes the argument that the transformation, $\langle C', k\rangle \rightarrow \langle C'', k''\rangle$, has the properties required. □

We may now assume that the circuit with which we are working has a large output gate (which may be of either denomination, since we only performed the transformation described above if the output gate z of our original circuit C was small).

Renaming for convenience, let C denote the circuit with which we are now working.

If g and g' are gates of the same logical character ($\vee$ or $\wedge$) with the output of g going to g', then they can be consolidated into a single gate without increasing the weft if g' is large. We term this a *permitted contraction.* Note that if g is large and g' is small, then the contraction may not preserve weft. We will assume that permitted contractions are performed whenever possible, interleaved with the following two operations:

(1) *Replacement of bottommost small gate components.*

Let $C_1, \ldots, C_m$ denote the bottommost connected components of C induced by the set of small gates, and having at least one large gate input. Since the output gate of C is large, each C_i gives output to a large gate g_i. If g_i is an *And* gate, then C_i should be replaced with equivalent sum-of-products circuitry. If g_i is an*Or* gate, then C_i should be replaced with product-of-sums circuitry. In either case, this immediately creates the opportunity for a permitted contraction. The replacement circuitry is small, since each gate in C_i is small, and the number of levels in C_i is bounded by a fixed constant h. This operation will increase the size of the circuit by a factor of 2^{2^h}. This step will be repeated at most h times, as we are working from the bottom up in transforming C.

(2) *Commuting small gates upward.*

After (1), and after permitted contractions, each bottommost small gate component of the modified circuit C' is, in fact, a single small gate, h_i, giving output to a large gate g_i. Without loss of generality, all of the input lines to h_i may be considered to come from large gates. The other possibility is that an input line may come from the input level of the circuit, but there is no increase in weft in treating this as an honorary large gate (of either denomination) for convenience.

Suppose that g_i is an *And* gate and that h_i is an *or* gate (the other possibility is handled dually).

There are three possible cases:

1. All of the input lines to h_i are from large *Or* gates.

2. All of the input lines to h_i are from large *And* gates.

3. The input lines to h_i are from both large *Or* gates and large *And* gates.

In case 1, we may consolidate h_i and all of the gates giving input to h_i into a single large *Or* gate without increasing weft.

In case 2, we replace the small $\vee$ (h_i) of large $\wedge$'s with the equivalent (by distribution) large $\wedge$ of small $\vee$'s. Since h_i may have 2^h inputs, this may entail a blowup in the size of the circuit from $|C'|$ to $|C'|^{2^h}$. This does not increase weft, and creates the opportunity for a permitted contraction.

In case 3, we similarly replace h_i and its input gates with circuitry representing a product-of-sums of the inputs to the large input gates of h_i. In this case, the replacement is a large $\wedge$ of large (rather than small) $\vee$ gates. Weft is preserved when we take advantage of the contraction now permitted between the large $\wedge$ gate and g_i.

We may need to repeat the cycle of (1) and (2), but at most h repetitions are required. The total blowup in the size of the circuit after at most h repetitions of the cycle is crudely bounded by $|C|^{2^{h^2}}$.

Step 4. Removing a bottommost *Or* gate.

The goal of this step is to obtain a homogenized circuit with large *And* output gate.

Let C be a weft t ($t \geq 2$) tree circuit with large *Or* output gate that we receive at the beginning of this step. We transform C into a weft t tree circuit, C', that has a large *And* output gate, and accepts a unique input vector of weight $k+1$ corresponding to each input vector of weight k accepted by C.

Let b be the number of input lines to the large *Or* output gate of C. Let $C_1, \ldots, C_b$ be the subcircuits corresponding to each of these input lines.

The set V of input variables for C' is the union of the following two sets of variables:

$V_0 =$ $\{x_1, ..., x_n\}$ the variables of C.

$V_1 =$ $\{\, y[1], \ldots, y[b]\}$

The new circuit C' is represented by the expression $E_1 \wedge E_2 \wedge E_3 \wedge E_4$ where:

$$E_1 = \sum_{i=1}^{b} y[i]$$
$$E_2 = \prod_{i=1}^{b-1} \prod_{j=i+1}^{b} (\neg y[i] + \neg y[j])$$
$$E_3 = \prod_{i=1}^{b} (\neg y[i] + C_i)$$
$$E_4 = \prod_{i=1}^{b} \prod_{j=1}^{i-1} (\neg y[i] + \neg C_j)$$

Each subcircuit C_i must have weft at most $t-1$ in C, and therefore in C'. Thus, C' has weft at most t.

C', accepts a unique input vector of weight $k+1$ corresponding to each input vector of weight k accepted by C.

Suppose that τ is a weight k input vector accepted by C. We build τ', a weight $k+1$ input vector accepted by C', as follows:

1. Set $\tau'(x_i) = 1$ if $\tau(x_i) = 1$, $\tau'(x_i) = 0$ if $\tau(x_i) = 0$.
2. Choose the lexicographically least index i such that C_i evaluates to 1 under τ and set $\tau'(y[i]) = 1$, set $\tau'(y[j]) = 0$ for $j = 1, \ldots, b$ with $j \neq i$.

For any input vector accepted by C':

1. E_1 and E_2 ensure that exactly one of the variables in V_1 must be set to 1,
2. E_3 ensures that if $y[i]$ is set to 1 then C_i must evaluate to 1,
3. E_4 ensures that if $y[i]$ is set to 1 then i must be the lexicographically least index i such that C_i evaluates to 1.

Thus, if τ is a witness for $\langle C, k\rangle$ then τ' is the only extension of τ that is a witness for $\langle C', k+1\rangle$. If τ' is a witness for $\langle C', k+1\rangle$ then the restriction τ of τ' to V_0 is a witness for $\langle C, k\rangle$.

The transformation yields a circuit with large *And* output gate, but possibly with *not* gates at lower levels. Thus, it may be necessary to repeat steps 2 and 3 to obtain a homogenized circuit with bottommost *And* gate.

Step 5. Organizing the small gates.

Let C be the tree circuit received from the previous step. C has the following properties:

1. the output gate is an *And* gate,
2. from the bottom, the circuit consists of layers which alternately consist of only large *And* gates or only large *Or* gates, for up to t layers, and
3. above this, there are branches B of height $h' = h - t$ consisting only of small gates.

Since a small gate branch B has depth at most h', it has at most $2^{h'}$ gates. Thus, in constant time (since h is fixed), we can find either an equivalent sum-of-products circuit with which to replace B, or an equivalent product-of-sums circuit with which to replace B.

Suppose the topmost level of large gates in C consists of *Or* gates. In this case, we replace each small gate branch B with an equivalent depth 2 sum-of-products circuit. The replacement circuitry is small, so the weft of the circuit will be preserved by this transformation. Each replacement circuit has a bottommost *or* gate, g_B, of fannin at most $2^{2^{h'}}$, and the *and* gates feeding into g_B have fannin at most $2^{h'}$. The transformation may entail a blowup in size by a factor bounded by $2^{2^{h'}}$.

Each g_B can be merged into the topmost level of large *Or* gates of C to produce C'. C' has t homogenized alternating layers of large gates, above which lies a single layer of small *and* gates.

Suppose the topmost level of large gates in C consists of *And* gates. In this case, we replace each small gate branch B with an equivalent depth 2 product-of-sums circuit and merge each g_B into the topmost level of large *And* gates of C to produce C'. C' has t homogenized alternating layers of large gates, above which lies a single layer of small *or* gates.

Note that any weight k input accepted by C will also be accepted by C', any weight k input rejected by C will also be rejected by C'. Thus, the witnesses for C are exactly the witnesses for C'.

Step 6. MONOTONE or ANTIMONOTONE conversion (two cases).

Let C be the circuit received from the previous step. Recall that C has t homogenized alternating layers of large gates, above which lies a single homogenized layer of small gates, and that C has a large *And* output gate. We can also assume that any *not* gates in C occur at the top level 0.

The goal in this step is to obtain a circuit having the same properties but that, in addition, is either MONOTONE (i.e. no inverters in the circuit) or (nearly) ANTIMONOTONE (i.e. all inputs to the level 1 small gates are negated with no other inverters in the circuit). The special character of the circuit thus constructed will enable us to obtain a circuit with the remaining single level of small gates eliminated, using the final transformation described in Step 7.

We consider two cases, depending on whether the topmost level of large gates in our received circuit C consists of *Or* gates or *And* gates.

In both cases, we employ the following key step in the transformation:

A monotone change of variables.

Let C be a circuit with input variables having both positive and negative fanout lines, and with large *And* output gate. We describe here a transformation that results in a circuit C' with input variables having only positive fanout lines. C' accepts a unique input vector of weight $2k$ corresponding to each input vector of weight k accepted by C.

Let $X = x[0], \ldots, x[n-1]$ be the set of input variables of C. Recall the size (n, k) *selection gadget*, $G_{n,k}$, described earlier. Each size $2k$ dominating set of $G_{n,k}$ corresponds to a unique weight k truth assignment for the set of n variables, $x[0], \ldots, x[n-1]$.

We base the design of C' on $G_{n,k}$ and C.

1. The set of input variables for C' corresponds to the set of vertices of $G_{n,k}$.

2. Corresponding to each positive fanout line from $x[i]$ in C there is an *Or* gate in C' with k input lines coming from the k variables representing vertices whose

presence in a size $2k$ dominating set of $G_{n,k}$ would represent the setting of $x[i]$ to true in the corresponding weight k truth assignment for $x[0], \ldots, x[n-1]$.

3. Corresponding to each negated fanout line from $x[i]$ in C there is an *Or* gate in C' with $O(n^{2})$ input lines coming from the variables representing vertices whose presence in a size $2k$ dominating set of $G_{n,k}$ would represent the setting of $x[i]$ to false in the corresponding weight k truth assignment for $x[0], \ldots, x[n-1]$.

4. Below these level 1 large *Or* gates C' has the same structure as C, with the output lines from the level 1 *Or* gates replacing the fanout lines from the original input variables of C.

5. Merged with the large *And* output gate of C' is a new circuit branch corresponding to the product-of-sums expression,

$$\prod_{u \in G_{n,k}} \sum N[u]$$

where $N[u]$ is the closed neighborhood of vertex u. The inputs to each of the sums are the input variables of C' corresponding to the vertices of each $N[u]$ in $G_{n,k}$. This ensures that any accepted input vector for C' must represent a dominating set of $G_{n,k}$.

Note that in any negated fanout line from an input variable in C has been replaced in C' by an *Or* gate with only positive inputs.

We now argue that this transformation is parsimonious.

Suppose that τ is a witness for $\langle C, k\rangle$. There is exactly one dominating set of size $2k$ for $G_{n,k}$ corresponding to τ. This dominating set corresponds to exactly one weight $2k$ input vector τ' for C', where vertices of $G_{n,k}$ are replaced by corresponding input variables for C'.

We argue that τ' must be accepted by C':

For $r = 0, \ldots, k-1$, if $x[i]$ is the rth variable of X set to true by τ then the vertex representing this fact ($a[r, i]$) must be present in the corresponding dominating set d_τ of $G_{n,k}$, and the corresponding variable must be set to true in τ'. In this case, the rth input line to each *Or* gate in C' representing a positive fanout from $x[i]$ in C must be set to 1. Note that, if $x[i]$ is set to true by τ, then no vertex representing the fact that $x[i]$ is false ($b[r, s, t]$ with $s < i$ and $s + t + 1 > i$) can be present in d_τ so none of the input lines to any *Or* gate in C' representing a negated fanout from $x[i]$ in C will be set to true.

For $r = 0, \ldots, k - 1$, if $x[i]$ is in the rth 'gap' of variables of X set to false by τ then a vertex representing this fact must be present in the corresponding dominating set d_τ of $G_{n,k}$, and the corresponding variable must be set to true in τ', this variable provides an input line to each *Or* gate in C' representing a negated fanout from $x[i]$ in C. If $x[i]$ is set to false by τ, then no vertex representing the fact that $x[i]$ is true can be present in d_τ so none of the input lines to any *Or* gate in C' representing a positive fanout from $x[i]$ in C will be set to true.

In either case, the output of each of the level 1 *Or* gates in C' reflects the output of the corresponding (positive or negative) fanout line from an original input variable of C. Since τ is accepted by C, it must be the case that the outputs of the level 1 *Or* gates satisfy the subcircuit C in C'.

Finally, since τ' represents a dominating set of $G_{n,k}$, the new product-of-sums circuit branch in C' will evaluate to true under τ'.

Suppose that τ' is a witness for $\langle C', 2k \rangle$. τ' corresponds to exactly one dominating set of size $2k$ for $G_{n,k}$. This dominating set corresponds to exactly one weight k input vector τ for C.

As argued above, the outputs from the (positive or negative) fanout lines of input variables of C under τ must reflect the outputs of the corresponding *Or* gates of C' under τ'. Since the outputs of these *Or* gates satisfy the subcircuit C in C' it must be the case that τ is accepted by C.

We assumed, at the beginning of this step, that all of the *not* gates of the received circuit C were at the top level 0. Any such *not* gate produced a negated fanout line from an input variable of C. If we employ the transformation described above, then each of these negated fanout lines in C will be replaced in C' by an *Or* gate with only positive inputs. Thus, the resulting circuit C' will be MONOTONE. However, the new level of large *Or* gates introduced by the transformation may result in C'

having weft $\geq t$.

We now show how this transformation can be employed to produce a circuit C' having all the properties we require for the input circuit to Step 7. That is, C' will consist of t homogenized alternating layers of large gates, above which lies a single homogenized layer of small gates, and C' will have a large *And* output gate. In addition, C' will be either MONOTONE (i.e. no inverters in the circuit) or (nearly) ANTIMONOTONE (i.e. all inputs to the level 1 small gates will be negated with no other inverters in the circuit).

We consider two cases, depending on whether the topmost level of large gates in our original circuit C consists of *Or* gates or *And* gates.

1. The topmost level of large gates in C consists of *Or* gates.

   We perform the transformation $C \rightarrow C'$ that we have described above. Now, from the top, C' consists of a layer of input variables (none of which are negated), below this lies a (new) layer of large *Or* gates at level 1, a layer of small *and* gates at level 2, and a layer of (original) large *Or* gates at level 3. Also present in C' are the enforcement *Or* gates with input lines coming from the input variables of C' and outputs going to the large *And* output gate of C'.

   We create a new circuit C'' by replacing each (level 2) small $\wedge$ of large $\vee$'s with the equivalent (by distribution) large $\vee$ of small $\wedge$'s, and then merging the level 2 layer of large *Or* gates thus created with those on level 3.

   The resulting circuit C'' has the form required with weft at most t, and accepts a unique input vector of weight $2k$ corresponding to each input vector of weight k accepted by C (since the witnesses for $\langle C'', 2k\rangle$ are exactly the witnesses for $\langle C', 2k\rangle$).

2. The topmost level of large gates in C consists of *And* gates.

   In this case, we first introduce a double layer of *not* gates below the topmost level of large *And* gates. Suppose the output of each *And* gate, A_i, in this topmost large-gate level goes to a large *Or* gate, B_i (note that, since the weft of C is at least 2, this must be the case). We will transform C so that the output of each A_i goes to a *not* gate $n_{i,1}$, the output of $n_{i,1}$ goes to $n_{i,2}$, and the output of $n_{i,2}$ goes to B_i.

The first layer of *not* gates in the resulting circuit are now commuted to the top (level 0), using DeMorgan's laws. This results in a circuit with all inputs negated, a layer of small *and* gates at level 1, and a layer of large *Or* gates at level 2, each with a negated output line going to a large *Or* gate at level 3. Note that the size of the resulting circuit is increased relative to C by at most a constant factor.

Renaming, let C be the circuit we have now constructed. We perform the monotone change of variables transformation $C \rightarrow C'$ described above.

Now, from the top, C' consists of a layer of input variables (none of which are negated), below this lies a (new) layer of large *Or* gates at level 1, a layer of small *and* gates at level 2, and a layer of (original) large *Or* gates at level 3, each with a negated output line going to a large *Or* gate at level 4. Also present in C' are the enforcement *Or* gates with input lines coming from the input variables of C' and outputs going to the large *And* output gate of C'.

As in case 1, we create a new circuit C'' by replacing each (level 2) small $\wedge$ of large $\vee$'s with the equivalent (by distribution) large $\vee$ of small $\wedge$'s, and then merging the level 2 layer of large *Or* gates thus created with those on level 3.

Finally, the remaining *not* gates, lying below the merged layer of *Or* gates in C'', are commuted to the top. This results in a circuit C^* with input variables that have negated fanout lines going to a layer of small *or* gates at level 1, and positive fanout lines going to the enforcement *Or* gates. The topmost level of large gates in C^* (level 2) consists of *And* gates.

The resulting circuit C^* has the form required with weft at most t, and accepts a unique input vector of weight $2k$ corresponding to each input vector of weight k accepted by C.

Step 7. Eliminating the remaining small gates.

Let C be the MONOTONE or (nearly) ANTIMONOTONE circuit received from the previous step. In this step we transform C into a circuit C' corresponding directly to a t-normalized boolean expression (that is, consisting of only t homogenized alternating layers of large *And* and *Or* gates). We will ensure that C' accepts a unique weight $k' = k \cdot 2^{k+1} + 2^k$ input vector corresponding to each weight k input vector accepted by C.

Suppose that C has m remaining small gates. If C is a MONOTONE circuit, with the topmost level of large gates consisting of *Or* gates, then C must have odd weft, bounded by t. In this case, the small (level 1) gates are *and* gates, and the inputs are all positive. If C is a (nearly) ANTIMONOTONE circuit, with the topmost level of large gates consisting of *And* gates, then C must have even weft, bounded by t. In this case, the small (level 1) gates are *or* gates, and the inputs are all negated.

For $i = 1, \ldots, m$ we define the sets $A_1, \ldots, A_m$ to be the sets of input variables to these small gates.

Suppose that g_i is a small *and* gate in C with inputs a, b, c, and d, that is, $A[i] = \{a, b, c, d\}$. The gate g_i can be eliminated by replacing it with an input line from a new variable $v[i]$ which represents the predicate $a = b = c = d = 1$. We will use additional circuit structure to ensure that $v[i]$ is set to true if and only if a, b, c, and d are also all set to true.

Suppose that g_i is a small *or* gate in C with inputs $\neg a$, $\neg b$, $\neg c$, and $\neg d$. The gate g_i can be eliminated by replacing it with a negated input line from $v[i]$.

Let $x[j]$ for $j = 1, \ldots, s$ be the input variables to C. The input variables for C' are the following:

1. one variable $v[i]$ for each of the sets A_i, $i = 1, \ldots, m$ to be used as described above
2. for each $x[j]$, 2^{k+1} copies of $x[j]$, $x[j, 0], x[j, 1], \ldots, x[j, 2^{k+1} - 1]$
3. "padding" variables, consisting of 2^k variables that are not, in fact, inputs to any gates in C', $z[1], \ldots, z[2^k]$

To produce C', we take our original circuit C and, for $i = 1, \ldots, s$, let $x[j, 0]$ take the place of $x[j]$ as an input to any large gate. We remove all the small gates on level 1 and replace these with appropriate input lines from the $v[i]$'s to large gates. We then add an enforcement mechanism for the change of variables. The necessary requirements can be easily expressed in product-of-sums form, so can be incorporated into the bottom two levels of the circuit as additional *Or* gates attached to the output *And* gate. We require the following implications concerning the new input variables:

1. for $j = 1, \dots, s$ and $r = 0, \dots, 2^{k+1} - 1$, $x[j, r] \Rightarrow x[j, r + 1 (\bmod\ 2^{k+1})]$
2. for each containment $A_i \subseteq A_{i'}$, $v[i'] \Rightarrow v[i]$
3. for each membership $x[j] \in A_i$, $v[i] \Rightarrow x[j, 0]$
4. for $i = 1, \dots, m$, $\left(\prod_{x[j] \in A_i} x[j, 0] \right) \Rightarrow v[i]$
5. for $w = 2, \dots, 2^k$, $z[w] \Rightarrow z[w - 1]$

This transformation may increase the size of the circuit by a linear factor exponential in k. We now argue that C' accepts a unique weight $k' = k \cdot 2^{k+1} + 2^k$ input vector corresponding to each weight k input vector accepted by C.

Suppose C accepts a weight k input vector, then setting the corresponding copies $x[i, j]$ among the new input variables accordingly, together with the appropriate settings for the new collective variables, the $v[i]$'s, yields a vector of weight between $k \cdot 2^{k+1}$ and $k \cdot 2^{k+1} + 2^k$ that is accepted by C'. The reason that the weight of this corresponding vector may fall short of $k' = k \cdot 2^{k+1} + 2^k$ is that not all of the subsets of the k input variables to C having value 1 may occur among the sets A_i. We obtain a vector of weight exactly k' by setting some of the "padding" variables, the $z[w]$'s, to true, if necessary. Note that there is only one way of choosing which padding variables to use, we must choose the required number of variables from the beginning of the sequence $z[1], z[2], \dots, z[2^k]$ since, for $w = 2, \dots, 2^k$, $z[w] \Leftarrow z[w - 1]$.

Suppose that C' accepts a vector of weight k'. Because of the implications in (1) above, exactly k sets of copies of inputs to C must have value 1 in the accepted vector. Because of the implications (2)-(4), the variables $v[i]$, for $i = 1, \dots, m$, must have values in the accepted input vector compatible with the values of the sets of copies. By the construction of C', this implies that the weight k input vector $x[j_1], x[j_2], \dots, x[j_k]$, corresponding to the k sets of copies must be accepted by C.

This concludes the proof of theorem 9.1. $\square$

References

[1] K. A. Abrahamson and M. R. Fellows: *Finite automata, bounded treewidth, and well-quasi-ordering.* Graph Structure Theory, editors N. Robertson and P. Seymour, Contempory Mathematics Vol 147, American Mathematical Society, pp 539-564, 1993.

[2] J. Alber: *Exact Algorithms for NP-hard Problems on Planar and Related Graphs: Design, Analysis, and Implementation.* PhD thesis in preparation, Universität Tübingen, Germany, 2002.

[3] J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier: *Fixed parameter algorithms for dominating set and related problems on planar graphs.* Algorithmica 33, pp 461-493, 2002.

[4] J. Alber, H. Fernau, and R. Niedermeier: *Parameterized complexity: exponential speed-up for planar graph problems.* Proc. 28th ICALP, Springer-Verlag LNCS 2368, pp150-159, 2002.

[5] N. Alon, R. Yuster, and U. Zwick: *Color-coding.* Journal of the ACM 42 (4), pp 844-856, 1995.

[6] A. Andrzejak *An algorithm for the Tutte polynomials of graphs of bounded treewidth.* Discrete Math. 190, pp 39-54, 1998.

[7] S. Arnborg, D. G. Corneil, and A. Proskurowski: *Complexity of finding embeddings in a k-tree.* SIAM J. Alg. Disc. Meth. 8, pp 277-284, 1987.

[8] S. Arnborg and A. Proskurowski: *Characterization and recognition of partial 3-trees.* SIAM J. Alg. Disc. Meth. 7, pp 305-314, 1986.

[9] V. Arvind and V. Raman: *Approximate Counting small subgraphs of bounded treewidth and related problems.* ECCC Report TR02-031, 2002.

[10] R. Balasubramanian, R. Downey, M. Fellows, V. Raman: unpublished manuscript.

[11] R. Balasubramanian, M. Fellows, V. Raman: *An improved fixed parameter algorithm for vertex cover.* Information Processing Letters 65 (3), pp163-168, 1998.

[12] C. Bazgan: *Schémas d'approximation et complexité paramétrée.* Rapport de stage de DEA d'Informatique à Orsay, 1995.

[13] H. L. Bodlaender: *NC-algorithms for graphs with small treewidth.* Proceedings of the 14th International Workshop on Graph-Theoretic Concepts in Computer Science WG'88, J. van Leeuwen (Ed.), Vol 344 LNCS, Springer-Verlag, pp 1-10, 1988.

[14] H. L. Bodlaender: *A linear time algorithm for finding tree decompositions of small treewidth.* SIAM J. Comput. 25, pp 1305-1317, 1996.

[15] H. L. Bodlaender: *A partial k-arboretum of graphs with bounded treewidth.* Technical Report UU-CS-1996-02, Department of Computer Science, Utrecht University, Utrecht, 1996.

[16] H. L. Bodlaender: *Treewdith: Algorithmic techniques and results.* Proc. 22nd MFCS, Springer-Verlag LNCS 1295, pp 19-36, 1997.

[17] H. L. Bodlaender: *A note on domino treewidth.* Discrete Math. and Theor. Comp. Sci. 3, pp 141-150, 1999.

[18] H. L. Bodlaender and J. Engelfreit: *Domino Treewidth.* J. Algorithms 24, pp 94-127, 1997.

[19] H. L. Bodlaender, M. R. Fellows, and M. T. Hallett: *Beyond NP-completeness for problems of bounded width: Hardness for the W-hierarchy.* Proceedings of the 26th Annual Symposium on Theory of Computing, pp 449-458, ACM Press, New York, 1994.

[20] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks: *Approximating treewidth, pathwidth, and minimum elimination tree height.* J. Algorithms 18, pp 238-255, 1995.

[21] H. L. Bodlaender and T. Hagerup: *Parallel algorithms with optimal speedup for bounded treewidth.* Procdeedings of the 22nd International Colloquium on Automata, Languages, and Programming, Z. Fülöp and F. Gécseg (Eds.), Vol 944 LNCS, Springer-Verlag, pp 268-279, 1995.

[22] H. L. Bodlaender and T. Kloks: *Efficient and constructive algorithms for the pathwidth and treewdith of graphs.* J. Algorithms 21, pp 358-402, 1996.

[23] H. L. Bodlaender, T. Kloks, and D. Kratsch: *Treewidth and pathwidth of permutation graphs.* Proceedings of the 20th International Colloquium on Automata, Langauges and Programming, A. Lingas, R. Karlsson, and S. Carlsson (Eds.), Vol 700 LNCS, Springer-Verlag, pp 114-125, 1993.

[24] H. L. Bodlaender and R. H. Möhring: *The pathwidth and treewdith of cographs.* SIAM J. Disc. Meth. 6, pp 181-188, 1993.

[25] V. Bouchitte and M. Habib: *NP-completeness properties about linear extensions.* Order 4, no 2, pp 143-154, 1987.

[26] G. Brightwell: *Graphs and Partial Orders.* London School of Economics, Mathematics Preprint Series, 1994.

[27] G. Brightwell and P. Winkler: *Counting linear extensions is #P-complete.* Proceedings of the 23rd ACM Symposium on Theory of Computing, pp 175-181, 1991.

[28] G. Brightwell and P. Winkler: *Counting linear extensions.* Order, Vol 8, pp 25-242, 1992.

[29] S. Buss: private communication, 1989.

[30] Liming Cai, J. Chen, R. G. Downey, and M. R. Fellows *The parametereized complexity of short computation and factorization.* Archive for Math Logic 36, pp 321-337, 1997.

[31] L. Cai, J. Chen, R. G. Downey and M. R. Fellows *Advice Classes of Parameterized Tractability.* Annals of Pure and Applied Logic 84, pp 119-138, 1997.

[32] K. Cattell, M. J. Dinneen, R. G. Downey, M. R. Fellows and M. A. Langston: *On Computing Graph Minor Obstruction Sets.* Theoretical Computer Science A 233 (1-2), pp107-127, 2000.

[33] M. Cesati: *Perfect Code is $W[1]$-complete.* Information Processing Letters Vol 81(3), pp 163-168, 2002.

[34] M. Cesati: *The Turing Way to the Parameterized Intractability.* manuscript.

[35] M. Cesati and L. Trevisan: *On the Efficiency of Polynomial Time Approximation Schemes.* Information Processing Letters 64 (4), pp 165-171, 1997.

[36] N. Chandrasekharan and S. T. Hedetniemi: *Fast parallel algorithms for tree decomposing and parsing partial k-trees.* Proceedings of the 26th Annual Allerton Conference on Communication, Control, and Computing, pp 283-292, 1988.

[37] J. Chen, I.A. Kanj and W. Jia: *Vertex Cover: Further Observations and Further Improvements.* Journal of Algorithms 41, pp 280-301, 2001.

[38] M. Chrobak and M. Slusarek: *On some packing problems related to dynamic storage allocation.* RAIRO Inform. Theor. Appl. 22, pp 487-499, 1988.

[39] C. J. Colbourn and W. R. Pulleyblank: *Minimizing setups in ordered sets of fixed width.* Order 1, pp 225-229, 1985.

[40] S. Cook: *The complexity of theorem proving procedures.* Proceedings of the 3rd ACM STOC, pp 151-158, 1971.

[41] B. Courcelle: *The monadic second-order logic of graphs I: Recognizable sets of finite graphs.* Information and Computation 85, pp 12-75, 1990.

[42] B. Courcelle, J.A. Makowsky and U. Rotics: *On the Fixed Parameter Complexity of Graph Enumeration Problems Definable in Monadic Second Order Logic.* Discrete Applied mathematics, Vol. 108, No. 1-2, pp 23-52, 2001.

[43] R. P. Dilworth: *A decomposition theorem for partially ordered sets.* Ann. of Math. 51, pp 161-166, 1950.

[44] M. J. Dinneen: *Bounded Combinatorial Width and Forbidden Substructures.* PhD thesis, University of Victoria, Canada, 1995.

[45] Guoli Ding and Bogdan Oporowski: *Some results on tree decompositions of graphs.* J. Graph Theory 20, pp 481-499, 1995.

[46] R. G. Downey and M. R. Fellows: *Fixed-parameter tractability and completeness.* Congressus Numeratium, Vol 87, pp 161-187, 1992.

[47] R. Downey and M. Fellows *Fixed-parameter Tractability and Completeness I: Basic Theory.* SIAM Journal of Computing 24, pp 873-921, 1995.

[48] R. G. Downey and M. R. Fellows *Parameterized Complexity* Springer-Verlag, 1999.

[49] D. Duffus, I. Rival and P. Winkler: *Minimizing setups for cycle-free ordered sets.* Proc. Amer. Math. Soc. 85, pp 509-513, 1982.

[50] M. H. El-Zahar and J. H. Schmerl: *On the size of jump-critical ordered sets* Order 1, pp 3-5, 1984.

[51] M. R. Fellows and M. A. Langston: *An analogue of the Myhill-Nerode theorem and its use in computing finite-basis characterizations.* Proceedings of the 30th Annual Symposium on Foundations of Computer Science, IEEE Computer Science Press, Los Alamitos, California, pp 520-525, 1989.

[52] Michael R. Fellows and Catherine McCartin: *On the parametric complexity of schedules to minimize tardy tasks.* Theoretical Computer Science 298, pp 317-324, 2003.

[53] Babette de Fluiter: *Algorithms for Graphs of Small Treewidth.* ISBN 90-393-1528-0.

[54] Markus Frick: *Easy Instances for Model-Checking.* Ph.D. Thesis, June 2001, Laboratory for Foundations of Computer Science, The University of Edinburgh, 2001.

[55] Markus Frick and Martin Grohe: *Deciding First-Order Properties of Locally Tree-Decomposable Graphs.* Proceedings of the 26th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 1644, pp 331-340, Springer-Verlag, 1999.

[56] J. Fouhy: *Computational Experiments on Graph Width Metrics.* MSc Thesis, Victoria University, Wellington, 2003.

[57] M. R. Garey and D. S. Johnson *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, New York, 1979.

[58] M. Grohe: personal communication.

[59] M. Grohe: *Computing Crossing Numbers in Quadratic Time.* Proceedings of the 32nd ACM Symposium on Theory of Computing, pp 231-236, 2001.

[60] A. Gyarfas and J. Lehel: *On-line and First Fit Coloring of Graphs.* J. Graph Theory, Vol. 12, No. 2, pp 217-227, 1988.

[61] J. E. Hopcroft and R. M. Karp: *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs* SIAM J. Computing 2, pp 225-231, 1973.

[62] S. Irani: *Coloring inductive graphs on-line.* Proceedings of the 31st Annual Symposium on Foundations of Computer Science, Vol 2, pp 470-479, 1990.

[63] H. A. Kierstead: *Recursive and On-Line Graph Coloring* In Handbook of Recursive Mathematics, Volume 2, pp 1233-1269, Studies in Logic and the Foundations of Mathematics, Volume 139, Elsevier, 1998.

[64] H. A. Kierstead: *The Linearity of First Fit Coloring of Interval Graphs.* SIAM J. on Discrete MAth, Vol 1, No. 4, pp 526-530, 1988.

[65] H. A. Kierstead and J. Qin: *Coloring interval graphs with First-Fit.* (Special issue: Combinatorics of Ordered Sets, papers from the 4th Oberwolfach Conf., 1991), M. Aigner and R. Wille (eds.), Discrete Math. 144, pp 47-57, 1995.

[66] H. A. Kierstead and W. A. Trotter: *An Extremal Problem in Recursive Combinatorics.* Congressus Numeratium 33, pp 143-153, 1981.

[67] N. G. Kinnersley and M. A. Langston: *Obstruction set isolation for the gate matrix problem.* Tech. Rep. CS-91-126, Computer Science Department, University of Tennessee, Knoxville, USA, 1991.

[68] T. Kloks: *Treewidth. Computations and Approximations.* Lecture Notes in Computer Science, Vol 842, Springer-Verlag, 1994.

[69] D. Kozen: *The Design and Analysis of Algorithms.* Springer-Verlag, 1991.

[70] J. Lagergren: *Algorithms and minimal forbiddden minors for tree decomposable graphs.* Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden, 1991.

[71] J. Lagergren: *Efficient parallel algorithms for graphs of bounded treewidth.* J. Algorithms 20, pp 20044, 1996.

[72] J. Lagergren and S. Arnborg: *Finding minimal forbidden minors using a finite congruence.* Proceedings of the 18th International Colloquiumon Automata, Languages and Programming, Vol 510 LNCS, Springer-Verlag, pp 532-543, 1991.

[73] L. Lovasz, M. E. Saks, and W. A. Trotter: *An On-Line Graph Coloring Algorithm with Sublinear Performance Ratio.* Bellcore Technical Memorandum, No. TM-ARH-013-014.

[74] J.A. Makowsky: *Colored Tutte Polynomials and Kauffman Brackets for Graphs of Bounded Tree Width.* Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, Washington DC, pp 487-495, 2001.

[75] M. S. Manasse, L. A. McGeoch, and D. D. Sleator: *Competitive Algorithms for Online Problems.* Proceedings of the 20th Annual ACM Symposium on Theory of Computing, pp 322-333, 1988.

[76] J. Matousek and R. Thomas: *Algorithms for finding tree-decompositions of graphs.* J. Algorithms 12, pp 1-22, 1991.

[77] Catherine McCartin: *An improved algorithm for the jump number problem.* Information Processing Letters 79, pp 87-92, 2001.

[78] Catherine McCartin: *Parameterized counting problems.* Proceedings of 27th International Symposium on Mathematical Foundations of Computer Science (MFCS 2002), pp 556-567, 2002.

[79] G. L. Nemhauser and L. E. Trotter Jr: *Vertex packings: Structural properties and algorithms.* Mathematical Programming 8, pp 232-248, 1975.

[80] R. Niedermeier and P. Rossmanith: *A general method to speed up fixed-parameter tractable algorithms.* Information Processing Letters 73, pp 125-129, 2000.

[81] R. Niedermeier: *Invitation to fixed-parameter algorithms.* manuscript, 2002.

[82] S.D. Noble: *Evaluating the Tutte Polynomial for graphs of bounded tree-width.* Combin. Probab. Comput. 7, pp 307-321, 1998.

[83] C. H. Papdimitriou: *Computational Complexity.* Addison Wesley, 1994.

[84] L. Perkovic and B. Reed: *An Improved Algorithm for Finding Tree Decompositions of Small Width.* International Journal of Foundations of Computer Science 11 (3), pp 365-371, 2000.

[85] W. R. Pulleyblank: *On minimizing setups in precedence constrained scheduling.* Discrete Applied Math., to appear.

[86] V. Raman: presentation at Dagstuhl Seminar on Parameterized Complexity, Dagstuhl, Germany, July 29-August 3, 2001.

[87] A. Razborov and M. Alekhnovich: *Resolution is Not Automatizable Unless W[P] is Tractable.* Proc. of the 42nd IEEE FOCS, pp 210-219, 2001.

[88] B. Reed: *Finding approximate separators and computing treewdith quickly.* Proceedings of the 24th Annual Symposium on Theory of Computing, ACM Press, New York, pp 221-228, 1992.

[89] I. Rival: *Optimal linear extensions by interchanging chains.* Proc. Amer. Math. Soc. 89, pp 387-394, 1983.

[90] N. Robertson and P. D. Seymour: *Graph minors I. Excluding a forest.* J. Comb. Theory Series B 35, pp 39-61, 1983.

[91] N. Robertson and P. D. Seymour: *Graph minors III. Planar tree-width.* J. Comb. Theory Series B 36, pp 49-64, 1984.

[92] N. Robertson and P. D. Seymour: *Graph minors - a survey.* Surveys in Combinatorics, I. Anderson (Ed.), Cambridge Univ. Press, pp 153-171, 1985.

[93] N. Robertson and P. D. Seymour: *Graph minors II. Algorithmic aspects of tree-width.* Journal of Algorithms 7, pp 309-322, 1986.

[94] N. Robertson and P. D. Seymour: *Graph minors V. Excluding a planar graph.* J. Comb. Theory Series B 41, pp 92-114, 1986.

[95] N. Robertson and P. D. Seymour: *Graph minors VI. Disjoint paths across a disc.* J. Comb. Theory Series B 41, pp 115-138, 1986.

[96] N. Robertson and P. D. Seymour: *Graph minors VII. Disjoint paths on a surface.* J. Comb. Theory Series B 45, pp 212-254, 1988.

[97] N. Robertson and P. D. Seymour: *Graph minors IV. Treewidth and well-quasi-ordering.* J. Comb. Theory Series B 48, pp 227-254, 1990.

[98] N. Robertson and P. D. Seymour: *Graph minors VIII. A Kuratowski theorem for general surfaces.* J. Comb. Theory Series B 48, pp 255-288, 1990.

[99] N. Robertson and P. D. Seymour: *Graph minors IX. Disjoint crossed paths.* J. Comb. Theory Series B 49, pp 40-77, 1990.

[100] N. Robertson and P. D. Seymour: *Graph minors X. Obstructions to tree-decomposition.* J. Comb. Theory Series B 52, pp 153-190, 1991.

[101] N. Robertson and P. D. Seymour: *Graph minors XI. Distance on a surface.* J. Comb. Theory Series B 60, pp 72-106, 1994.

[102] N. Robertson and P. D. Seymour: *Graph minors XII. Excluding a non-planar graph.* J. Comb. Theory Series B 64, pp 240-272, 1995.

[103] N. Robertson and P. D. Seymour: *Graph minors XIII. The disjoint paths problem.* J. Comb. Theory Series B 63, pp 65-110, 1995.

[104] N. Robertson and P. D. Seymour: *Graph minors XIV. Extending an embedding.* J. Comb. Theory Series B 65, pp 23-50, 1995.

[105] N. Robertson and P. D. Seymour: *Graph minors XV. Giant steps.* J. Comb. Theory Series B 68, pp 112-148, 1996.

[106] D. Seese: *The structure of models of decidable monadic theories of graphs.* Ann. Pure and Appl. Logic, Vol 53, pp 169-195, 1991.

[107] D. D. Sleator and R. E. Tarjan: *Amortized Efficiency of List Update and Paging Rules.* Comunication of the ACM 28, pp 202-208, 1985.

[108] M. M. Syslo: *Minimizing the jump number for partially-ordered sets: A graph-theoretic approach, II.* Discrete Mathematics 63, pp 279-295, 1987.

[109] M. M. Syslo: *An algorithm for solving the jump number problem.* Discrete Mathematics 72, pp 337-346, 1988.

[110] M. M. Syslo: *On some new types of greedy chains and greedy linear extensions of partially ordered sets.* Discrete Applied Math. 60, pp 349-358, 1995.

[111] M. Szegedy: private communication, reported in [63].

[112] L. Valiant: *The complexity of computing the permanent.* Theoret. Comput. Sci., Vol 8, pp 189-201, 1979.

[113] D. Welsh and A. Gale: *The Complexity of Counting Problems.* Aspects of Complexity, editors R. Downey and D.Hirschfeldt, de Gruyter Series in Logic and Its Applications, pp 115-154, 2001.

Index