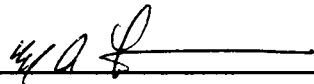


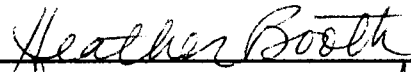
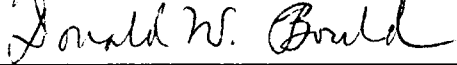
To the Graduate Council:

I am submitting herewith a dissertation written by Rajeev Govindan entitled "Algorithmic Methods for Circuit Layout and Partitioning." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.



Michael. A. Langston, Major Professor

We have read this dissertation
and recommend its acceptance:


Accepted for the Council:



Associate Vice Chancellor and
Dean of the Graduate School

**Algorithmic Methods
for
Circuit Layout and Partitioning**

A Dissertation
Presented for the
Doctor of Philosophy Degree
The University of Tennessee, Knoxville

Rajeev Govindan

May 1998

Acknowledgements

I thank my research advisor, Prof. Michael Langston, for his guidance. Over the years, I have learned much from him; not just about well-quasi-orders and non-constructive algorithms, but also about the evils of split infinitives and run-on sentences. I am grateful to Profs. Booth, Boudin and Raghavan for serving on my graduate committee and for agreeing to read my dissertation.

I thank my family, especially my parents, who were a steady source of support for me even when they had troubles of their own; Sidd, for good advice and the occasional game of tennis; Anil and Babu, for interesting discussions and games of chess. Finally, I thank Susan for being a good friend and partner during some tough times.

Abstract

Automating the physical design of VLSI circuits is a very hard problem that has increased in significance as circuits have grown in size and complexity. Several variations of the general problem exist, depending mainly on the underlying technology. Many attempts to automate circuit layout begin with modeling circuits as graphs or hypergraphs. Laying out the circuit so that some objective function is optimized then corresponds to laying out its graph so that some graph metric is optimized.

We begin by considering pathwidth. This metric is directly related to the area of circuits laid out in Gate Matrix and Weinberger array styles. It is $\mathcal{NP}$ -hard to determine pathwidth, even on severely restricted classes of graphs. We present algorithms to approximate the pathwidth of several related classes of graphs, such as k -chordal, outerplanar, series-parallel and Halin graphs. Our algorithms work in $O(n \log n)$ time on graphs of order n , and produce solutions within small constant factors of the optimum.

Next we focus on a different aspect of the circuit layout problem, one that arises during design using multiple field-programmable gate arrays (FPGAs). A critical step during multiple-FPGA design is the partitioning of a circuit into pieces that satisfy size and pin-count constraints. We examine variants of partitioning with these constraints, and develop complexity results and algorithms for the general and fixed-parameter versions of each problem.

We conclude on a more theoretical note, designing some foundational graph tests that can be used in a circuit layout library. We develop an algorithm to decide whether K_4 , the complete graph on four vertices, is immersed in an arbitrary input graph. We also devise an algorithm to decide whether the k -bond, the graph consisting of two vertices joined by k edges, is immersed in a series-parallel graph. Both algorithms work in linear time. Finally, we present a fast algorithm that may be useful as a preprocessing step in immersion tests.

Contents

1	Introduction and Background	1
1.1	Notation	5
1.2	Data Structures	6
2	Layout Approximation	8
2.1	Prior Results	9
2.2	Our Work	11
2.3	The Pathwidth of AC Graphs	11
2.3.1	The Reduction	12
2.4	Tree Decompositions	17
2.5	Chordal Graphs	19
2.6	Outerplanar Graphs	22
2.6.1	An Algorithm for Biconnected Graphs	22
2.6.2	Efficiency	25
2.6.3	Tackling Non-Biconnected Graphs	27
2.7	Series-Parallel Graphs	29
2.7.1	Two-Terminal Series-Parallel Graphs	29
2.7.2	Procedure Rearrange	31
2.7.3	Constructing Tree Decompositions	34
2.7.4	Full Input	37
2.7.5	Non-Full Input	39

2.7.6	Tackling Non-Biconnected Graphs	43
2.8	Graph Decompositions	45
2.9	Halin Graphs	47
2.9.1	Outerplanar Decompositions	48
2.9.2	Correctness and Complexity	51
2.9.3	Performance Bounds	53
2.10	Experimental Results	55
2.11	Discussion	57
3	The Complexity of Partitioning	59
3.1	The Immersion Order	60
3.2	A Fundamental Problem	61
3.3	Adding One More Parameter	64
3.3.1	Complexity	65
3.3.2	Hard Instances	68
3.4	Related Cutting Problems	70
3.5	Refining the Complexity	76
3.6	Obstruction Sets	81
3.7	Approximations	83
3.8	Discussion	83
4	Toward a Library of Tests	85
4.1	A K_4 Test	87
4.1.1	Three-Edge Connectivity	87
4.1.2	The Decision Algorithm	89
4.1.3	The Correctness of K_4 -test	90
4.1.4	Finding a Model	101
4.1.5	Experimental Results	106
4.1.6	Parallelization	106

4.2	A k -bond Test	108
4.2.1	Correctness and Complexity	109
4.3	Finding k -edge-connected Sets	113
4.3.1	Correctness and Complexity	115
4.4	Discussion	118
5	Conclusions	119
5.1	Summary of Results	119
5.2	Open Problems	120
	Bibliography	122
	Appendix: Glossary	132
	Vita	139

List of Figures

1.1	The structure of a node in an adjacency list.	6
1.2	A graph and its representation.	7
2.1	The graph G_v that replaces a vertex v	13
2.2	T and G_T , when T has order two.	13
2.3	T and G_T , when T has order three.	14
2.4	A path and its w -extensible tree decomposition.	23
2.5	A difficult series-parallel graph.	29
2.6	A 2TSP graph and a series-parallel decomposition tree for it.	30
2.7	The transformation performed by rearrange-top	31
2.8	A 2TSP graph G	34
2.9	A series-parallel decomposition tree D for G	35
2.10	D after rearrangement and coloring.	35
2.11	A tree decomposition of G	35
2.12	K_4^- and $(K_4^-)^2$	40
2.13	A Halin graph.	48
2.14	Halin graph after preprocessing.	49
2.15	Outerplanar graph obtained from Halin graph.	49
2.16	Outerplanar decomposition of Halin graph.	50
2.17	Halin decomposition of outerplanar graph.	54
3.1	Examples of mine graphs.	69

3.2	A grid expansion.	79
4.1	Edge-disjoint paths in a three-edge-connected graph.	92
4.2	The graph M	92
4.3	Graphs used in the proof of Lemma 4.6.	93
4.4	Graphs used in the proof of Lemma 4.7.	94
4.5	Graphs used in the proof of Lemma 4.8.	95
4.6	Models of K_4 that span a cut-point.	96
4.7	A model of K_4 with corners u, v, x and y	97
4.8	Graphs used in the proofs of Lemmas 4.12 and 4.14.	98
4.9	Graphs used in the proof of Lemma 4.15.	100
4.10	Paths to be found if $G - \{v\}$ has three or more connected components.	104
4.11	Paths to be found if v has at least two neighbors in u 's component.	104
4.12	Paths to be found if v has no neighbor besides u in u 's component.	104
4.13	Paths to be found if the K_4 model lies in a biconnected component.	104
4.14	Possible k -bond models in G	110
4.15	Possible k -bond models in G'	112
4.16	Edge-disjoint paths between u and v	114

List of Tables

2.1	Experimental Results on Series-Parallel Graphs	56
2.2	Experimental Results on Outerplanar Graphs	56
3.1	Complexity of FPGA Minimization	66
4.1	Running Times of K_4 Testing and Model-Finding Algorithms	107

Chapter 1

Introduction and Background

Digital circuits have grown tremendously in size and complexity over the last two decades. As a result, it has become increasingly necessary to develop efficient algorithms to automate the various aspects of circuit design. A long series of complex steps, each of which can benefit from automation, is required before a specification of a circuit can be translated into actual hardware. The interested reader is referred to [DeM] for an overview of this process and of the problems encountered at each stage.

Our work is relevant to one of these steps: that of laying out a circuit to satisfy certain specified criteria. For this step, the input is a model of a circuit. Often, the model is a graph or hypergraph, with information associated with each node and edge (or hyperedge). The constraints that determine the validity of a layout and the objective functions that determine its quality depend on the underlying technology, economic factors and the domain of application of the circuit, among other things. A whole slew of graph-theoretic problems results when attempting to deal with these variations. See [Len] for a fairly comprehensive survey of this field.

In the first chapter of this dissertation, we examine the pathwidth metric of graphs. Pathwidth was first defined and investigated for reasons totally unrelated to circuit design [RS2]. Later, it was discovered that this metric models, provided some simplifying assumptions are made, the problem of minimizing the area of circuits laid out

in the Gate Matrix or Weinberger Array styles [FL5].

Pathwidth has application in other areas as well, and has been studied by several researchers [EST, KoT, KF]. It is known to be $\mathcal{NP}$ -hard, even on structurally restricted classes of graphs [MoS, Gus]. Nevertheless, polynomial-time algorithms exist for some classes of graphs; for instance, a linear-time algorithm is known for trees [EST]. Exactly how rich a class of graphs is required before the problem becomes $\mathcal{NP}$ -hard is an open question. We provide a partial answer by demonstrating that Pathwidth is $\mathcal{NP}$ -hard on AC graphs, which are only slightly more complex than trees. Approximating pathwidth in the absolute sense¹ is equally hard, and even relative approximation algorithms² have proved elusive. Proposed heuristics have no performance guarantees, and can be shown to fail badly on certain inputs [DKL]. Thus it is natural to try to devise approximations that can provide guarantees about the quality of the solutions produced.

We present algorithms to approximate the pathwidth of k -chordal, outerplanar and series-parallel graphs to within small constant factors. We then generalize the technique upon which these algorithms are built, to obtain a relative approximation algorithm for the pathwidth of Halin graphs. We also sketch some other possible applications of the generalized technique.

In the second chapter, we examine problems that arise during a comparatively new style of layout that uses multiple FPGAs. A critical step during this style of design is the partitioning of a circuit into pieces that satisfy specified constraints, while attempting to optimize some objective function. Although much work has been done on circuit partitioning in general, the new technology demands different constraints than have been considered in the literature. The most radical difference is the presence of pin-count (degree) constraints.

¹An absolute approximation algorithm produces a solution within a constant additive factor of the optimum.

²A relative approximation algorithm produces a solution within a constant multiplicative factor of the optimum.

We analyze several variants of the partitioning problem with degree constraints, and obtain complexity results for the general and fixed-parameter versions of each problem. These problems, with one exception, are shown to be $\mathcal{NP}$ -hard in full generality. For the lone “easy” problem, we are able to develop a polynomial-time algorithm, albeit one with quintic complexity. We also prove that one of these problems, of particular practical relevance, is $\mathcal{NP}$ -hard on bounded-degree input. This result is significant, because graphs that represent circuits typically have bounded degree.

Our fixed-parameter problems are closed in the immersion³ order. This is interesting, because an elegant and conceptually simple approach is known to tackle problems that are closed in this order. Such problems can be characterized in terms of a finite number of forbidden substructures called *obstructions*. A polynomial-time algorithm exists to test for the presence of an arbitrary obstruction [FL5]. Thus, in principle, one may verify that a given circuit has a satisfactory partition by checking that it does not contain any of the obstructions. Furthermore, for the problems we consider, efficient self-reductions may be used to construct satisfactory partitions once it has been determined that they exist.

With this approach in mind, we perform an exploration of the sets of obstructions for our immersion-closed partition problems. Unfortunately, we find that the cardinality of the obstruction sets increases exponentially with increasing values of input parameters. This combinatorial explosion suggests that checking for the presence of every obstruction will be impractical. It may be possible to circumvent this difficulty to some degree, by only checking for a carefully selected subset of the entire set of obstructions, thus sacrificing accuracy for speed. Even then, fast tests for critical obstructions must be developed, since the generalized immersion test of [FL5] makes use of the impractical vertex-disjoint paths algorithm of [RS13].

³ $H \leq G$ in the immersion order if H can be obtained from G by a series of subgraph and lifting operations.

In the third chapter, we focus on designing efficient immersion tests for two fundamental graphs. First, we develop a linear-time algorithm to check whether K_4 , the complete graph on four vertices, is immersed in an input graph. This algorithm also returns a minimal subgraph containing a K_4 , if any exist. Ours is the first practical algorithm to perform this test.

Next, we present a fast test for k -bonds, graphs of order two whose vertices are adjacent by k edges. Our algorithm works only on series-parallel graphs. It is easy to see that for any k , k -bonds can be detected by a quadratic number of calls to a network flow subroutine. With sufficient care, it is in fact possible to accomplish this with only a linear number of calls. The complexity of the latter algorithm is quartic in general, and quadratic on series-parallel graphs. The algorithm that we devise is simpler, and has linear complexity.

The immersion of graph H in graph G can also be viewed as a mapping that takes vertices of H to vertices of G and edges of H to edge-disjoint paths in G . This characterization suggests that edge-connectivity, which measures the number of edge-disjoint paths between vertices, is relevant to immersion-testing. In fact, our algorithm for K_4 uses a simple and fast three-edge-connectivity routine to preprocess the input graph [BGLR]. Moreover, it can be shown that if H has edge-connectivity k , then vertices in G that are potential images of vertices in H must themselves have edge-connectivity k or more. Motivated by this, we design an algorithm to find sets of mutually k -edge-connected vertices in an input graph.

As with k -bonds, k -edge-connected vertex sets can be identified using a linear number of calls to a network flow algorithm. Thus the k -edge-connected sets in a graph of order n can be found in $O(n^4)$ time. If the graph is series-parallel, then the complexity drops to $O(n^2)$. The algorithm we present uses a completely different approach, and is restricted to series-parallel graphs. It takes $O(kn)$ time, and should outperform algorithms based on network flow for small k .

1.1 Notation

In this section, we define terms and symbols that are used recurrently in this dissertation. Terms whose usage is limited to specific chapters are, for the most part, defined just before they are used.

The notation we use pertaining to graphs is standard, and conforms to the usage in [Har]. We consider only finite undirected graphs. These may contain multiple edges, unless stated otherwise, but no loops, since loops are irrelevant to all the problems we consider. We also assume that graphs are connected, as this permits a cleaner presentation of results. All of our algorithms are applicable to disconnected graphs; one merely needs to run them on each connected component in turn.

Given a graph G , $V(G)$ denotes the vertex set of G and $E(G)$ denotes the edge set of G . The *order* of G is the number of vertices in it. The *size* of G is the number of edges in it. We typically use the variables n and m to denote order and size, respectively. A *cut point* of G is a vertex whose removal disconnects G . A *biconnected component* of G is a maximal subgraph with no cut points.

If v is a vertex in G , the *degree* of v (denoted $\delta_G(v)$) is the number of edges in G that are incident on v . As long as there is no ambiguity about the graph under discussion, we drop the subscript and use $\delta(v)$ to denote the degree of v . We employ the same notation for the degree of vertex sets: for $S \subseteq V(G)$, $\delta_G(S)$ is the number of edges in G that have exactly one endpoint in S .

In simple graphs, edges can be uniquely identified by their endpoints. When the graph under discussion is simple, we will typically use this method of specifying an edge. For instance, if e is an edge with endpoints a and b , we will use ab to denote e . We introduce the notation $[ab]$ to denote a simple path between a and b . A path $[ab]$ may consist of a single edge, or may contain several edges.

1.2 Data Structures

This section describes the data structures used by our algorithms.

All classes of graphs to which our algorithms apply have a linear number of edges. Therefore, it is not space-efficient to store input graphs in the conventional adjacency matrix format. Instead we use a doubly-linked adjacency list format, which enables us to save space and moreover, permits efficient implementation of certain operations.

Figure 1.1 depicts the structure of each node in the adjacency list of a vertex. There are fields to store the vertex number and the number of copies of the multiedge corresponding to the node. There are pointers to the previous and next nodes in the list, and a pointer to the “counterpart” of this node (if vertices a and b are adjacent, then the node in a ’s adjacency list whose vertex label field is b and the node in b ’s adjacency list whose vertex label field is a are counterparts). Figure 1.2 shows a graph and its representation. Null pointers have been omitted.

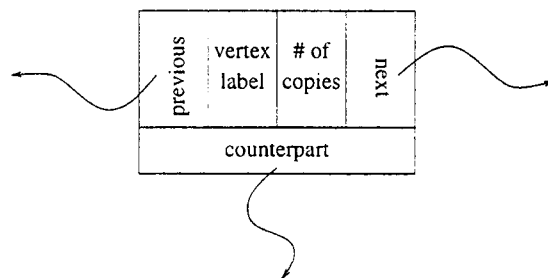


Figure 1.1: The structure of a node in an adjacency list.

Many of our algorithms work by successive elimination of vertices with one or two neighbors. The adjacency list format permits a constant-time implementation of this operation. To illustrate, let v with neighbors u and w , denote a vertex with exactly two neighbors. Then the edges incident on v can be eliminated by scanning the two nodes in the adjacency list of v . The counterpart pointer enables us to delete nodes corresponding to these edges from the adjacency lists of u and w , without actually scanning those lists.

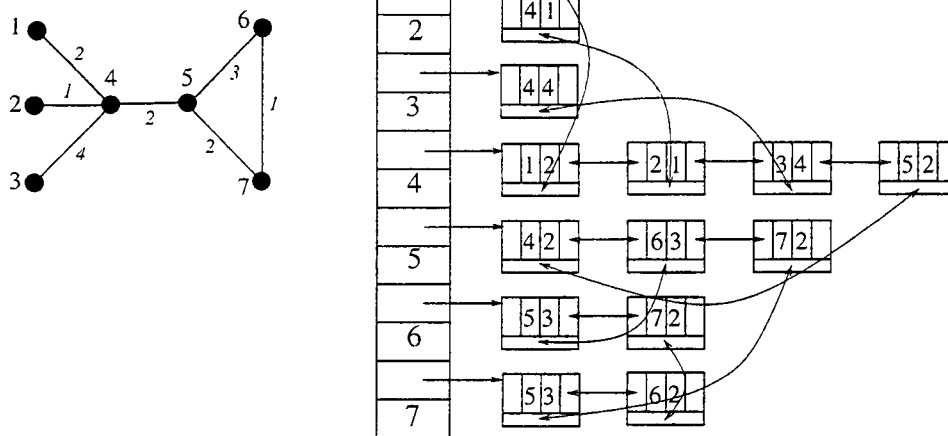


Figure 1.2: A graph and its representation.

Our algorithms also use stacks and queues, which can be efficiently implemented by means of linked lists.

Chapter 2

Layout Approximation

In this chapter we consider the problem of determining the pathwidth of a graph. Pathwidth was originally defined by Robertson and Seymour in one of their seminal papers on Graph Minors [RS2]. Since then, this graph width metric has found application in areas as disparate as circuit layout [FL5] and natural language processing [KoT].

A *path decomposition* of a graph G is a sequence $X_1, \dots, X_r$ of subsets of $V(G)$ such that the following two properties hold.

- Edge covering: for each edge $e \in E(G)$, some X_i contains both endpoints of e .
- Intersection: for $1 \leq i \leq j \leq k \leq r$, $X_i \cap X_k \subseteq X_j$.

The *width* of a path decomposition $X_1, \dots, X_r$ is one less than the size of the largest set X_i , $1 \leq i \leq r$. The *pathwidth* of G (denoted $pw(G)$) is the smallest width of all its path decompositions.

The close connection between pathwidth and a variety of VLSI layout optimization problems, such as Gate Matrix Layout and folding of programmable logic arrays and Weinberger arrays, was first pointed out in [FL5]. There it is shown, for instance, that the number of tracks required to lay a circuit out in Gate Matrix Layout style is exactly one more than the pathwidth of the circuit's intersection graph (this graph is

formed by mapping nets in the circuit to vertices and adding edges between vertices if the corresponding nets are incident on the same gate). Thus one can calculate the number of tracks required for a layout by constructing an intersection graph and determining its pathwidth. Moreover, an optimal path decomposition of the graph provides an optimal layout for the circuit. A detailed survey of pathwidth applications in VLSI can be found in [Möh].

The pathwidth metric is also closely related to several well-known graph problems. Consider, for example, the vertex separation problem [EST], whose formulation was motivated by the task of finding good separators in graphs. The pathwidth of a graph is exactly one less than its vertex separation [Kin1]. Searching a graph is a classical problem, where one attempts to minimize the number of searchers it takes to capture a fugitive who is free to move along the edges. Pathwidth and search number differ by at most two [EST]. The Node Searching problem places more limits on searchers, but is otherwise similar to the Searching problem. The node search number of a graph is one more than its pathwidth [EST]. The interval graph augmentation problem is that of finding an interval graph of minimum clique number that contains a given graph as a subgraph. This turns out to be equivalent to the problem of constructing an optimal path decomposition [KP]. Cutwidth is a metric defined in terms of cuts in a linear layout of a graph. Cutwidth has applications in several areas, including circuit layout. An algorithm to determine pathwidth can be used, along with a simple transformation, to compute the cutwidth of any graph of maximum degree three [MoS].

2.1 Prior Results

It is $\mathcal{NP}$ -hard to determine pathwidth [KF], and even to approximate it to within an additive constant of the optimum [DKL]. Restricting the input does not simplify the problem significantly; for instance, the pathwidth problem remains $\mathcal{NP}$ -hard on

planar graphs of maximum degree three [MoS] and on chordal graphs [Gus].

In contrast, the fixed-parameter version of the pathwidth problem is tractable, at least in principle. Let n denote the order of a graph and let k denote a constant. Dynamic programming can be used to construct a path decomposition of width k or less if any exist, in $O(n^{2k^2+4k+2})$ time [EST]. Methods based on well-quasi-order theory can be used to show that it is decidable in quadratic time whether a graph has pathwidth at most k [FL6]. Recent improvements have even led to a constructive linear-time algorithm [Bod3, BK]. Nevertheless, the low asymptotic complexity of these algorithms hides huge constants of proportionality that render them impractical for all but very small k . There has been limited success with ad hoc approaches for some values of k . Practical linear-time algorithms exist for $k \leq 3$ [KK], but these methods are not applicable when $k > 3$. A fast “almost always correct” algorithm is known that decides if a graph has pathwidth at most three [GLR], but this approach is also unlikely to be viable for much larger values of k . No practical algorithm is known to determine if a graph has pathwidth four.

There are few classes of graphs for which efficient pathwidth algorithms exist. Trees are a rare exception; the pathwidth of trees can be determined in linear time [EST]. It is also known that the pathwidth of graphs of bounded treewidth (trees have treewidth one) can be determined in polynomial time [BK]. Though this result is of theoretical significance, the algorithm is impractical even on graphs of treewidth two.

Considering the intractability of the pathwidth problem, it is natural to search for approximation algorithms. Although many heuristics have been published ([HPK, Li, WHW]), none can guarantee the quality of its solution. These heuristics tend to be variations on a greedy approach, and have been shown to produce arbitrarily bad solutions on some input instances [DKL]. An algorithm is known that, given a graph of order n , produces a path decomposition of width at most $\log^2 n$ times the optimum [BGHK].

2.2 Our Work

We show that Pathwidth is $\mathcal{NP}$ -hard on graphs with acyclic clique-graphs, also known as AC graphs. AC graphs form a subclass of chordal graphs [BR]. Thus our proof is an improvement over the result of [Gus]. We then devise relative approximation algorithms for certain classes of graphs. Our algorithms approximate pathwidth to within small constant factors in $O(n \log n)$ time, where n denotes the order of the input graph. We discuss k -chordal, outerplanar, series-parallel and Halin graphs in detail. Since all these classes of graphs have bounded treewidth, the methods in [BK] can, in principle, be used to determine their pathwidth exactly in polynomial time. This is not a practical option, however, because of the high degree of the polynomial and its enormous multiplicative constant.

We begin by presenting our proof of $\mathcal{NP}$ -hardness for AC graphs. The reduction used in this proof requires the consideration of graphs with multiple edges. But from Section 2.4 to the end of this chapter, we restrict attention to simple graphs, because multiple copies of edges are irrelevant to the pathwidth metric. Our approximation algorithms are based on the fact that tree decompositions with certain properties can be converted into good path decompositions. This reduces the problem of approximating pathwidth to that of constructing satisfactory tree decompositions, and is the basis of our algorithms for k -chordal, outerplanar and series-parallel graphs. For Halin graphs, we generalize the notion of tree decomposition, and base our approximation on this generalization.

2.3 The Pathwidth of AC Graphs

The *clique graph* of G is a graph whose vertices correspond to the maximal cliques of G , and in which an edge exists between a pair of vertices if and only if the corresponding cliques in G have one or more vertices in common. G is said to be an *AC graph* if its clique graph is acyclic. AC graphs have the following alternate characterization:

A graph is an AC graph if and only if it is chordal and every vertex in it belongs to at most two maximal cliques [BR]. (A graph is said to be *chordal* if it contains no chordless cycles of length four or more.)

A *linear layout* of G is a one-to-one mapping from $V(G)$ to $\{1, \dots, |V(G)|\}$. This mapping may be interpreted as describing how the vertices of G are to be laid out in linear fashion. The *cutwidth* of a linear layout of G is the maximum number of edges from $E(G)$ that are cut by a line drawn between consecutive vertices. The *cutwidth* of G (denoted $cw(G)$) is the smallest cutwidth of all the linear layouts of G . An instance of the Cutwidth problem consists of a graph G and an integer k . The question to be answered is whether G has cutwidth k or less.

2.3.1 The Reduction

The reduction is from Cutwidth, restricted to trees with multiedges. In the rest of this section, T denotes a tree with multiedges; that is, T is a graph such that we obtain a tree by replacing each multiedge in T by a single edge. We assume that the number of edges between a pair of vertices in T is bounded by a polynomial in $|V(T)|$. Cutwidth is $\mathcal{NP}$ -hard on such graphs [MoS].

Given T , we construct an AC graph G_T as follows: Let s denote $1 + |E(T)|$. First, replace each $v \in V(T)$ by a graph G_v of order $7s$. G_v is the union of three cliques L_v, M_v and R_v (the left clique, middle clique and right clique, respectively), each containing $3s$ vertices (see Figure 2.1). Of the $3s$ vertices in M_v , s belong also to L_v , another s belong also to R_v , and the remaining s do not belong to either L_v or R_v . L_v and R_v are disjoint.

The edges in T determine how the various replacement graphs intersect to form G_T . For each edge between vertices u and v in T , we identify a vertex in $M_u - (L_u \cup R_u)$ and a vertex in $M_v - (L_v \cup R_v)$. We do this in such a way that no vertex in G_T belongs to more than two middle cliques. This is possible since for each $v \in V(T)$, $|M_v - (L_v \cup R_v)| = s > |E(T)| \geq \delta(v)$. The order of G_T is at most $7s \times |V(T)|$, a

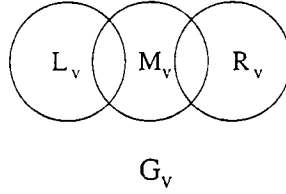


Figure 2.1: The graph G_v that replaces a vertex v .

polynomial in $|V(T)|$. Figures 2.2 and 2.3 show simple examples of the construction of G_T . In Figure 2.2, T consists of a pair of vertices adjacent by w edges. There are w vertices in $M_u \cap M_v$. Figure 2.3 shows a slightly more complicated example. All circles represent cliques of order twenty one ($21 = (6 + 1)3$). They have been given different dimensions in the figure so that their intersections can be drawn cleanly. The following relations hold: $|M_a \cap M_b| = 1$, $|M_a \cap M_c| = 3$, $|M_a \cap M_d| = 2$, and for $i \in \{a, b, c, d\}$, $|M_i \cap L_i| = |M_i \cap R_i| = 7$.

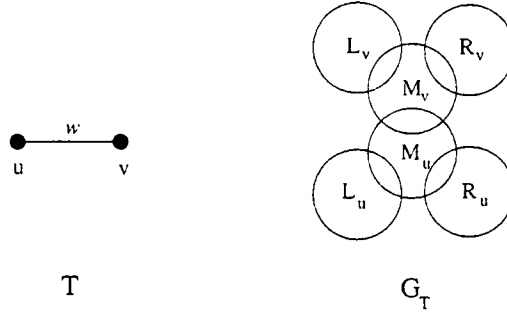


Figure 2.2: T and G_T , when T has order two.

Lemma 2.1 G_T is an AC graph.

Proof Let G denote the clique graph of G_T . It follows from the construction of G_T that each left clique, middle clique and right clique in it is maximal. So G contains $3|V(T)|$ vertices. Of these, the $2|V(T)|$ vertices that correspond to left and right cliques in G_T have degree one, and therefore, cannot be part of a cycle in G . Consider the subgraph G' of G induced by the $|V(T)|$ vertices that correspond to middle cliques in G_T . An edge exists between a pair of vertices in G' if and only if the

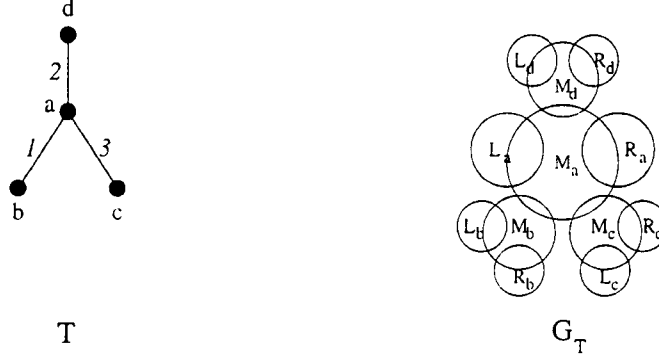


Figure 2.3: T and G_T , when T has order three.

corresponding middle cliques have a non-empty intersection. But each middle clique in G_T corresponds to a vertex in T , and two middle cliques intersect if and only if the corresponding vertices in T are adjacent. Thus there is a bijection from $V(G')$ to $V(T)$ such that two vertices are adjacent in G' if and only if they are adjacent in $V(T)$. In other words, G' is isomorphic to the tree obtained by replacing multiedges in T by single edges. Since G' , and hence G , is acyclic, the lemma follows. ■

In the lemmas that follow, G_T denotes the AC graph derived from T . L_v , M_v and R_v denote the left clique, middle clique and right clique in G_v , the subgraph of G_T that corresponds to v . We first establish two preliminary results about the width and structure of optimal path decompositions of G_T . We use the well-known fact that for any graph G , all vertices in a clique in G must be contained in a single subset of any path decomposition of G .

Lemma 2.2 $pw(G_T) \leq 4|E(T)| + 2$.

Proof Let A be the set of all vertices in G_T that are in some two middle cliques. Number the vertices in T at random from 1 to $|V(T)|$. Construct a path decomposition $X_1, \dots, X_{3|V(T)|}$ for G_T , where $X_{3i-2} = L_i \cup A$, $X_{3i-1} = M_i \cup A$, and $X_{3i} = R_i \cup A$, for $1 \leq i \leq |V(T)|$. This path decomposition has width $3(|E(T)| + 1) + |A| - 1 \leq 3(|E(T)| + 1) + |E(T)| - 1 = 4|E(T)| + 2$. ■

Let $X_1, \dots, X_r$ denote a path decomposition of a graph G . Let C_1, C_2, C_3 denote

cliques in G . Let i, j, k denote the smallest integers such that $V(C_1) \subseteq X_i$, $V(C_2) \subseteq X_j$ and $V(C_3) \subseteq X_k$. Then C_1 occurs *before*, *after* and *with* C_2 in the decomposition if $i < j$, $i > j$ and $i = j$ respectively. C_2 occurs *between* C_1 and C_3 if $i < j < k$ or $k < j < i$.

Lemma 2.3 Let u and v denote distinct vertices in T . Then, in any optimal path decomposition of G_T , M_v occurs between L_v and R_v , but not between L_u and R_u .

Proof We prove both parts of the lemma by contradiction. Suppose M_v does not occur between L_v and R_v . Of course M_v cannot occur with L_v or R_v , or else the path decomposition would have width at least $|M_v \cup L_v| - 1 = |M_v \cup R_v| - 1 = 5(|E(T)| + 1) - 1$, contradicting the upper bound of Lemma 2.2. So assume, without loss of generality, that R_v occurs between L_v and M_v . Then the first subset in the path decomposition that contains R_v must also contain all the vertices in $L_v \cap M_v$. The path decomposition would then have width at least $|R_v| + |L_v \cap M_v| - 1 = 4|E(T)| + 3$, contradicting optimality.

Now suppose M_v occurs between L_u and R_u . We have already shown that M_u must occur between L_u and R_u . M_v cannot occur with M_u , because then the path decomposition would have width at least $|M_v \cup M_u| - 1 \geq 5(|E(T)| + 1)$, which cannot be optimal. So assume, without loss of generality, that M_v occurs between M_u and R_u . Now the path decomposition has width at least $|M_v| + |M_u \cap R_u| - 1 \geq 4|E(T)| + 3$, again contradicting optimality. ■

We are now ready to present our main result, which relates the cutwidth of T to the pathwidth of $G(T)$. Consider a linear layout of a graph. For $v \in V(G)$, E_v denotes the set of edges cut by a line drawn between v and the vertex to its right in the layout. We define E_v to be $\emptyset$ if v is the rightmost vertex in the layout.

Lemma 2.4 Let s denote $|E(T)| + 1$. Then $cw(T) = pw(G_T) + 1 - 3s$.

Proof First we show that $cw(T) \leq pw(G_T) + 1 - 3s$. Consider an optimal path decomposition $X_1, \dots, X_r$ of G_T . Lay out the vertices of T according to the following

rule: For all $a, b \in V(T)$, if M_a occurs before M_b in $X_1, \dots, X_r$, lay out a before b . By Lemma 2.3, this produces a unique linear layout of T . Let v be a vertex in T such that $|E_v|$ is maximum. Assume, without loss of generality, that R_v occurs after M_v in $X_1, \dots, X_r$, and let i denote the smallest integer such that $R_v \subseteq X_i$. If an edge between two vertices u and w is in E_v , then, from Lemma 2.3 and the way we generated the layout, we infer that R_v occurs between M_u and M_w . This implies that $M_u \cap M_w \subseteq X_i$. The number of vertices in $M_u \cap M_w$ equals the number of edges in T between u and w . So X_i contains at least $|R_v| + |E_v| = 3s + |E_v|$ vertices. Since $pw(G_T) \geq |X_i| - 1$ and $cw(T) \leq |E_v|$, it follows that $cw(T) \leq pw(G_T) + 1 - 3s$.

Now we show that $cw(T) \geq pw(G_T) + 1 - 3s$. Given an optimal linear layout of T , we construct a path decomposition $X_1, \dots, X_{3|V|}$ for G_T as follows: For $v \in V(T)$, let $f(v)$ denote the position of v in the layout. For each v , place the vertices in L_v , M_v and R_v in the sets $X_{3f(v)-2}$, $X_{3f(v)-1}$ and $X_{3f(v)}$ respectively. For all adjacent vertices u and v in T such that $f(u) < f(v)$, place the vertices in $M_u \cap M_v$ in each of the sets $X_{3f(u)-1}, \dots, X_{3f(v)-1}$. Since every maximal clique in G_T is contained in some X_i , all vertices and edges in G_T are covered. Let a denote an arbitrary vertex in G_T . If a belongs to a left clique or a right clique, or to just one middle clique, then a is contained in exactly one subset or in a pair of consecutive subsets. If a belongs to more than one middle clique, it must belong to exactly two middle cliques (say M_u and M_v , where $f(u) < f(v)$), since G_T is an AC graph. Then it is an element of the sets $X_{3f(u)-1}, \dots, X_{3f(v)-1}$. In all cases, a is an element of a continuous sequence of sets, ensuring that the intersection property is satisfied. From our construction of $X_1, \dots, X_r$, it follows that an arbitrary subset X_i contains all the vertices in L_v , M_v , or R_v for some vertex v in T , in addition to all the vertices in $M_u \cap M_w$, where there is an edge in T between u and w that belongs to E_v . Then $|X_i| \leq 3s + |E_v| \leq 3s + cw(T)$, implying that $cw(T) \geq pw(G_T) + 1 - 3s$. ■

Thus, to determine the cutwidth of T , it suffices to construct G_T and then determine the pathwidth of G_T . The construction of G_T can be accomplished in polynomial

time, since the order and size of G_T are polynomial functions of the order and size of T . Hence we obtain the following theorem.

Theorem 2.1 It is $\mathcal{NP}$ -hard to determine the pathwidth of AC graphs.

2.4 Tree Decompositions

A *tree decomposition* of G is a pair (T, Y) , where T is a tree and $Y = \{Y_i \mid i \in V(T)\}$ is a collection of subsets of $V(G)$ such that the following two properties hold.

- Edge covering: for each edge $e \in E(G)$, some Y_i contains both endpoints of e .
- Intersection: for all $i, j, k \in V(T)$, if j is on the path between i and k in T , $Y_i \cap Y_k \subseteq Y_j$.

The *width* of a tree decomposition (T, Y) is one less than the size of the largest set in Y . The *treewidth* of G (denoted $tw(G)$) is the smallest width of all its tree decompositions.

Tree decompositions are of algorithmic interest primarily because the availability of a bounded-width tree decomposition permits fast algorithms for several important graph problems [ALS, Bod2]. Many natural graphs families have bounded treewidth [Bod1]; thus tree decompositions are of great utility. Our interest in tree decompositions stems from the fact that it is straightforward to convert them to path decompositions. We present a procedure, **td2pd**, that performs the conversion. Our procedure requires a routine to construct an optimal path decomposition of a tree. We employ the method presented in [EST], which takes $O(n \log n)$ time on a tree of order n . Since the run-time of **td2pd** is dominated by this routine, **td2pd** runs in $O(n \log n)$ time as well.

Procedure td2pd**Input:** A tree decomposition (T, Y) of a graph G .**Output:** A path decomposition of G .**begin procedure** $X_1, \dots, X_r :=$ an optimal path decomposition of T ;**for** $1 \leq i \leq r$ **do** $P_i := \bigcup_{j \in X_i} Y_j$;output $P_1, \dots, P_r$;**end procedure**

Theorem 2.2 Let (T, Y) denote a width- t tree decomposition of a graph G . Then $\text{td2pd}((T, Y))$ returns a path decomposition of G with width no more than $(t + 1)(pw(T) + 1) - 1$.

Proof Let $X_1, \dots, X_r$ denote the optimal path decomposition of T constructed in td2pd , and let $P_1, \dots, P_r$ denote the output of td2pd . Then, for $1 \leq i \leq r$, $|P_i| = |\bigcup_{j \in X_i} Y_j| \leq (t + 1)(pw(T) + 1)$. Thus the width condition is satisfied, and we only need to check that $P_1, \dots, P_r$ is a valid path decomposition of G .

It is easy to see that $P_1, \dots, P_r$ covers all edges in G . We prove by contradiction that $P_1, \dots, P_r$ has the intersection property. If the intersection property does not hold, then for some $1 \leq i < j < k \leq r$, there is a vertex v in $P_i \cap P_k$ that is not in P_j . Since $v \in P_i \cap P_k$, there must exist $l \in X_i$ and $m \in X_k$ such that v belongs to Y_l and Y_m . Consider the subsets V_1 and V_2 of $V(T)$, where $V_1 = \bigcup_{p < j} X_p - X_j$ and $V_2 = \bigcup_{p > j} X_p - X_j$. The intersection property of $X_1, \dots, X_r$ implies that V_1 and V_2 are disjoint. Moreover, there is no edge in T connecting V_1 and V_2 , because some X_q must contain both endpoints of such an edge, contradicting the disjointness of V_1 and V_2 . Thus every path between V_1 and V_2 in T contains a vertex from X_j . In particular, the path between l and m must contain a vertex, say h , from X_j . By the intersection property of (T, Y) , $v \in Y_h$. Since $h \in X_j$, $Y_h \subseteq P_j$ and $v \in P_j$, a contradiction. ■

If t is bounded by a constant and if $pw(T) \leq k \times pw(G)$ for some fixed k , then the width of the path decomposition returned by td2pd is within a constant factor

of $pw(G)$. The graphs we consider in this paper have bounded treewidth. Optimal tree decompositions for them can be constructed quickly. Thus, an algorithm to find a tree decomposition with $pw(T) \leq k \times pw(G)$ translates immediately into a relative approximation algorithm for pathwidth. We shall show that for k -chordal and outerplanar graphs, it is possible to ensure $pw(T) \leq pw(G)$. For series-parallel graphs, the guarantee is slightly worse, namely, $pw(T) \leq 2 \times pw(G) + 1$. For Halin graphs we are unable to obtain a good bound on $pw(T)$; instead, we approximate their pathwidth using a slightly different technique.

We say that (T, Y) is *simple* if T is a subgraph of G and $v \in Y_v$ for all $v \in V(T)$. (We insist that the labels of $V(T)$ respect those of $V(G)$.) Because pathwidth cannot be increased by taking a subgraph, if (T, Y) is simple, then $pw(T) \leq pw(G)$. Our algorithms construct (T, Y) by combining tree decompositions of G 's subgraphs. Let (T', Y') and (T'', Y'') denote tree decompositions of subgraphs G' and G'' , respectively. If $V(T')$ and $V(T'')$ are disjoint, and there are vertices $u \in V(T')$ and $v \in V(T'')$ such that all the vertices in $V(G') \cap V(G'')$ are in both Y'_u and Y''_v , then we may obtain a tree decomposition of $G' \cup G''$ by adding an edge between u and v . We call this a type (i) combination. If $V(T') \cap V(T'')$ is a singleton set $\{w\}$, and all the vertices in $V(G') \cap V(G'')$ are in Y'_w or Y''_w , then we may obtain a tree decomposition of $G' \cup G''$ by setting Y_w to $Y'_w \cup Y''_w$. We call this a type (ii) combination. If the edge uv exists in G' or G'' , simple tree decompositions are closed under type (i) combinations. In any case, they are closed under type (ii) combinations. Type (i) combinations preserve optimality. Type (ii) combinations preserve optimality if $|Y_w| \leq tw(G' \cup G'') + 1$.

2.5 Chordal Graphs

Recall that a graph is chordal if every induced cycle of length four or more in it contains a chord. Chordal graphs have been studied extensively over a long period, and several equivalent characterizations are known for them. The following result,

which characterizes chordal graphs in terms of a vertex elimination scheme, is useful to us.

Let G denote a graph of order n . A *simplicial vertex* in G is one whose neighborhood induces a clique in G ; all vertices of degree one are simplicial vertices. An ordering $(v_1, \dots, v_n)$ of $V(G)$ is said to be a *perfect elimination ordering* if, for $1 \leq i \leq n$, v_i is a simplicial vertex in the subgraph of G induced by $\{v_i, \dots, v_n\}$. G is chordal if and only if it possesses a perfect elimination ordering [FG].

Given a perfect elimination ordering, an optimal tree decomposition (T, Y) of G can be constructed using a straightforward recursive algorithm ([Klo, Lemma 2.1.13]). (T, Y) has width $k - 1$, where k denotes the clique number¹ of G . We modify the algorithm so that (T, Y) is also simple. We can then convert (T, Y) into a path decomposition of G of width at most $k \times (pw(G) + 1) - 1$, using **td2pd**. **Chordaltd**, our algorithm for constructing (T, Y) , requires a perfect elimination ordering as part of its input. Such an ordering can be constructed in $O(|E(G)|)$ time [RT].

Procedure chordaltd

Input: A chordal graph G and a perfect elimination ordering $(v_1, \dots, v_n)$ for G. Output: A simple optimal spanning tree decomposition (T, Y) of G. begin procedure if $V(G) = 1$ then $T := \{v_1\}$ and $Y := \{Y_{v_1}\}$, where $Y_{v_1} = \{v_1\}$; else begin $(T', Y') := \text{chordaltd}(G - \{v_1\}, (v_2, \dots, v_n))$; $T'' := \{v_1\}$ and $Y'' := \{Y_{v_1}\}$, where Y_{v_1} consists of v and its neighbors; $i :=$ the smallest number such that $v_1 v_i \in E(G)$; $T := T' \cup \{T''\} \cup \{v_1 v_i\}$ and $Y := Y' \cup Y''$; end output (T, Y); end procedure
--

Lemma 2.5 Let G denote a chordal graph with a perfect elimination ordering

¹The clique number of a graph is the order of the largest clique in it.

$(v_1, \dots, v_n)$. Let (T, Y) denote the result of the call to **chordaltd** $(G, (v_1, \dots, v_n))$. Then (T, Y) is a simple optimal spanning tree decomposition of G .

Proof The proof proceeds by induction on the order of G . We show the following: (T, Y) is simple and optimal, T spans G , and for each clique C in G , $V(C) \subseteq Y_{v_j}$, where v_j is the smallest-numbered vertex in C .

The lemma holds for the basis case, when G contains just one vertex. So let $(T', Y'), (T'', Y'')$ and i be as defined in **chordal-td**. Let N denote the set of neighbors of v_1 . Let G' and G'' denote the subgraphs of G induced by $\{v_2, \dots, v_n\}$ and $\{v_1\} \cup N$, respectively. Since v_1 is a simplicial vertex in G , N induces a clique in G , and hence, in G' . Note that v_i is the smallest-numbered vertex in N . By the induction hypothesis, (T', Y') is simple and optimal, T' spans G' , and $V(N) \subseteq Y_{v_i}$. (T'', Y'') is a simple optimal tree decomposition of G'' , which is a complete graph. Therefore, (T, Y) , which is a type (i) combination of (T', Y') and (T'', Y'') , with the edge $v_1 v_i$ existing in G'' , is simple. It is optimal because (T', Y') and (T'', Y'') are optimal. T spans G because $V(T) = V(T') \cup V(T'') = V(G') \cup \{v_1\} = V(G)$. The only cliques in G that are not in G' are those containing v_1 ; all of these cliques are contained in Y_{v_i} . Thus the induction is complete. ■

The steps in **chordaltd** that take more than constant time are those where the neighborhood of v_1 is determined, (T'', Y'') is constructed, and i is computed. Each of these operations can be performed in $O(\delta(v_1))$ time. Since n calls to **chordaltd** suffice to construct (T, Y) , the total complexity of the procedure is $O(\sum_{i=1}^n \delta(v_i)) = O(|E(G)|)$. Chordal graphs with clique number k contain at most kn edges. Therefore, if k is a constant, a perfect elimination ordering for G , and hence (T, Y) , can be constructed in $O(n)$ time. (T, Y) can be converted into a path decomposition of width at most $k \times (pw(G) + 1) - 1$ using **td2pd**, giving us the following result.

Theorem 2.3 Let G denote a chordal graph of clique number k , where k is a constant. Then a path decomposition of G with width at most $k \times (pw(G) + 1) - 1$ can

be constructed in linear time.

2.6 Outerplanar Graphs

A graph is *outerplanar* if it has a planar embedding with all vertices lying on a single face. Outerplanar graphs have treewidth at most two. In this section, we develop an algorithm, **op-td**, that, for an outerplanar graph G , constructs a width-two tree decomposition (T, Y) with $pw(T) \leq pw(G)$. By Theorem 2.2, running **td2pd** on (T, Y) produces a path decomposition with width at most $3 \times pw(G) + 2$.

Although an algorithm is known that achieves this bound [Yan], our method [GLY] is more streamlined, and has clearer proofs of correctness and complexity. We also note that **op-td** is subsumed by our own approximation algorithm for series-parallel graphs, which form a superclass of outerplanar graphs. In spite of this, we believe **op-td** is of interest, because it has a simpler implementation.

2.6.1 An Algorithm for Biconnected Graphs

To simplify our presentation, we concentrate initially on biconnected graphs (those without cut points).

Lemma 2.6 Let G be biconnected, outerplanar and of order at least three. Let v denote a vertex in G . Then G contains a path P with at least two edges, and with endpoints w and x , such that the following conditions hold:

- $G - (P - \{w, x\})$ is biconnected, outerplanar and contains v ,
- w and x are adjacent in G (and hence, in $G - (P - \{w, x\})$), and
- every edge in G is either in P or in $G - (P - \{w, x\})$.

Proof If G is a cycle, then the lemma is satisfied by setting P to $G - \{uv\}$, where u is a vertex adjacent to v . Otherwise, fix an outerplanar layout of G . Let E_i denote the set of internal edges of G (those not on the external face). Orient the layout so

that some edge e' is rightmost and some edge e'' is leftmost in E_i . If v is to the left of e' , then setting P to the path consisting of all the external edges to the right of e' satisfies the lemma. Otherwise, set P to the path containing all the external edges to the left of e'' . ■

If a path P contains at least two edges and has endpoints w and x , then it has a width-two tree decomposition (T, Y) such that $T = P - \{w, x\}$ and for $i \in V(T)$, $Y_i = \{x, i, j\}$, where j is the neighbor of i on w 's side (the sets Y_i actually form a path decomposition of P). We call (T, Y) a w -*extensible* tree decomposition of P . Figure 2.4 shows a path and its w -extensible tree decomposition. The sets Y_i are shown inside the ovals.

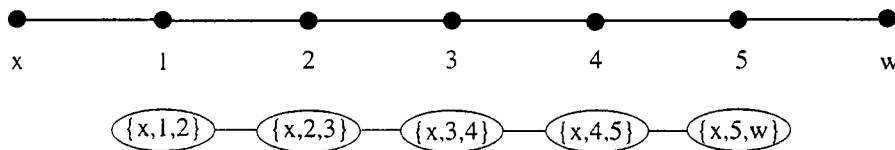


Figure 2.4: A path and its w -extensible tree decomposition.

Note that for every edge $ij \in E(P)$, either $\{i, j\} \subseteq Y_i$ or $\{i, j\} \subseteq Y_j$. We use the notion of extensibility to derive **bc-op-td**, our algorithm to construct simple optimal tree decompositions of biconnected outerplanar graphs.

At this point, we may as well assume that vertex v , which is part of the input to **bc-op-td**, is chosen at random. A specific choice of v is necessary when G is a biconnected component of a larger graph (see Section 2.6.3).

Lemma 2.7 Let G be biconnected and outerplanar, and let v denote a vertex in G . Let (T, Y) denote the result of the call to **bc-op-td** (G, v) . Then (T, Y) is a simple optimal tree decomposition of G , and T is a spanning tree of $G - \{v\}$.

Proof We prove, using induction on $|E(G)|$, a somewhat stronger result. We show that (T, Y) is simple and optimal, that T spans $G - \{v\}$, and that for each edge ij

Procedure bc-op-td

```

Input:   A biconnected outerplanar graph  $G$  of order two or more, and a vertex  $v$  in  $G$ .
Output: A simple optimal tree decomposition  $(T, Y)$  of  $G$ , with  $T$  spanning  $G - \{v\}$ .
begin procedure
  if  $|V(G)| = 2$ 
    then begin
       $u :=$  the vertex adjacent to  $v$ ;
       $T := \{u\}$  and  $Y := \{Y_u\}$ , where  $Y_u := \{u, v\}$ ;
    end
  else begin
     $P :=$  a path, between some two vertices  $w$  and  $x$ , that satisfies Lemma 2.6;
     $(T', Y') := \text{bc-op-td}(G - (P - \{w, x\}))$ ;
    if  $\{w, x\} \subseteq Y'_w$ 
      then begin
         $e :=$  the edge incident on  $w$  in  $P$ ;
         $(T'', Y'') :=$  a  $w$ -extensible tree decomposition of  $P$ ;
      end
    else begin
         $e :=$  the edge incident on  $x$  in  $P$ ;
         $(T'', Y'') :=$  an  $x$ -extensible tree decomposition of  $P$ ;
      end
     $T := T' \cup T'' \cup \{e\}$  and  $Y := Y' \cup Y''$ ;
  end
  output  $(T, Y)$ ;
end procedure

```

in G , either $i \in V(T)$ with $\{i, j\} \subseteq Y_i$ or $j \in V(T)$ with $\{i, j\} \subseteq Y_j$.

The lemma holds for the basis case, in which G contains just one edge. If $|E(G)| > 1$, let P , with endpoints w and x , denote a path that satisfies Lemma 2.6. Let G' denote the graph $G - (P - \{w, x\})$. Thus v is in G' . By the induction hypothesis, $\text{bc-op-td}(G')$ returns a simple optimal tree decomposition (T', Y') with T' spanning $G' - \{v\}$, and with $\{w, x\} \subseteq Y'_w$ or $\{w, x\} \subseteq Y'_x$. Assume, without loss of generality, that $\{w, x\} \subseteq Y'_w$. Let (T'', Y'') denote a w -extensible tree decomposition of P . Note that $T'' = P - \{w, x\}$ and that $\{w, x\} \subseteq Y''_a$, where a is w 's neighbor in P . T is formed by adding an edge between the vertex w in T' and the vertex a in T'' . The only vertices common to G' and P are w and x , which are contained in both Y'_w and Y''_a . Therefore (T, Y) is a valid tree decomposition of G . It is optimal because (T'', Y'')

is optimal and (T', Y') has width two. It is simple because (T', Y') and (T'', Y'') are both simple, with the edge wa existing in G . T' spans $G - \{v\}$ because T' spans $G' - \{v\}$ and T'' spans $P - \{w, x\}$. To complete the induction, observe that for each edge $ij \in E(G)$, $\{i, j\} \subseteq Y_i$ or $\{i, j\} \subseteq Y_j$, because either $\{i, j\} = \{w, a\} \subseteq Y_a''$ or $\{i, j\}$ is contained in one of Y_i', Y_j', Y_i'' or Y_j'' . ■

2.6.2 Efficiency

Let G , of order n , denote the input to **bc-op-td**. We store G in the doubly-linked adjacency list format described in Section 1.2. Note that the number of edges in G is linear in n ; in fact, since outerplanar graphs have treewidth at most two, there are fewer than twice as many edges as vertices. Let P , with endpoints w and x , and length p , denote a path satisfying Lemma 2.6. In each call to **bc-op-td**, the only steps that take more than constant time are (i) finding P , (ii) setting G to $G - (P - \{w, x\})$, and (iii) constructing a w - or x -extensible tree decomposition of P . Of these, steps (ii) and (iii) can be accomplished in $O(p)$ time. Since the order of G decreases by $O(p)$ after step (ii), the total number of operations performed in steps (ii) and (iii) during all the recursive calls to **bc-op-td** is $O(n)$. Step (i) is implemented as follows.

Some preprocessing is required. First, in linear time, we construct an outerplanar layout of G . Next we scan the layout in a clockwise direction, starting at an arbitrary vertex, and number vertices in the order in which they are encountered. Finally, we rearrange the adjacency list of each vertex a so that neighbors numbered lower than a occur before neighbors numbered higher than a . This rearrangement can be done in linear time by scanning the adjacency list of each vertex in turn, creating temporary lists holding lower-numbered and higher-numbered vertices, and appending the second list to the first.

Once the preprocessing is completed, paths that satisfy Lemma 2.6 are found during a second clockwise scan. A variable k , initialized to 1, stores the number of the last vertex scanned. During each call to **find-path**, the scan starts from this vertex

and continues until a satisfactory path is found. A stack *vstack*, initially empty, holds previously scanned vertices of degree three or more. We say that two vertices are *internal neighbors* if they are connected by an internal edge. For $1 \leq i \leq n$, v_i denotes the vertex numbered i .

Procedure find-path

```

Input:   A biconnected outerplanar graph  $G$ , where  $V(G) = \{v_1, \dots, v_n\}$ ,
           a stack, vstack, of vertices, and an integer  $k$ .
Output: A path  $P$  that satisfies Lemma 2.6.
begin procedure
  if  $G$  is a cycle
    then  $P := G - \{v_1v_2\}$ ;
  else begin
    while  $v_k$  has no lower-numbered internal neighbor
      do begin
        if  $\delta(v_k) \geq 3$  then push  $v_k$  on to vstack;
         $k := k + 1$ ;
      end
       $P :=$  the path  $v_j, \dots, v_k$ , where  $v_j$  is the vertex on top of vstack;
      if  $\delta(v_j) = 3$  then pop  $v_j$  from vstack;
    end
    output  $P$ ;
  end procedure

```

Lemma 2.8 Any path returned by **find-path** satisfies Lemma 2.6.

Proof Let G denote the graph input to **find-path**, and let $P = v_j, \dots, v_k$, where $j < k$, denote the path returned. The lemma holds if G is a cycle. So assume that G contains one or more internal edges. Let v_a and v_b denote, respectively, the lowest-numbered internal neighbor of v_k and the highest-numbered internal neighbor of v_j . From the description of **find-path**, it follows that all internal neighbors of v_j have higher numbers than j , and at least one internal neighbor of v_k has a lower number than k . So $a < k$ and $b > j$. For $j < i < k$, $\delta(v_i) = 2$, else either v_i would have been pushed after v_j on to *vstack*, or the path $v_j, \dots, v_i$ would have been returned. So

$a \leq j$ and $b \geq k$. It is not possible that $a < j$ and $b > k$ both, because the edges $v_a v_k$ and $v_j v_b$ would intersect, contradicting the outerplanarity of G . Thus either $a = j$ or $b = k$, implying that the internal edge $v_j v_k$ exists, and hence, that $G - (P - \{v_j, v_k\})$ is outerplanar. Every edge in G is either in P or in $G - (P - \{v_j, v_k\})$, because all internal vertices in P have degree two in G . ■

Lemma 2.9 If the input graph to **bc-op-td** is of order n , then the total amount of work done in **find-path**, during all the calls made to it, is linear in n .

Proof Observe that the variable k in **find-path** never exceeds n . This is because k is incremented only if no vertex numbered k or lower has a lower-numbered internal neighbor. Since k starts at 1, never decreases, and never exceeds n , the number of iterations of the while loop, summed over all the calls to **find-path**, is at most n . Note also that the total number of calls is at most n , since the order of the input graph decreases after each call. Thus each statement in **find-path** is executed $O(n)$ times. The only statement that takes more than constant time is the assignment of a value to P , which takes $O(|V(P)|)$ time. Since the order of G decreases by $O(|V(P)|)$ after each such assignment, the total amount of work done during this operation is also $O(n)$. ■

From Lemma 2.9 and the discussion after Lemma 2.7, it follows that **bc-op-td** has linear complexity.

2.6.3 Tackling Non-Biconnected Graphs

We now generalize our algorithm to handle all outerplanar graphs.

Lemma 2.10 Let G be outerplanar. Let (T, Y) denote the result of the call to **op-td**(G). Then (T, Y) is a simple optimal spanning tree decomposition of G .

Proof The proof proceeds by induction on the number of biconnected components of G . The basis case, when G is biconnected, follows from Lemma 2.7 and the modifica-

Procedure op-td

```

Input:   An outerplanar graph  $G$  of order two or more, and sets  $B$  and  $C$  of its
           biconnected components and cut points.
Output: A simple optimal tree decomposition  $(T, Y)$  of  $G$ , with  $T$  spanning  $G$ .
begin procedure
  if  $G$  is biconnected
    then begin
       $u, v :=$  any two adjacent vertices in  $G$ ;
       $(T', Y') := \text{bc-op-td}(G, v)$ ;
       $T := T' \cup \{v\} \cup \{uv\}$  and  $Y := Y' \cup \{Y_v\}$ , where  $Y_v = \{v\}$ ;
    end
  else begin
     $B_i :=$  an element of  $B$  that contains exactly one vertex  $v$  from  $C$ ;
    if  $v$  is not a cut point in  $G - (B_i - \{v\})$  then  $C := C - \{v\}$ ;
     $(T', Y') := \text{bc-op-td}(B_i, v)$ ;
     $(T'', Y'') := \text{op-td}(G - (B_i - \{v\}), B - \{B_i\}, C)$ ;
     $u :=$  an arbitrary neighbor of  $v$  in  $B_i$ ;
     $T := T' \cup T'' \cup \{uv\}$ , and  $Y := Y' \cup Y''$ ;
  end
  output  $(T, Y)$ ;
end procedure

```

tions made to (T, Y) after the call to $\text{bc-op-td}(G, v)$. So let $B_i, v, (T', Y'), (T'', Y'')$ and u be as defined in **op-td**. Let $\hat{G}$ denote $G - (B_i - \{v\})$. From the proof of Lemma 2.7, we know that (T', Y') is simple and optimal, that it spans $B_i - \{v\}$, and that $\{u, v\} \subseteq Y'_u$ (there is no Y'_v). By the induction hypothesis, (T'', Y'') is a simple optimal spanning tree decomposition of $\hat{G}$. Thus, by construction, (T, Y) is a simple optimal spanning tree decomposition of G . ■

Biconnected components and cut points can be found using a depth-first search. Procedure **op-td** builds an optimal tree decomposition using **bc-op-td**. This tree decomposition is converted into a path decomposition using **td2pd**. Recalling Theorem 2.2, and noting that **td2pd** is asymptotically the slowest of the aforementioned steps, we achieve the following result.

Theorem 2.4 If G is an outerplanar graph of order n , a path decomposition of G with width at most $3 \times pw(G) + 2$ can be constructed in $O(n \log n)$ time.

2.7 Series-Parallel Graphs

We attempt to generalize the work of the previous section to all graphs with treewidth at most two, a class better known as series-parallel graphs. Unfortunately, it is known from [Yan] that there are series-parallel graphs for which no simple optimal tree decompositions exist. Figure 2.5 shows one such graph.

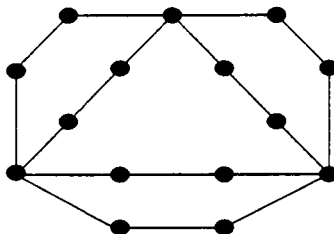


Figure 2.5: A difficult series-parallel graph.

Consequently, a different approach is necessary for series-parallel graphs that are non-outerplanar. For any such graph G , we describe how to construct a tree decomposition (T, Y) with $pw(T) \leq 2 \times pw(G) + 1$. Once more, we focus initially on biconnected graphs, as this enables us to take advantage of the following characterization: every biconnected series-parallel graph is two-terminal series-parallel.

2.7.1 Two-Terminal Series-Parallel Graphs

Two-terminal series-parallel graphs (2TSP graphs from now on) are defined in terms of a base graph and two composition operators. The base graph is a copy of K_2 , with one vertex designated as the “source” and the other as the “sink.” The *series* operator combines two graphs by identifying one’s source with the other’s sink. The *parallel* operator combines two graphs by identifying source with source and sink with sink.

A 2TSP graph G can be encoded by a directed binary *series-parallel decomposition tree* D that indicates how G is constructed from copies of the base graph and the two composition operators. If G has source u , sink v and just one edge uv , then D is a

single node with the label (u, v) . Otherwise, G must be formed by the composition of two smaller 2TSP graphs. If D_1 and D_2 denote series-parallel decomposition trees for these two graphs, then D is formed by joining the roots of D_1 and D_2 to a new node that will be the root of D . Figure 2.6 shows a 2TSP graph and one of its series-parallel decomposition trees. For clarity, internal nodes are marked with a “P” or an “S”, depending on whether they correspond to parallel or series compositions.

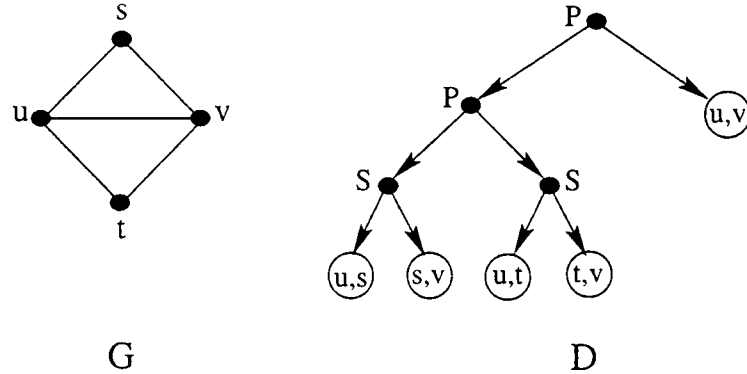


Figure 2.6: A 2TSP graph and a series-parallel decomposition tree for it.

We use the following notation for the sake of brevity. $G(D)$ denotes the 2TSP graph encoded by D , and $t(D)$ denotes its top node (i.e., root). If $x(D)$ is a node in D , $xl(D)$ and $xr(D)$ denote the left and right child of x , respectively. Thus any node in D can be referred to by a string that starts with t and is followed by a sequence of l 's and r 's. We drop the parenthetical D if there is no ambiguity about the tree under discussion. D_x denotes the subtree of D rooted at x .

The source and sink of x are defined to be the same as the source and sink of $G(D_x)$. If x is an internal node, x is either a *parallel node* or a *series node* depending on whether $G(D_x)$ is the result of a parallel composition or a series composition. If x is a series node, the *series-tree* of x is the subtree induced by x and all series nodes y such that there is a path consisting only of series nodes from x to y . Note that the children of the leaves in a series-tree are all non-series nodes.

2.7.2 Procedure Rearrange

If G is 2TSP, a series-parallel decomposition tree D for G can be constructed in linear time [VTL]. It is straightforward to obtain a tree decomposition (T, Y) of G from D . In order to obtain a relative pathwidth approximation, however, $pw(T)$ must be within a constant multiplicative factor of $pw(G)$. Not every tree T satisfies this property; for instance, it is easy to derive optimal tree decompositions of arbitrarily high pathwidth for simple cycles, which have pathwidth two.

We employ Procedure **rearrange** to obtain a good bound on $pw(T)$. This procedure rearranges some subtrees of D using a subroutine, **rearrange-top**, and assigns one of two colors to each node. These colors serve as a guide to our algorithm that converts series-parallel decomposition trees into tree decompositions.

Figure 2.7 illustrates the transformation performed by **rearrange-top** when invoked on a tree whose root is colored blue. The series tree on the left is transformed into the one on the right. All nodes marked with an S belong to the series-tree; the remaining nodes are non-series nodes. The figure depicts the case where the non-series nodes are leaf nodes. The transformation is identical if some or all of these are parallel nodes.

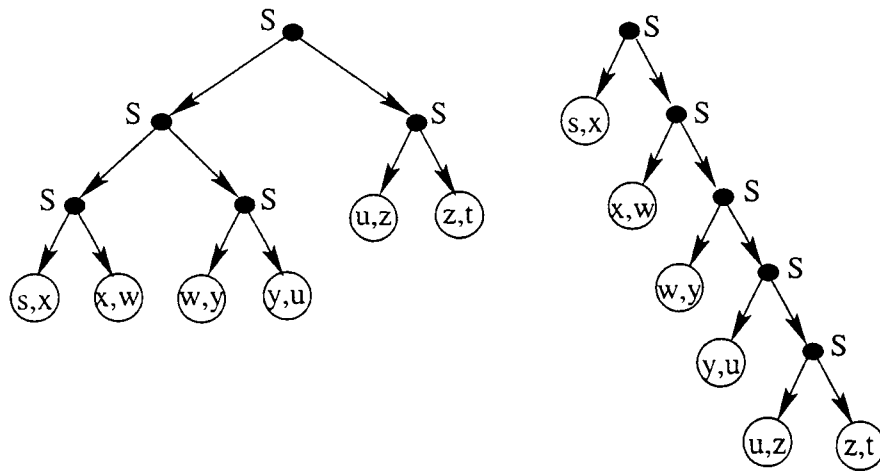


Figure 2.7: The transformation performed by **rearrange-top**

Procedure rearrange

Input: A series-parallel decomposition tree D and a color $c \in \{red, blue\}$.
Action: D is rearranged without affecting $G(D)$. Each node in D is colored red or blue.
begin procedure
 color of $t(D) := c$;
 if $t(D)$ is a parallel node
 then rearrange(D_{tl}, c), rearrange(D_{tr}, c);
 if $t(D)$ is a series node
 then begin
 rearrange-top(D);
 rearrange(D_{tl}, red), rearrange($D_{tr}, blue$);
 end
end procedure

Subroutine rearrange-top

Input: A series-parallel decomposition tree D with its root colored red or blue.
Action: The series tree of $t(D)$ is rearranged into a path without affecting $G(D)$.
begin subroutine
 $m :=$ the number of nodes in the series-tree of $t(D)$;
 $B :=$ an array containing the nodes of the series-tree in any order;
 if $t(D)$ is colored blue
 then begin
 $A :=$ an array containing the children of the leaves of the series-tree
 in left-to-right order;
 for i from 1 to $m - 1$ **do**
 left-child of $B[i] := A[i]$, right-child of $B[i] := B[i + 1]$;
 left-child of $B[m] := A[m]$, right-child of $B[m] := A[m + 1]$;
 end
 else begin
 $A :=$ an array containing the children of the leaves of the series-tree
 in right-to-left order;
 for i from 1 to $m - 1$ **do**
 left-child of $B[i] := B[i + 1]$, right-child of $B[i] := A[i]$;
 left-child of $B[m] := A[m + 1]$, right-child of $B[m] := A[m]$;
 end
 end subroutine

Lemma 2.11 Let D denote a series-parallel decomposition tree rooted at a series node that is colored red or blue. After a call to **rearrange-top**(D), the following properties hold:

- the series-tree of $t(D)$ is a path.

- if $t(D)$ is blue, then every node in the series tree has a non-series left child,
- if $t(D)$ is red, then every node in the series tree has a non-series right child, and
- $G(D)$ is unchanged.

Proof The first three properties follow from the description of **rearrange-top**. To show that the fourth holds. let $v_1, \dots, v_n$ denote the children of the leaves of the series-tree of $t(D)$, in left-to-right order. Then $G(D)$ is the result of composing in series the graphs encoded by $D_{v_1}, \dots, D_{v_n}$. **Rearrange-top** does not modify any of $D_{v_1}, \dots, D_{v_n}$, or change their left-to-right order. Since series compositions are associative, $G(D)$ stays the same. ■

After D is modified using **rearrange**, all parallel nodes in it have the same color as their children. Every series node has a red left child and a blue right child. Therefore, by Lemma 2.11, left children of blue series nodes and right children of red series nodes are non-series nodes. It follows that nodes that belong to the same series tree all have the same color.

The complexity of **rearrange** is dominated by the time spent in **rearrange-top**. During each call to **rearrange-top**, the number of operations performed is linear in the order of the series-tree that is rearranged. In the worst case, both this number and the number of calls to **rearrange-top** could be $\Theta(n)$, where n denotes the order of D . Although this seems to indicate that **rearrange** takes $\Theta(n^2)$ time, it can be implemented to run in linear time, as explained below.

Let x denote a series node in D . Consider the series tree of x after it has been rearranged into a path by **rearrange-top**. If x is colored blue, then every vertex in the path, except x itself, is the right child of the previous vertex. If x is colored red, then every vertex except x is the left child of the previous vertex. Moreover, all vertices in the path are given the same color as x by **rearrange**. This implies that further calls to **rearrange-top** on nodes in the series tree of x have no effect. All of these calls can be eliminated, leaving one call to **rearrange-top** for each maximal

series tree in D . The work done in the remaining calls is linear in the total number of vertices in all the disjoint series-trees in D , which is $O(n)$. Thus the total work done in **rearrange-top** is linear, and **rearrange** runs in linear time.

We say that D is *full* if for every parallel node in it, there is a leaf node with the same source and sink. In other words, if D is full, then the source and sink of every parallel node are adjacent in $G(D)$. Note that **rearrange** preserves fullness, because no parallel nodes are modified or created.

2.7.3 Constructing Tree Decompositions

After D has been rearranged, it is converted into a tree decomposition. Our conversion procedure, **sp2td**, constructs an optimal tree decomposition (T, Y) of $G(D)$. We prove that if D is full, then $pw(T) \leq pw(G)$. This result leads to a factor-of-three pathwidth approximation for biconnected outerplanar graphs, using a different approach than the one in Section 2.6. We also prove that a weaker bound, $pw(T) \leq 2 \times pw(G) + 1$, holds if D is not full. This result enables us to approximate the pathwidth of series-parallel graphs to within a factor of six of the optimum.

An example of the construction of (T, Y) is shown in Figures 2.8 through 2.11. Figures 2.8 and 2.9 show G and D . Figure 2.10 shows D after it is rearranged and colored using **rearrange**, assuming that $t(D)$ is colored blue. Nodes that are shown filled in are red; those that are not filled in are blue. Only one call to **rearrange-top** is needed; that call rearranges D_{ti} . (T, Y) itself is shown in Figure 2.11.

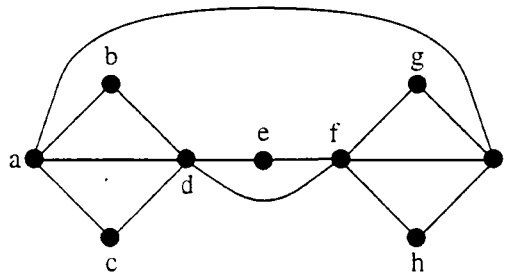


Figure 2.8: A 2TSP graph G .

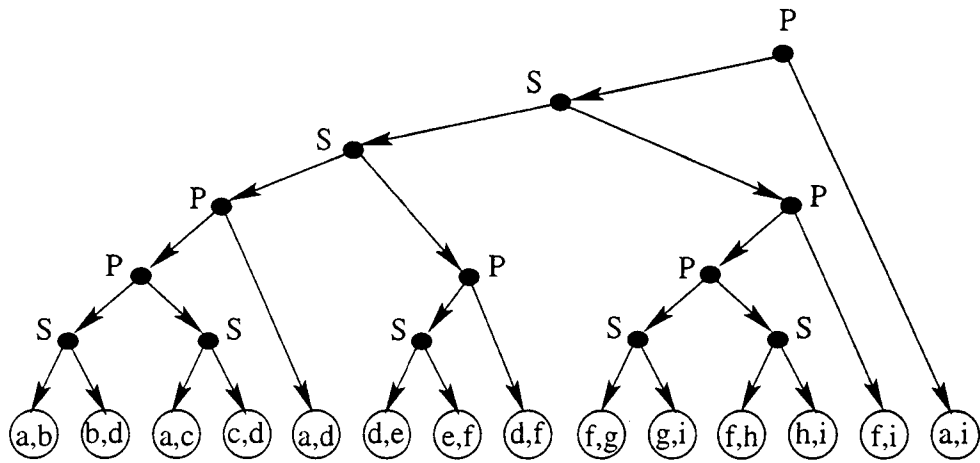


Figure 2.9: A series-parallel decomposition tree D for G .

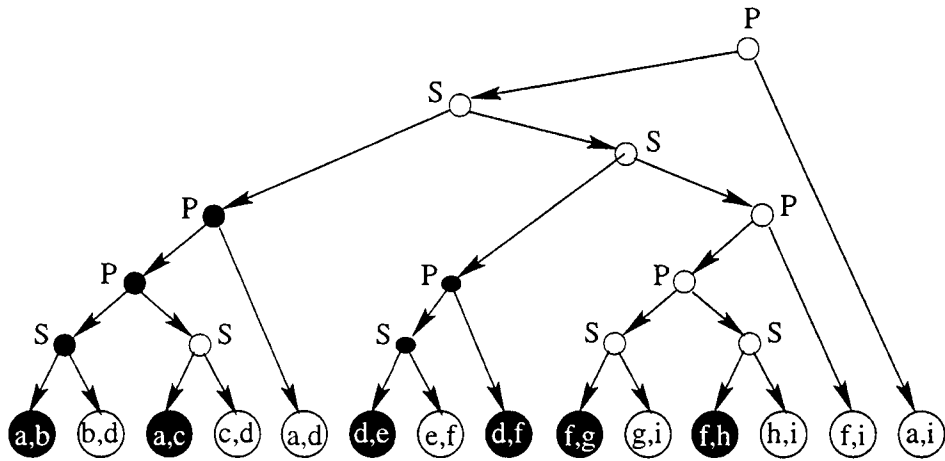


Figure 2.10: D after rearrangement and coloring.

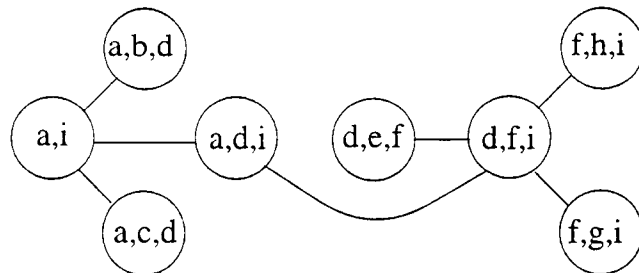


Figure 2.11: A tree decomposition of G .

Procedure sp2td

Input: A series-parallel decomposition tree D that has been rearranged and colored.

Output: A width-two tree decomposition of $G(D)$.

begin procedure

$s :=$ the source of $t(D)$ and $t :=$ the sink of $t(D)$;

if $t(D)$ is a leaf node

then if $t(D)$ is colored red

then $T := (\{t\}, \emptyset)$ and $Y := \{Y_t\}$, where $Y_t = \{s, t\}$;

else $T := (\{s\}, \emptyset)$ and $Y := \{Y_s\}$, where $Y_s = \{s, t\}$;

else begin

$(T', Y') := \text{sp2td}(D_{tl})$ and $(T'', Y'') := \text{sp2td}(D_{tr})$;

if $t(D)$ is a parallel node

then begin

$T := T' \cup T''$;

if $t(D)$ is colored red

then $Y := (Y' - \{Y'_t\}) \cup (Y'' - \{Y''_t\}) \cup \{Y_t\}$, where $Y_t = Y'_t \cup Y''_t$;

else $Y := (Y' - \{Y'_s\}) \cup (Y'' - \{Y''_s\}) \cup \{Y_s\}$, where $Y_s = Y'_s \cup Y''_s$;

end

else begin

$u :=$ the source of $tr(D)$;

if $t(D)$ is colored red

then begin

$T := T' \cup T'' \cup \{t\} \cup \{ut\}$;

$Y_u := \{s, u, t\}$ and $Y_t = \{s, t\}$;

$Y := (Y' - \{Y'_u\}) \cup (Y'' - \{Y''_u\}) \cup \{Y_u, Y_t\}$;

end

else begin

$T := T' \cup T'' \cup \{s\} \cup \{su\}$;

$Y_u := \{s, u, t\}$ and $Y_s = \{s, t\}$;

$Y := (Y' - \{Y'_u\}) \cup (Y'' - \{Y''_u\}) \cup \{Y_u, Y_s\}$;

end

end

end

 output (T, Y) ;

end procedure

Lemma 2.12 Let (T, Y) denote the result of the call to $\text{sp2td}(D)$, where D is a series-parallel decomposition tree that has been rearranged and colored using **rear-range**. Let s and t denote the source and sink of $t(D)$. Then the following properties hold:

- (T, Y) is a tree decomposition of $G(D)$ with width at most two,
- if $t(D)$ is colored red, then $V(T) = V(G(D)) - \{s\}$ and $Y_t = \{s, t\}$, and
- if $t(D)$ is colored blue, then $V(T) = V(G(D)) - \{t\}$ and $Y_s = \{s, t\}$.

Proof The proof proceeds by induction on the order of D . We analyze only the case where $t(D)$ is colored red, as the analysis when $t(D)$ is colored blue is similar. In the basis case, when $t(D)$ is a leaf node, $T = (\{t\}, \emptyset)$ and $Y = \{\{s, t\}\}$, and the lemma holds. So let (T', Y') , (T'', Y'') and u be as defined in **sp2td**. Let G , G' and G'' denote $G(D)$, $G(D_{tl})$ and $G(D_{tr})$, respectively.

If $t(D)$ is a parallel node, then both $tl(D)$ and $tr(D)$ must be colored red. By the induction hypothesis, (T', Y') and (T'', Y'') have width at most two, $V(T') = V(G') - \{s\}$, $V(T'') = V(G'') - \{s\}$ and $Y'_t = Y''_t = \{s, t\}$. (T, Y) is a type (ii) combination of (T', Y') and (T'', Y'') at t . $V(T) = (V(G') - \{s\}) \cup (V(G'') - \{s\}) = V(G) - \{s\}$. $Y_t = Y'_t \cup Y''_t = \{s, t\}$. (T, Y) has width at most two since (T', Y') and (T'', Y'') have width at most two, and since $|Y_t| = |\{s, t\}| < 3$.

If $t(D)$ is a series node, then $tl(D)$ must be red and $tr(D)$ must be blue. By the induction hypothesis, (T', Y') and (T'', Y'') have width at most two, $V(T') = V(G') - \{s\}$, $V(T'') = V(G'') - \{u\}$, $Y'_u = \{s, u\}$ and $Y''_u = \{u, t\}$. The construction of (T, Y) may be viewed as a type (ii) combination of (T', Y') and (T'', Y'') at u , followed by a type (i) combination of the resulting tree decomposition and (T''', Y''') , where $T''' = (\{t\}, \emptyset)$ and $Y''' = \{\{s, t\}\}$. $V(T) = (V(G') - \{s\}) \cup (V(G'') - \{t\}) \cup \{t\} = V(G) - \{s\}$. $Y_t = \{s, t\}$ because it is directly assigned that value. (T, Y) has width at most two since (T', Y') and (T'', Y'') have width at most two, and since both Y_u and Y_t contain no more than three elements. ■

2.7.4 Full Input

The result of Lemma 2.12 is not strong enough to translate to a relative pathwidth approximation, because it does not provide a bound for $pw(T)$ in terms of $pw(G)$. We

now prove that if D is full, then T is a subgraph of G , implying that $pw(T) \leq pw(G)$.

Lemma 2.13 Let (T, Y) denote the result of the call to `sp2td( $D$ )`, where D is a full series-parallel decomposition tree that has been rearranged and colored using `rearrange`. Let s and t denote the source and sink of $t(D)$. Then T spans $G(D) - \{s\}$ or $G(D) - \{t\}$, depending on whether $t(D)$ is colored red or blue.

Proof We examine the case when $t(D)$ is colored red (the other case is similar). From Lemma 2.12, it is already known that $V(T) = V(G(D)) - \{s\}$. It only remains to prove that $E(T) \subseteq E(G(D))$. We proceed by induction on the order of D . In the basis case, when $t(D)$ is a leaf node, $E(T) = \emptyset$, and the lemma holds. So let T', T'' and u be as defined in `sp2td`. If $t(D)$ is a parallel node, then $E(T) = E(T') \cup E(T'')$, and the lemma follows from the induction hypothesis. If $t(D)$ is a series node, then $E(T) = E(T') \cup E(T'') \cup \{ut\}$. By the induction hypothesis, $E(T') \cup E(T'') \subseteq E(G(D))$. Vertices u and t are the source and sink of $tr(D)$, which is a non-series node (Lemma 2.11). Because D is full, $ut \in E(G(D))$. ■

Thus $pw(T) \leq pw(G(D))$ if D is full. Unfortunately, in general, we do not know how to obtain full series-parallel decomposition trees for biconnected series-parallel graphs. In fact, as mentioned previously, not all of these graphs possess such decomposition trees. But if $G(D)$ is outerplanar, then we can obtain a satisfactory tree, as we now demonstrate.

Lemma 2.14 Let G denote a biconnected outerplanar graph of order two or more. Then G has a full series-parallel decomposition tree D .

Proof The proof proceeds by induction on $|V(G)|$. The lemma holds for the basis case, when $|V(G)| = 2$. So assume that G has order three or more. By Lemma 2.6, G contains a path P with endpoints w and x , such that $G - (P - \{w, x\})$ is biconnected and outerplanar, $wx \in E(G - (P - \{w, x\}))$, and every edge in G is either in P or in $G - (P - \{w, x\})$. Let G' denote $G - (P - \{w, x\})$. By the induction hypothesis, G' has

a full series-parallel decomposition tree D' . Since $wx \in E(G')$, there must be a leaf node in D' labeled (w, x) or (x, w) . Let y denote this node, and assume, without loss of generality, that y is labeled (w, x) . Any path has a series-parallel decomposition tree whose internal nodes are all series nodes and whose source and sink are equal to the endpoints of the path. Let D'' , with source w and sink x , denote such a tree for P . To obtain D , we attach D'' to D' at y by setting one child of y to a new node labeled (w, x) and the other child of y to the root of D'' . D contains only one parallel node that is not present in D' : the node y . As y has source w and sink x , and a leaf node labeled (w, x) exists in D , D stays full. ■

Note that the proof of Lemma 2.14 is constructive, meaning that it describes a procedure for building D . Let n denote the order of G . Then every step, except for the one where P is found, can be seen to take at most $O(n)$ time in total. It follows from the analysis in Section 2.6.2 that paths to play the role of P can be found in $O(n)$ time as well. Thus D can be constructed from G , and then converted into a path decomposition of width at most $3 \times pw(G) + 2$ using **sp2td** and **td2pd**, all in linear time.

2.7.5 Non-Full Input

If D is not full, then T is no longer a subgraph of $G(D)$. It turns out that the structure of T can still be related to that of $G(D)$, albeit in a more complicated fashion. To derive this relationship, we need to define a few terms and establish some preliminary results.

Given a graph G , G^2 denotes the graph with $V(G^2) = \{v_i \mid v \in V(G) \text{ and } i \in \{1, 2\}\}$, and $E(G^2) = \{u_i v_j \mid uv \in E(G) \text{ and } i, j \in \{1, 2\}\} \cup \{v_1 v_2 \mid v \in V(G)\}$. Informally, G^2 is the graph obtained by replacing each vertex in G by a copy of K_2 and each edge in G by edges between all pairs of vertices in the cliques corresponding to its end points. See Figure 2.12 for an example (K_4^- denotes K_4 minus an edge).

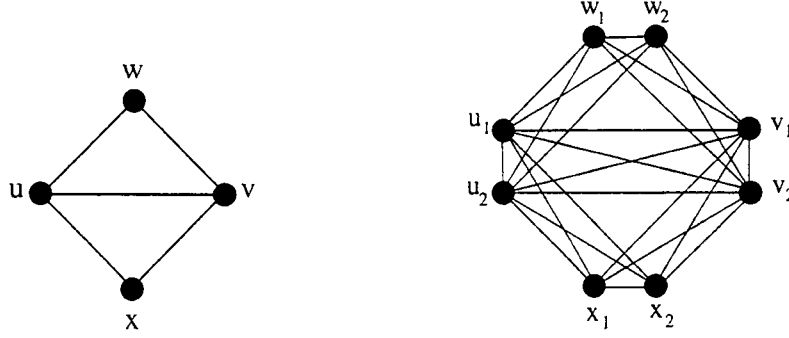


Figure 2.12: K_4^- and $(K_4^-)^2$.

Observation 2.1 For any graph G , $pw(G^2) \leq 2 \times pw(G) + 1$.

Proof Given a path decomposition of G with width m , replacing each vertex v by the two vertices v_1 and v_2 yields a path decomposition of G^2 with width $2m + 1$. ■

A graph H is said to be *topologically contained* in G (denoted $H \leq_t G$) if there exists an injection from $V(H)$ to $V(G)$ such that the images of adjacent vertices in H are connected by vertex-disjoint paths in G . The subgraph of G consisting of the images of all the vertices and edges in H is called a *model* of H in G . Suppose that $H \leq_t G^2$ and that $V(H) \subseteq V(G)$. Then we say that a model of H in G^2 is *direct* if each $v \in V(H)$ is mapped to $v_1 \in V(G^2)$. If $H \leq_t G$, then $pw(H) \leq pw(G)$ [RS2]. If $H \leq_t G^2$, then $pw(H) \leq pw(G^2) \leq 2 \times pw(G) + 1$. Our performance guarantee for **sp2td** relies on this bound.

Let x denote a parallel node in a series-parallel decomposition tree D . Then the *left set* of x is the set of all vertices v in $G(D)$ such that the subgraph of D induced by nodes with v as source or sink is contained in D_{xl} , but not in D_{yl} for any parallel node y in D_{xl} . In other words, v is in the left set of x if and only if x is the lowest parallel node whose left subtree contains all nodes with v as source or sink. For example, in Figure 2.6, vertex t belongs to the left set of $t(D)$. Vertex s belongs to the left set of $tl(D)$. Vertices u and v are not in the left set of any node.

We make use of the following observations.

- The left sets of distinct nodes are mutually disjoint.

This observation is a direct consequence of the definition of left sets.

- If y denotes a parallel node in D_{xl} , then replacing D_y by D_{yr} does not add any vertices to the left set of x .

The observation follows from noting that, if z is any node that remains in D after D_y is replaced by D_{yr} , then the lowest parallel node whose left subtree contains z stays the same.

Lemma 2.15 Let x denote a parallel node in a series-parallel decomposition tree D . Let s and t denote the source and sink of x . Then $G(D)$ contains a path P with endpoints s and t , and with all its internal vertices in the left set of x .

Proof The proof proceeds by induction on the number of parallel nodes in D_{xl} . In the basis case, when D_{xl} contains no parallel nodes, the left set of x consists of all the vertices in $G(D_{xl})$ except s and t . Since $G(D_{xl})$ is a connected subgraph of $G(D)$, P exists. So let y denote a parallel node in D_{xl} . Construct a tree D' by replacing D_y with D_{yr} . As observed before, this operation does not add any new vertices to the left set of x . By the induction hypothesis, $G(D')$ contains P . Since $G(D')$ is a subgraph of $G(D)$, $G(D)$ contains P as well. ■

Lemma 2.16 Let (T, Y) denote the result of the call to **sp2td**(D), where D is a series-parallel decomposition tree that has been rearranged and colored using **rear-range**. Suppose $t(D)$ is colored red. Let s denote the source of $t(D)$ and let G denote $G(D)$. Then there is a direct model of T in $(G - \{s\})^2$.

Proof We make use of the following observation about **sp2td**. Suppose D contains a parallel node x with source u and sink v . Let (T', Y') , (T'', Y'') and (T''', Y''') denote the tree decompositions of D_{xl} , D_{xr} and D_x constructed by **sp2td**. Then $T''' = T' \cup T''$, and $Y''' = (Y' - \{Y'_u\}) \cup (Y'' - \{Y''_u\}) \cup \{Y_u\}$, where $Y_u = Y'_u \cup Y''_u$. By

Lemma 2.12, $Y_u = Y'_u = Y''_u = \{u, v\}$. Now suppose the left child of x is a leaf node. Then $T' = (\{v\}, \emptyset)$ and $Y' = \{u, v\}$. It follows that $T''' = T''$ and $Y''' = Y''$. Thus, if xl is a leaf node, then $(T''', Y''') = (T'', Y'')$.

Conversely, if a leaf node with source u and sink v does not exist, then we can add a new node y between x and its parent, and set the left child of y to a leaf node z labeled (u, v) and the right child of y to x , without affecting (T, Y) . We give y and z the same color as x , to be consistent with the way **rearrange** colors nodes. Let D_+ denote the tree obtained after this operation has been performed on all parallel nodes in D whose source and sink are not adjacent in $G(D)$. Note that D_+ is a full tree. As we have observed, **sp2td** constructs identical tree decompositions from D_+ and D .

Consequently, by Lemma 2.13, (T, Y) is an optimal tree decomposition of $G(D_+)$, with T spanning $G(D_+) - \{s\}$. We now describe a direct model of T in $(G - \{s\})^2$. Map each vertex v in T to the vertex v_1 in $(G - \{s\})^2$. Map each edge uv in T to the edge u_1v_1 , if u_1v_1 exists in $(G - \{s\})^2$. Since T is a subgraph of $G(D_+)$, any unmapped edges must belong to $E(G(D_+)) - E(G)$. We map each such edge xy to a path P between x_1 and y_1 such that (i) all internal vertices in P have subscript 2, and (ii) for each internal vertex z_2 , the corresponding vertex z in G is in the left set of a parallel node in D with x and y as its source and sink. Lemma 2.15 guarantees the existence of P . Since left sets of distinct parallel nodes are disjoint, paths that are the images of distinct edges do not share any internal vertices. Moreover, they do not contain the images of vertices in T , except as endpoints. To complete the proof, observe that none of these paths contains s_1 or s_2 , since s is not in the left set of any node in D . ■

Since $T \leq_t (G - \{s\})^2$ and $pw((G - \{s\})^2) \leq pw(G^2)$, $pw(T) \leq 2 \times pw(G) + 1$.

2.7.6 Tackling Non-Biconnected Graphs

In this subsection, we generalize our algorithm to handle all series-parallel graphs. Recall that the biconnected components of a series-parallel graph are 2TSP. Our procedure, **sp-td**, makes use of the following fact: The end points of any edge in a biconnected 2TSP graph can be made the source and sink of the root of a series-parallel decomposition tree for the graph [VTL].

Procedure **sp-td**

```

Input:   A series-parallel graph  $G$  of order two or more, and lists  $B$  and  $C$  of its
           biconnected components and cut points.
Output: A tree decomposition  $(T, Y)$  of  $G$  with width at most two and with  $T \leq_t G^2$ .
begin procedure
  if  $G$  is biconnected
    then begin
       $v, u :=$  an arbitrary pair of adjacent vertices in  $G$ ;
       $D :=$  a series-parallel decomposition tree for  $G$ 
        with source  $v$  and sink  $u$ ;
      rearrange( $D, red$ );
       $(T', Y') := \mathbf{sp2td}(D)$ ;
       $T := T' \cup \{v\} \cup \{uv\}$  and  $Y := Y' \cup \{Y_v\}$ , where  $Y_v = \{v\}$ ;
    end
  else begin
     $B_i :=$  an element of  $B$  that contains exactly one vertex  $v$  from  $C$ ;
    if  $v$  is not a cut point in  $G - (B_i - \{v\})$  then  $C := C - \{v\}$ ;
     $D :=$  a series-parallel decomposition tree for  $B_i$ 
      with source  $v$  and sink  $u$ ;
    rearrange( $D, red$ );
     $(T', Y') := \mathbf{sp2td}(D)$ ;
     $(T'', Y'') := \mathbf{sp-td}(G - (B_i - \{v\}), B - \{B_i\}, C)$ ;
     $u :=$  an arbitrary neighbor of  $v$  in  $B_i$ ;
     $T := T' \cup T'' \cup \{uv\}$  and  $Y := Y' \cup Y''$ ;
  end
  output  $(T, Y)$ ;
end procedure

```

Lemma 2.17 Let G denote a series-parallel graph of order two or more. Then the result of the call to **sp-td**(G) is a tree decomposition (T, Y) of G with width at most

two and with $T \leq_t G^2$.

Proof The proof proceeds by induction on the number of biconnected components of G . We prove the following: (T, Y) is a width-two tree decomposition of G , $V(T) = V(G)$, $w \in Y_w$ for all $w \in V(T)$, and G^2 contains a direct model of T .

The basis case, when G is biconnected, follows from Lemma 2.16 and the modifications made to (T, Y) after the call to **sp2td**(D). So let $B_i, v, (T', Y'), (T'', Y'')$ and u be as defined in **sp-td**. Let $\hat{G} = G - (B_i - \{v\})$. By Lemmas 2.12 and 2.16, (T', Y') has width at most two, $V(T') = V(B_i) - \{v\}$, $Y'_u = \{u, v\}$, and $(B_i - \{v\})^2$ contains a direct model of T' . By the induction hypothesis, (T'', Y'') has width at most two, $V(T'') = V(\hat{G})$, $v \in Y''_v$, and $\hat{G}^2$ contains a direct model of T'' . (T, Y) is a type (i) combination of (T', Y') and (T'', Y'') , and is, therefore, a tree decomposition of G with width at most two. $V(T) = V(B_i) - \{v\} \cup V(\hat{G}) = V(G)$. For each $w \in V(T)$, $w \in Y_w$, because $w \in Y'_w$ or $w \in Y''_w$. A direct model of T in G^2 can be obtained by joining the models of T' in $(B_i - \{v\})^2$ and T'' in $\hat{G}^2$ by the edge $u_1 v_1$. ■

The complexity of **sp-td** is determined by that of **rearrange** and **sp2td**. Since both these routines run in linear time, **sp-td** is linear-time as well. Thus we achieve the following result.

Theorem 2.5 Let G denote a series-parallel graph of order n . Then a path decomposition of G with width at most $6 \times pw(G) + 5$ can be constructed in $O(n \log n)$ time.

Proof Lemma 2.17 implies that a width-two tree decomposition (T, Y) of G , with $pw(T) \leq pw(G^2) \leq 2 \times pw(G) + 1$, can be constructed. (T, Y) can be converted into a path decomposition with width at most $6 \times pw(G) + 5$ using **td2pd**. Since the construction of (T, Y) takes linear time and the conversion using **td2pd** takes $O(n \log n)$ time, the theorem follows. ■

2.8 Graph Decompositions

The properties of tree decompositions that enable their conversion to path decompositions are shared by other types of decompositions. For instance, one can drop the requirement that the underlying graph of the decomposition be a tree. It is also possible to relax the edge covering requirement. Below, we define graph decompositions, relaxing these two requirements of tree decompositions. A similar notion can be found in [Bod3].

A *graph decomposition* of a graph G is a pair (H, Y) , where H is a graph and $Y = \{Y_i \mid i \in V(H)\}$ is a collection of subsets of $V(G)$ such that the following two properties hold.

- Weak edge covering: for each edge $uv \in E(G)$, either $\exists i \in V(H)$ such that $\{u, v\} \subseteq Y_i$, or $\exists i, j \in E(H)$ such that $\{u\} \subseteq Y_i$ and $\{v\} \subseteq Y_j$.
- Intersection: $\forall v \in V(G)$, the set $\{Y_i \mid v \in Y_i\}$ is either singleton or induces a connected subgraph of H .

The *width* of a graph decomposition (H, Y) is one less than the size of the largest set in Y . H is said to be the *underlying graph* of the decomposition.

Recall that path and tree decompositions require a stronger notion of edge covering than graph decompositions. Thus a graph decomposition with a path or a tree as its underlying graph need not be a valid path or tree decomposition. Weakening the edge covering property allows us to construct decompositions of lower width than would be possible otherwise. As a trivial illustration, any complete graph G has a width 0 decomposition whose underlying graph is G itself. But if we were to insist that graph decompositions obey the stronger edge covering property of path and tree decompositions, then any decomposition with G as its underlying graph would have width at least $|V(G)| - 1$.

Although the intersection property of graph decompositions appears to be a different notion from that of path and tree decompositions, it is not. If H is a path or

a tree, then the two definitions are identical. So path and tree decompositions are valid graph decompositions.

We are particularly interested in decompositions whose underlying graphs are outerplanar. We call such decompositions *outerplanar decompositions*. To our knowledge, outerplanar decompositions have not been studied previously.

In Section 2.4, we presented a procedure, **td2pd**, to convert a tree decomposition to a path decomposition. It is straightforward to generalize **td2pd** so that it accepts a graph decomposition as input. The new procedure, which we call **gd2pd**, is given below.

Procedure gd2pd

Input: A graph decomposition (H, Y) of a graph G .

Output: A path decomposition of G .

begin procedure

$X_1, \dots, X_r :=$ a path decomposition of H ;

for $1 \leq i \leq r$ **do**

$P_i := \bigcup_{j \in X_i} Y_j$;

output $P_1, \dots, P_r$;

end procedure

Theorem 2.6 Let (H, Y) denote a width- t graph decomposition of a graph G . Let p denote the width of the path decomposition $X_1, \dots, X_r$ that is constructed in the first step of **gd2pd**. Then the call to **gd2pd** $((H, Y))$ returns a path decomposition of G with width no more than $(t + 1)(p + 1) - 1$.

Proof The proof is very similar to that of Theorem 2.2. Let $P_1, \dots, P_r$ denote the output of **gd2pd**. Then, for $1 \leq i \leq r$, $|P_i| = |\bigcup_{j \in X_i} Y_j| \leq (t + 1)(p + 1)$. Thus the width condition is satisfied, and we only need to check that $P_1, \dots, P_r$ is a valid path decomposition of G .

Observe that the weak edge covering property of (H, Y) and the edge covering property of $X_1, \dots, X_r$ together imply that $P_1, \dots, P_r$ covers all edges in G .

We prove by contradiction that $P_1, \dots, P_r$ has the intersection property. If this property does not hold, then for some $1 \leq i < j < k \leq r$, there is a vertex v in $P_i \cap P_k$ that is not in P_j . Consider the subsets V_1 and V_2 of $V(H)$, where $V_1 = \bigcup_{p < j} X_p - X_j$ and $V_2 = \bigcup_{p > j} X_p - X_j$. The intersection property of $X_1, \dots, X_r$ implies that V_1 and V_2 are disjoint. Moreover, there is no edge in H connecting V_1 and V_2 , because some X_q must contain both endpoints of such an edge, contradicting the disjointness of V_1 and V_2 . Thus, any subset of $V(H)$ that contains at least one vertex from each of V_1 and V_2 must induce a disconnected subgraph of H . But the set $\{i \mid v \in Y_i\}$ is such a set (this set contains at least one vertex from X_i and one from X_k), contradicting the intersection property of (H, Y) . ■

The width of the path decomposition returned by **gd2pd** is within a constant factor of $pw(G)$ if (i) t is a constant and (ii) $p \leq k \times pw(G)$ for some fixed k . We shall show that, for Halin graphs, both conditions can be satisfied. Thus, for this class of graphs, **gd2pd** can be used to derive a relative approximation algorithm for pathwidth.

2.9 Halin Graphs

A Halin graph is a planar graph obtained by taking a planar embedding of a tree with no degree-two vertices and connecting all the leaf nodes in a cycle (see Figure 2.13 for an example). A given planar embedding of a tree determines a unique cyclic order on its leaves, and hence, determines a unique Halin graph [Bon]. It is fairly straightforward to show that Halin graphs have treewidth at most three, and to construct optimal tree decompositions [Wim]. It appears difficult, however, to obtain decompositions with underlying trees of small pathwidth relative to that of the input graph. Therefore, we employ graph decompositions instead, and present an algorithm to construct bounded-width outerplanar decompositions of Halin graphs. We prove that this method leads to path decompositions of width at most eighteen times the

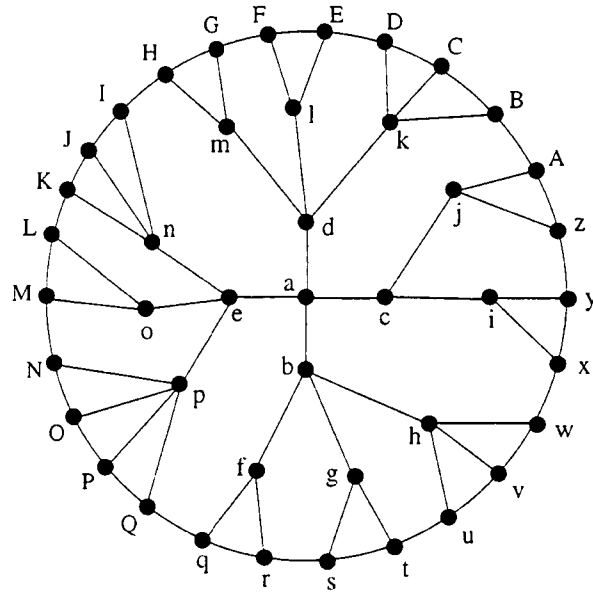


Figure 2.13: A Halin graph.

optimum; this is probably a loose upper bound.

2.9.1 Outerplanar Decompositions

We now present our procedure, **halin-gd**, that constructs outerplanar decompositions of Halin graphs. Figures 2.14 through 2.16 show an example of the operation of **halin-gd**. Figure 2.14 shows the graph of Figure 2.13 after it has been rooted, its leaves numbered, and `lowest.leaf` and `highest.leaf` values computed. The root node is at the center of the circle. The node labeled *q* in Figure 2.13 has been given the number 1. Figure 2.15 shows the outerplanar graph constructed from the graph of Figure 2.14. Dashed and bold lines represent edges that have been deleted from and added to the Halin graph, respectively. Figure 2.16 shows the outerplanar decomposition of the Halin graph of Figure 2.13. Note that the decomposition has width two.

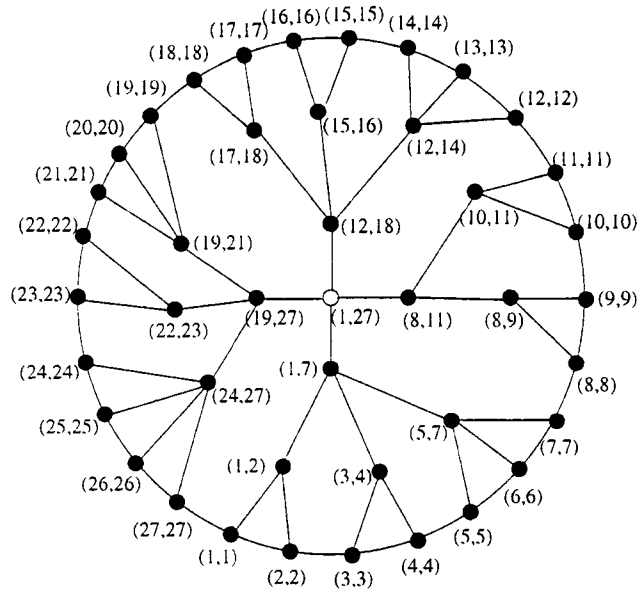


Figure 2.14: Halin graph after preprocessing.

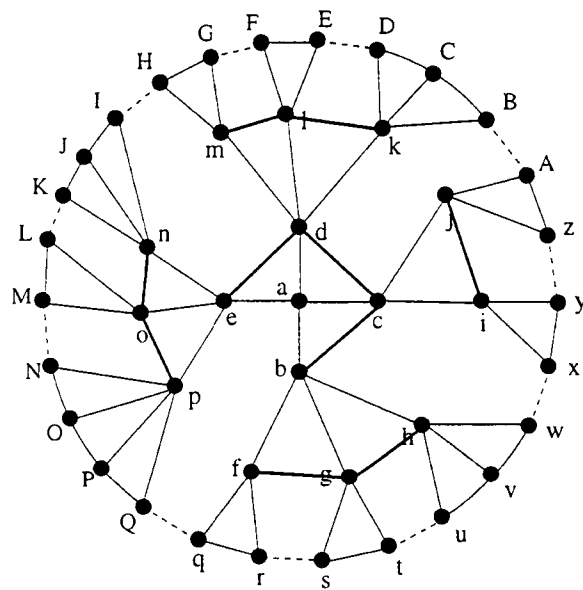


Figure 2.15: Outerplanar graph obtained from Halin graph.

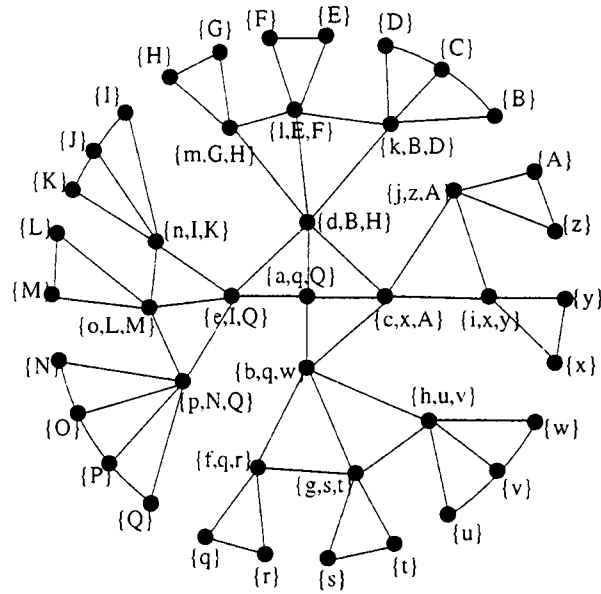


Figure 2.16: Outerplanar decomposition of Halin graph.

Procedure halin-gd

Input: A Halin graph G .

Output: A width-two outerplanar decomposition (G', Y') of G .

begin procedure

find a tree T and a cycle C on the leaves of T , such that $G = T \cup C$;

choose an arbitrary internal vertex r of T as the root;

$l :=$ the number of leaves in T ;

number the leaves of T from 1 to l so that the numbers are consistent with their order in C and so that the leaves numbered 1 and l belong to different subtrees of r ;

for each vertex $v \in V(G)$

do begin

$lowest_leaf(v) :=$ the leaf node with the lowest number in the subtree of T rooted at v ;

$highest_leaf(v) :=$ the leaf node with the highest number in the subtree of T rooted at v ;

end

$G' := T \cup E$, where $E = \{ij \mid i \text{ and } j \text{ are siblings in } T \text{ such that } highest_leaf(i) \text{ and } lowest_leaf(j) \text{ have consecutive numbers}\}$;

for each vertex $i \in V(G')$ **do**

$Y'_i := \{i\} \cup \{lowest_leaf(i)\} \cup \{highest_leaf(i)\}$;

output (G', Y') ;

end procedure

2.9.2 Correctness and Complexity

We address the correctness and complexity of **halin-gd**. In the rest of this section, $G, T, C, r, l, \text{lowest_leaf}, \text{highest_leaf}, G'$ and Y' are defined as in **halin-gd**. We make a few preliminary observations.

Observation 2.2 Given G, T and C can be found in linear time.

Proof If G has order four, then pick any three vertices in G as the leaves of T . Otherwise, the leaves of T are precisely those vertices of degree three, whose neighbors are each contained in at least one triangle. Once the leaves are known, T and C can be easily determined. ■

Observation 2.3 G' is an outerplanar graph with the same vertex set as G .

Proof G' has the same vertex set as T , which has the same vertex set as G . Each biconnected component of G' is a subgraph induced by an internal vertex of T and its children. The biconnected components of G' are outerplanar, implying that G' itself is outerplanar. ■

G and G' are almost alike in structure. The difference is that in G' , some of the edges of C have been deleted, and some edges between siblings in T have been added. There is a natural correspondence between the missing edges and the new edges, as suggested in Figure 2.15. Our construction of (G', Y') is based on this correspondence, even though this is not immediately obvious. For each vertex v in G , let L_v and R_v denote the sets $\{w \mid \text{lowest_leaf}(w) = v\}$ and $\{w \mid \text{highest_leaf}(w) = v\}$, respectively. Then L_v consists of the nodes on the path between v and the highest (that is, closest to the root) node x in T such that $\text{lowest_leaf}(x) = v$. Similarly, R_v consists of the nodes on the path between v and the highest node y in T such that $\text{highest_leaf}(y) = v$. If v is an internal node in T , then $L_v = R_v = \emptyset$. If v is a leaf node in T , then at least one of L_v and R_v is equal to $\{v\}$.

Lemma 2.18 (G', Y') is a width-two outerplanar decomposition of G .

Proof (G', Y') has width two, because each Y'_i contains at most three elements. Let v denote a vertex in G . To show that (G', Y') has the intersection property, we will prove that $\{i \mid v \in Y'_i\}$ is either singleton, or induces a path in G' . Note that $\{i \mid v \in Y'_i\} = \{v\} \cup L_v \cup R_v$. If v is an internal node in T , then $\{v\} \cup L_v \cup R_v = \{v\}$ is a singleton set. If v is a leaf node, then $\{v\} \cup L_v \cup R_v$ is equal to either L_v or R_v , depending on which of L_v and R_v is equal to $\{v\}$. In any case, this set is either singleton or induces a path in T , and hence, in G' .

It only remains to prove that the weak edge covering property holds. Clearly, (G', Y') covers (weakly) all the edges in T . So let ab , where b has a higher number than a , denote an edge in C . Either a is numbered 1 and b is numbered l , or a and b have consecutive numbers. In the former case, $a = \text{lowest_leaf}(r)$ and $b = \text{highest_leaf}(r)$, where r is the root of T . The edge ab is strongly covered since a and b are both elements of Y'_r . In the latter case, the lowest common ancestor of a and b in T must have a pair of children c and d such that $\text{highest_leaf}(c) = a$ and $\text{lowest_leaf}(d) = b$. By our construction of G' , we know that the edge cd exists in G' . Moreover, $a = \text{highest_leaf}(c) \in Y'_c$ and $b = \text{lowest_leaf}(d) \in Y'_d$. Thus ab is weakly covered. ■

Each step in **halin-gd** can be implemented to run in linear time. G and G' are represented as doubly-linked adjacency lists. Observation 2.2 explains how to find T and C . The leaves of T can be numbered, and lowest_leaf and highest_leaf values computed, by simple recursive procedures. G' can be constructed by copying G , deleting some of the edges incident on the leaves of T , and adding some edges between siblings in T . Edge-addition is a constant time operation; edge-deletion is constant-time if the endpoints have bounded degree, as is the case here. Each set Y'_i can be constructed in constant time, and there are only a linear number of these sets. Overall, **halin-gd** runs in linear time.

2.9.3 Performance Bounds

We have proved that a width-two outerplanar decomposition (G', Y') of G can be constructed in linear time. A path decomposition of G' , of width at most $3 \times pw(G') + 2$, can be obtained using **sp-td** and **td2pd**. Since (G', Y') has width two, we can convert this into a path decomposition of G of width at most $3 \times (3 \times pw(G') + 2) = 9 \times pw(G') + 6$, using **gd2pd**. To show that this is a relative approximation, we must bound $pw(G')$ by a linear function of $pw(G)$. We do this by demonstrating the existence of a width-one graph decomposition (G, Y) of G' , implying that $pw(G') \leq 2 \times pw(G) + 1$.

For each node v in T , let $toyleft(v)$ denote the highest (that is, closest to the root) node x in T , excluding the root itself, such that $lowest_leaf(v) = lowest_leaf(x)$. Let $topright(v)$ denote the highest node x , excluding the root, such that $highest_leaf(v) = highest_leaf(x)$. Observe that the set $\{w \mid toyleft(w) = x\}$ is identical to $L_v - \{r\}$, where r is the root. Similarly, $\{w \mid topright(w) = x\} = R_v - \{r\}$.

Lemma 2.19 For each vertex $i \in V(G)$, other than the root, let Y_i denote the set $\{toyleft(i)\} \cup \{topright(i)\}$. For the root r , let Y_r denote the set $\{r\}$. Then (G, Y) is a width-one graph decomposition of G' .

Proof (G', Y') has width one, because each Y_i contains at most two elements. Let u denote a vertex in G . If $u = r$, then $\{i \mid u \in Y_i\} = \{r\}$, a singleton set. Otherwise, $\{i \mid u \in Y_i\} = \{i \mid toyleft(i) = u\} \cup \{i \mid topright(i) = u\}$. Let v and w denote $lowest_leaf(u)$ and $highest_leaf(u)$, respectively. There are three possibilities: $u = toyleft(v)$ and $u = topright(w)$, $u = toyleft(v)$ but $u \neq topright(w)$, and $u \neq toyleft(v)$ but $u = topright(w)$. Depending on which of these three cases apply, $\{i \mid u \in Y_i\}$ equals $L_v \cup R_w$, L_v or R_w . Since L_v induces a path in T between v and u , and R_w induces a path in T between w and u , the intersection property holds in all cases.

To verify that the edge-covering property holds, observe that all edges in G' that

are also in G are covered. The remaining edges in G' have siblings in T as endpoints. Let ab denote one of these edges. Assume, without loss of generality, that $\text{highest_leaf}(a)$ has a lower number (modulo l) than $\text{lowest_leaf}(b)$. Let g and h denote $\text{highest_leaf}(a)$ and $\text{lowest_leaf}(b)$, respectively. Then $\text{topright}(g) = a$ and $\text{topleft}(h) = b$. From our construction of (G, Y) , it follows that $a \in Y_g$ that $b \in Y_h$. Moreover, since the edge ab exists in G' , g and h are consecutive in C , and hence, adjacent in G . Thus ab is covered. ■

Figure 2.17 shows the Halin decomposition of the outerplanar graph of Figure 2.15.

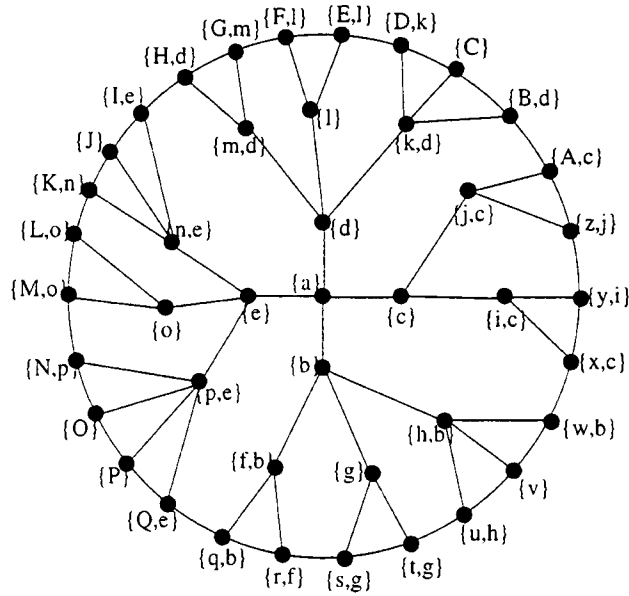


Figure 2.17: Halin decomposition of outerplanar graph.

Theorem 2.7 Let G denote a Halin graph of order n . Then a path decomposition of G of width at most $18 \times pw(G) + 17$ can be constructed in $O(n \log n)$ time.

Proof It follows from Observation 2.3 and Lemma 2.18 that **halin-gd** can be used to construct a width-two outerplanar graph decomposition (G', Y') of G . By Theorem 2.4, a path decomposition of G' of width at most $3 \times pw(G') + 2$ can be constructed.

By Theorem 2.6, this path decomposition can be converted into a path decomposition of G of width at most $(2 + 1) \times (3 \times pw(G') + 3) - 1 = 9 \times pw(G') + 8$. By Lemma 2.19 and Theorem 2.6, $pw(G') \leq 2 \times pw(G) + 1$. Thus the aforementioned path decomposition of G has width at most $9 \times (2 \times pw(G) + 1) + 8 = 18 \times pw(G) + 17$. The procedures **halin-gd**, **sp-td**, **td2pd** and **gd2pd** that are employed to construct the path decomposition all run in no more than $O(n \log n)$ time. ■

2.10 Experimental Results

We have developed pathwidth approximation algorithms for chordal, outerplanar, series-parallel and Halin graphs. Although upper bounds on the performance ratios of these algorithms were proved, there is no reason to suppose that these bounds are tight. So we conducted experiments with the purpose of discovering whether the bounds are approached in practice.

Unfortunately, it is difficult to measure the quality of the path decompositions produced, because there is no practical way to obtain optimal path decompositions for comparison. For outerplanar and series-parallel graphs, we solved this problem by testing our algorithms on pseudo-random graphs of known pathwidth. We were unable to handle chordal and Halin graphs using this method.

Our routine to generate pseudo-random outerplanar and series-parallel graphs employs a composition technique that is known to increase pathwidth by one. Some degree of randomness is obtained by changing a parameter associated with the composition. One drawback of this approach is the inability to generate graphs of a given order. Instead, for each of several values of pathwidth, we generated a large number of graphs and then selected those whose orders were within five percent of the desired number. Of course, there is nothing magical about the choice of five percent as the permissible variation; other small numbers give very similar results.

The approximation algorithms were implemented in the C programming language.

Table 2.1: Experimental Results on Series-Parallel Graphs

number of vertices	pathwidth		performance ratio	
	optimal	approximate	mean	worst case
50	4.1	5.0	1.2	1.8
100	4.3	5.8	1.3	2.3
200	4.5	6.6	1.5	2.0
500	4.9	7.7	1.6	2.0
1000	5.5	8.8	1.6	2.2
2000	5.8	9.3	1.7	2.2
5000	6.3	10.3	1.7	2.3
10000	6.6	11.1	1.7	2.3
20000	6.8	11.8	1.8	2.3

Table 2.2: Experimental Results on Outerplanar Graphs

number of vertices	pathwidth		performance ratio	
	optimal	approximate	mean	worst case
50	4.0	4.9	1.2	1.9
100	4.2	5.6	1.3	2.1
200	4.5	6.4	1.4	2.1
500	5.0	7.5	1.5	2.1
1000	5.3	8.6	1.6	2.1
2000	5.6	9.1	1.6	2.1
5000	6.2	10.2	1.6	2.2
10000	6.5	11.3	1.7	2.3
20000	6.8	11.9	1.7	2.3

Tests on a SPARC ULTRA indicate that the implementation is fast in practice, never taking more than a few seconds to compute path decompositions of graphs with several thousand vertices. The results of our experiments are shown in Tables 2.1 and 2.2. For each number shown in the first column, we tested our algorithms on a hundred graphs whose orders are within five percent of that number. The second and third columns list the mean values of the optimal pathwidth and the approximate pathwidth. The fourth and fifth columns show the mean and worst-case ratio of the approximate pathwidth to the optimal pathwidth. It is clear that the path

decompositions produced tend to have significantly smaller width than the worst-case guarantee. For all graphs tested, we obtained path decompositions with width less than three times the optimal pathwidth.

2.11 Discussion

All procedures used in our algorithms are linear, except for **td2pd**, the procedure that converts tree decompositions to path decompositions. The non-linearity of **td2pd** is due to the routine used to construct path decompositions of trees, which takes $(n \log n)$ time [EST]. This routine can actually determine pathwidth in linear time. We believe it can be improved to construct path decompositions in linear time as well. If this improvement is possible, then all our algorithms would work in linear time.

Our experiments on outerplanar and series-parallel graphs suggest that the worst-case performance ratios we derived for our algorithms may be loose upper bounds. For both classes, the worst ratio obtained experimentally is 2.3, whereas the guaranteed ratios are 3 for outerplanar graphs and 6 for series-parallel graphs. Thus the actual worst-case ratio for outerplanar graphs is in the range 2.3–3, and that for series-parallel graphs is in the range 2.3–6. A more careful analysis of the algorithms could yield tighter bounds.

Although **td2pd** and **gd2pd** can be applied to any tree or graph decomposition, provably good results are obtained only if the decompositions satisfy certain conditions. One requirement is that the decompositions have bounded width. The other is a good bound on the pathwidth of the underlying graph of the decompositions. We conjecture that any graph has an optimal tree decomposition whose underlying tree has pathwidth no more than the pathwidth of the graph itself. A constructive proof of this conjecture would provide a relative approximation algorithm for any class of graphs for which tree decompositions of bounded width can be found efficiently.

In addition to the graphs discussed in this chapter, this class includes all graphs of treewidth three or four, k -outerplanar graphs and graphs with disk-dimension k , for fixed k , and others.

We note that graph decompositions are potentially useful for purposes other than approximating pathwidth. For instance, a procedure to construct tree decompositions from graph decompositions can be obtained merely by replacing the word “path” in the first line of **gd2pd** with “tree.” The proof of correctness is essentially identical to that of Theorem 2.6. If the input graph decomposition and its tree decomposition both have bounded width, then so does the tree decomposition constructed by **gd2pd**. We believe such an approach can be used to build tree decompositions of width $3k - 1$ for k -outerplanar graphs, by decomposing them into $k - 1$ -outerplanar graphs. Graph decompositions may also be of use in finding vertex separators, because separators in any decomposition of a graph G correspond to separators in G .

Chapter 3

The Complexity of Partitioning

In this chapter, we study some new variants of graph partitioning. Let G denote a graph, some of whose vertices may be distinguished. A *partition* of G is a collection of disjoint subsets of $V(G)$ whose union is $V(G)$, such that each subset contains at most one distinguished vertex. Our definition generalizes the usual one, which applies only to graphs with no distinguished vertices. For the graph partitioning problem, the question to be answered is whether G has a partition with certain specified properties.

We analyze the complexity of problem variants arising from different desirable properties. The properties we consider are motivated by the comparatively new style of circuit design using multiple field-programmable gate arrays (FPGAs). FPGAs are commonly used for emulating complex designs and are now replacing application specific chips in some domains. A valid layout of a circuit on multiple FPGAs corresponds to a partition of its graph into subgraphs that satisfy size and degree constraints. While partitioning has been studied extensively, variants that involve degree constraints have been ignored until recently.

All the problems we examine turn out to be $\mathcal{NP}$ -hard, with one exception. For this singular case, we present a polynomial-time algorithm. For the others, we analyze the complexity when restrictions are placed on input graphs and when parameter values are fixed. We find that although polynomial-time algorithms are possible

in some instances, the problems stay $\mathcal{NP}$ -hard in many cases. Interestingly, the fixed-parameter versions of our problems are amenable to techniques that exploit the “immersion order.” Before proceeding, we pause to make a brief foray into this order.

3.1 The Immersion Order

Let e_1 , with endpoints u and v , and e_2 , with endpoints v and w , where $u \neq v \neq w$, denote a pair of edges. Then e_1 and e_2 are *lifted* by deleting e_1 and e_2 , and adding an edge e_3 between u and w . A graph H is said to be *immersed* in G , denoted $H \leq_i G$, if a graph isomorphic to H can be obtained from G by taking subgraphs and lifting pairs of edges. A family $\mathcal{F}$ of graphs is said to be *closed under immersion* (or *immersion-closed*) if $G \in \mathcal{F}$ and $H \leq_i G$ implies that $H \in \mathcal{F}$.

Let $\leq$ denote a reflexive and transitive binary relation defined on the elements of a set Q . A *descending chain* is a sequence of distinct elements $q_0, q_1, q_2, \dots$ of Q , where $q_i \geq q_{i+1}$ for $i \geq 0$. An *antichain* is a set $A \subseteq Q$ such that if p and q are distinct elements of A , then $p \not\leq q$ and $q \not\leq p$. Q is said to be *well-quasi-ordered* by $\leq$ if all descending chains and antichains are finite. Robertson and Seymour proved that the $\leq_i$ relation well-quasi-orders the set of finite undirected graphs [RS23]. This result settled a long-standing conjecture due to Nash-Williams [Nas].

Consider a family of graphs $\mathcal{F}$ that is closed in the immersion order. $\mathcal{F}$ can be completely characterized by listing the minimal elements in its complement $\overline{\mathcal{F}}$. The minimal elements of $\overline{\mathcal{F}}$ are called *obstructions* to $\mathcal{F}$. Interestingly, the set of obstructions to $\mathcal{F}$ must be finite, or else it would constitute an infinite antichain, contradicting the result of [RS23]. The finiteness of the set of obstructions suggests the following algorithm to determine membership in $\mathcal{F}$.

- Enumerate all elements in the set of obstructions to $\mathcal{F}$.
- For each element H in the obstruction set, check if H is immersed in the input graph G .

- If $H \leq_i G$ for some H in the obstruction set, then $G \notin \mathcal{F}$, else $G \in \mathcal{F}$.

Note that this algorithm is guaranteed to terminate because the obstruction set is finite. A key step in the algorithm is the check to determine whether $H \leq_i G$, where H is an element of the obstruction set. A naive implementation of this step that tries all possible sequences of lifts and subgraph operations would be highly superpolynomial, making this approach infeasible. Fortunately, when H is a fixed graph, it is possible to do far better.

Theorem 3.1 [FL5] For every fixed graph H , there exists a polynomial-time algorithm that, given a graph G , decides whether $H \leq_i G$.

Difficulties still remain in making the membership-testing algorithm practical. For one thing, the algorithms promised by Theorem 3.1 have huge multiplicative constants and degrees, and are far from practical. Second, enumerating the obstruction set to a particular family is a non-trivial task, which is compounded by the fact that the Robertson-Seymour theorems are not constructive.

3.2 A Fundamental Problem

We now begin our exploration of partition problems with degree constraints. The most basic problem we study was proposed in [Lan], and is known as the Minimum Degree Graph Partition problem.

Minimum Degree Graph Partition (MDGP)

Input: A graph G , and integers d and k .

Output: YES, if G has a partition into subsets of degree at most d and size at most k and NO, otherwise.

An equivalent restatement of MDGP is to ask if there is a set of edges in G whose deletion leaves disjoint subsets of $V(G)$, of size at most k and degree at most d .

MDGP can thus be viewed either as a partitioning problem, or as a cutting problem. This duality holds for all the problems we discuss. Traditionally, a problem has been viewed as a cutting problem if the focus is on the set of edges that is cut, and as a partitioning problem if the focus is on the subsets obtained.

Theorem 3.2 MDGP is $\mathcal{NP}$ -hard.

Proof The reduction is from Graph Bisection, an $\mathcal{NP}$ -complete restriction of Minimum Cut into Bounded Sets [GJ]. The input to Graph Bisection consists of a graph G and an integer d . The problem is to determine if G has a bisection of degree at most d ; that is, does G have a partition into two subsets whose sizes differ by at most one, such that the degree of each subset is at most d ? Note that Graph Bisection stays $\mathcal{NP}$ -complete when restricted to graphs of even order, because isolated vertices can be added or deleted without changing “yes” instances to “no” instances or vice versa.

Consider a graph G of even order. Let $|V(G)| = 2n$ and $|E(G)| = m$. Construct a graph G' by choosing an arbitrary vertex v from G and adding $m+1$ edges between v and every other vertex in G . We claim that G is a “yes” instance of Graph Bisection(d) if and only if G' is a “yes” instance of MDGP($(m+1)n+d, n$). Suppose G has a bisection of degree at most d . Then, since any bisection of G' has degree $(m+1)n$ more than the same bisection of G , G' is a “yes” instance of Graph Bisection($(m+1)n+d$), and hence, a “yes” instance of MDGP($(m+1)n+d, n$). To prove the converse, suppose G' has a partition into subsets of size at most n and degree at most $(m+1)n+d$. In such a partition, let S_j denote the subset that contains v . At most n vertices can be excluded from S_j , or else $\delta(S_j) \geq (m+1)(n+1) > (m+1)n+d$, a contradiction. Thus S_j contains exactly n vertices. By combining all the subsets besides S_j , we obtain a bisection of G' with degree $(m+1)n+d$. Since the same bisection of G has degree d , G is a “yes” instance of Graph Bisection(d). To complete the proof, we note that the construction of G' can be accomplished in polynomial time. ■

Theorem 3.3 For any fixed d and k , the decision and search versions of MDGP(d, k) can be solved in polynomial time.

Proof We first observe that for fixed d and k , the family of “yes” instances is closed in the immersion order. For any graph G , neither the subgraph operation nor lifting a pair of edges increases the degree of any subset of $V(G)$. Thus if G has a partition into subsets of size at most k and degree at most d , then so does any graph immersed in G . The immersion-closure of MDGP implies that when d and k are fixed, a polynomial-time algorithm exists for the decision problem [FL5].

The search version can be solved using a simple self-reduction scheme [LP]. We iteratively invoke the decision algorithm, adding $d + 1$ edges between a different pair of vertices in G each time. Depending on whether G stays a “yes” instance or not, the edges are either retained or deleted. When this process terminates, vertices that belong to the same subset will be connected to each other by $d + 1$ edges. The subsets can then be easily determined. ■

Note that the algorithms guaranteed by Theorem 3.3 are not constructively known, and may have very high degrees. It turns out that constructive linear-time algorithms exist, exploiting the following condition.

Locality Condition [Pla] G is a “yes” instance of MDGP(d, k) if and only if $\forall v \in V(G)$, there is a connected subset of size at most k and degree at most d that contains v .

Now we examine the case in which only one of d and k is fixed. Observe that in a graph of order n and size m , there are $O(n^k)$ subsets of size k or less, and $O(m^d) = O(n^{2d})$ connected subsets of degree d or less. Thus, for each vertex, checking for the existence of a connected subset of size at most k and degree at most d containing that vertex can be done in $O(n^{\min\{k, 2d\}})$ time. It follows that deciding whether a graph is a “yes” instance takes polynomial time if either d or k is fixed. The fact that MDGP is polynomial-time solvable for fixed d has the interesting implication that it

is solvable in polynomial time on graphs of bounded degree. This is because d need never be greater than the maximum degree in the input graph.

3.3 Adding One More Parameter

MDGP finds applications in circuit design and related fields. Consider, for example, the problem of partitioning a circuit (graph) into a number of identical FPGAs. The size of an FPGA is the number of logic blocks (vertices) it contains. The degree of an FPGA is its pin-count, the maximum number of wires that can connect the part of the circuit inside the FPGA to the part outside. The parameters k and d of MDGP reflect these two constraints. MDGP is limited, however, in that it only asks if a circuit has a valid partition; in reality, one usually wants to find a partition that uses as few FPGAs as possible [CLCDL, KBK]. Hence we consider the following combinatorial problem.

FPGA Minimization

Input: A graph G , and integers d , k and p .

Output: YES, if G has a partition into p or fewer subsets of degree at most d and size at most k , and NO, otherwise.

It might seem that by modeling circuits using graphs, we are ignoring external signals, such as primary inputs and outputs. For example, in our model, any graph G has a valid partition as long as $k \geq |V(G)|$, even if $d = 0$ and $p = 1$. This implies that the corresponding circuit can be packed into an FPGA that has no I/O pins, which is not true in the presence of external signals. This wrinkle is easily handled by first mapping each external signal to a path of length k , in which adjacent vertices are connected by $d + 1$ edges (the idea is to force all the vertices in the path into a single subset). The rest of the circuit is mapped as usual, with nodes being mapped to vertices and nets to edges. The parameter p is incremented by the number of external signals. The parameters d and k are left unchanged.

3.3.1 Complexity

FPGA Minimization is $\mathcal{NP}$ -hard in general, because it contains MDGP as a special case (we merely set the parameter p to $|V(G)|$). Seven restrictions of the general problem may be obtained by fixing one or more of the three parameters. Fixing a single parameter, whether it be d , k or p , still leaves the problem $\mathcal{NP}$ -hard. In fact, even if d and k are both fixed, FPGA minimization remains $\mathcal{NP}$ -hard. If d and p are fixed, a polynomial-time algorithm may be derived from the observation that the total number of edges between all the subsets in a valid partition is bounded by $\frac{dp}{2}$, a constant. If k and p are fixed, any “yes” instance can have at most kp vertices, which is a constant. Such an instance can, in principle, be recognized in constant time by table look-up. When all three parameters are fixed, there is a polynomial-time algorithm due to immersion closure. Of course the polynomial-time algorithms for the previous two cases apply when all parameters are fixed as well. Note that these brute-force approaches are impractical for all but very small parameter values.

Table 3.1 lists the complexity of all of these variants. Only the results for the decision version are shown, since each problem in the table can be self-reduced using the technique mentioned in the proof of Theorem 3.3. Thus the complexity of the search problem is within a linear factor of the complexity of the corresponding decision problem.

We have already provided, at the beginning of this section, justification for the entries in the first and last three rows. Since FPGA Minimization with $p = 2$ and $k = \lceil \frac{|V(G)|}{2} \rceil$ is identical to Graph Bisection, the entry in row 4 is valid. The entry in row 3 is subsumed by that in row 5. This leaves two entries (row 2, corresponding to fixed d , and row 5, corresponding to fixed d and k) that require justification.

Fixed d

Theorem 3.4 FPGA Minimization is $\mathcal{NP}$ -hard for any fixed $d \geq 0$.

Proof The reduction is from 3-Partition. An instance of this problem consists of an

Table 3.1: Complexity of FPGA Minimization

parameter(s) fixed	complexity
none	$\mathcal{NP}$ -complete
d	$\mathcal{NP}$ -complete for $d \geq 0$
k	$\mathcal{NP}$ -complete for $k \geq 3$
p	$\mathcal{NP}$ -complete for $p \geq 2$
d and k	$\mathcal{NP}$ -complete for $k \geq 3$ and $d \geq 12$
d and p	In $\mathcal{P}$
k and p	In $\mathcal{P}$
d, k and p	In $\mathcal{P}$

integral bound B and $3m$ elements $a_1, \dots, a_{3m}$, each of size between $\frac{B}{4}$ and $\frac{B}{2}$, such that $\sum_{i=1}^{3m} size(a_i) = mB$. The question to be answered is whether there is a 3-partition of the input elements; that is, can the elements be gathered into m disjoint sets such that the sum of the sizes of the elements in each set is m ? The problem derives its name from the fact all sets in a solution must have cardinality three.

We assume that the bound B is polynomial in m ; 3-Partition remains $\mathcal{NP}$ -complete with this restriction [GJ]. We map an instance of 3-Partition to an instance of FPGA Minimization by mapping each element a_i to a clique C_i of order $size(a_i)$. The resulting graph G consists of $3m$ connected components, each of which is a clique. Any solution to the 3-Partition instance can be used to construct a partition of G into subsets of size B and degree 0, by replacing each element a_i with $V(C_i)$. Conversely, any partition of G into m or fewer subsets of size at most B and degree 0, must consist of exactly m subsets, each of size exactly B . Moreover, each subset must consist of all the vertices from some three cliques. We can construct a satisfactory 3-partition by placing a_i and a_j in the same set if and only if $V(C_i)$ and $V(C_j)$ are in the same subset in G 's partition. Thus the instance of 3-Partition is a "yes" instance if and

only if G is a “yes” instance of FPGA Minimization($0, B, m$). To complete the proof, we note that G can be constructed in time $O(mB^2)$, which is polynomial in m . ■

Fixed d and k

We show that an $\mathcal{NP}$ -complete restriction of Partition into Triangles reduces to this variant. An instance of Partition into Triangles consists of a graph whose order is a multiple of three. The problem is to determine whether the graph has a partition into subsets of size three, such that the vertices in each subset induce a triangle. Partition into Triangles can be proven $\mathcal{NP}$ -complete by means of a reduction from Exact Cover By 3-Sets (X3C) [GJ]. An instance of X3C consists of a finite set X with $|X| = 3n$, and a collection C of 3-element subsets of X . X3C is known to be $\mathcal{NP}$ -complete even if each element in X is present in at most three subsets in C [GJ]. This fact implies that Partition into Triangles is $\mathcal{NP}$ -complete on simple graphs of maximum degree six or less (see the reduction on pages 68–69 of [GJ]).

Accordingly, let G denote a simple graph of order $3n$ and maximum degree at most six. Let $V = v_1, \dots, v_{3n}$ denote the vertices in G . We construct a supergraph G' of G as follows: Set $V(G')$ to $V \cup U$, where $U = \{u_1, \dots, u_{3n}\}$. For $1 \leq i \leq n$, add 18 edges each between u_{3i} and u_{3i-1} , u_{3i} and u_{3i-2} , and u_{3i-1} and u_{3i-2} . For $1 \leq i \leq 3n$, add $6 - \delta_G(v)$ edges between v_i and u_i .

Lemma 3.1 G is a “yes” instance of Partition into Triangles if and only if G' is a “yes” instance of FPGA Minimization($3, 12, 2n$).

Proof Suppose G has a partition into triangles. Then the subsets of this partition, along with n subsets of the form $\{u_{3i}, u_{3i-1}, u_{3i-2}\}$ for $1 \leq i \leq n$, comprise a partition of G' into $2n$ subsets of size at most three and degree at most 12. Thus G' is a “yes” instance of FPGA Minimization($3, 12, 2n$).

Conversely, if G' is a “yes” instance of FPGA Minimization($3, 12, 2n$), then, for $1 \leq i \leq n$, u_{3i} , u_{3i-1} and u_{3i-2} must belong to the same subset in any valid partition.

Thus n of the subsets in the partition contain only vertices from U and the remaining n contain only vertices from V . Consider an arbitrary set S from among the latter n . Since the subgraph of G induced by V is simple, and since $\delta(v_i) = 6$ for each v_i , $\delta(S) \leq 12$ if and only if the three vertices in S induce a triangle in G . Therefore, the n subsets that consist of vertices from V form a partition into triangles of G . ■

Since the construction of G' can be accomplished in polynomial time, we obtain the following result.

Theorem 3.5 FPGA Minimization is $\mathcal{NP}$ -hard for any fixed $d \geq 12$ and $k \geq 3$.

3.3.2 Hard Instances

No practical exact algorithm is known for any of the fixed-parameter variants of FPGA Minimization cataloged in Table 3.1. Of all of these variants, the one in the fifth row (fixed d and k , variable p) comes closest to practice. This is because in a given instance of a partitioning problem, the size and pin-count of the available FPGA are fixed, whereas the number of FPGA chips is usually variable. Since this variant is $\mathcal{NP}$ -complete, it is unlikely that a polynomial-time algorithm exists.

Practitioners have turned to heuristics as a viable alternative [CLCDL, KBK, WK]. Heuristics for FPGA Minimization typically have no performance guarantees and moreover, fail badly on certain contrived graphs. An example of a class of hard instances is the family of graphs that we humorously dub “mine graphs.” For $k \geq 4$ and $m \geq 1$, the mine graph consists of a central clique of order k , with a pendant vertex attached to each vertex in the clique. All edges have multiplicity m . Figure 3.1 depicts mine graphs for $k = 5, m = 1$ and $k = 12, m = 1$.

It is easy to see that a mine graph with a central clique of order k and edges of multiplicity m is a “yes” instance of $\text{FPGA Minimization}(km, k, 2)$. A satisfactory partition consists of two subsets, one containing all the vertices in the central clique, and the other containing all the pendant vertices. What makes the mine graph a hard

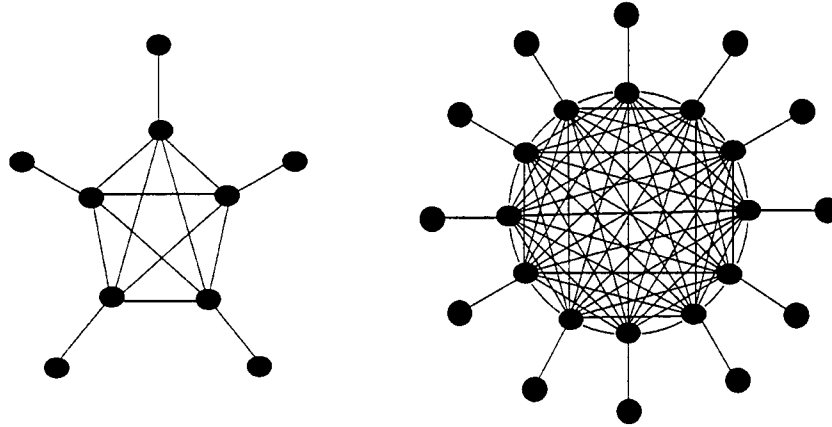


Figure 3.1: Examples of mine graphs.

instance is that any partition in which the clique vertices are split among subsets must either be comprised of at least k subsets (each containing one vertex from the clique) or have degree more than km . So a heuristic that attempts to minimize the number of subsets while satisfying the size and degree constraints must either find the optimal partition or produce a solution that is worse than optimal by at least a factor of $\frac{k}{2}$. Known heuristics that build a partition incrementally are not likely to be optimal, because they would tend to include a pendant vertex and its adjacent vertex in the clique in the same subset. The pendant vertices thus function as “detonators” that fracture the central clique, preventing optimality. Heuristics based on the Kernighan-Lin algorithm [KeL] will do no better, because any sequence of moves that lead from a random initial partition to an optimal partition must generate intermediate partitions that grossly violate the degree constraint.

Of course mine graphs are artificial, since graphs containing big cliques and a large number of pendant vertices are unlikely to occur in practice. A more realistic family of graphs that foil heuristic approaches can be obtained by replacing the central clique in mine graphs by a grid of appropriate size and the pendant vertices by sparse subgraphs.

3.4 Related Cutting Problems

In this section we study some related cutting problems. Recall that any cutting problem can also be viewed as a partitioning problem. For the sake of notational convenience, we now present Multiway Cut in the latter fashion.

Multiway Cut

Input: A graph G , some of whose vertices are distinguished, and an integer d .

Output: YES, if G has a partition in which the total number of edges between subsets is at most d , and NO, otherwise.

Multiway Cut is a well-studied problem. It is $\mathcal{NP}$ -hard, but approximable in polynomial time to within a factor of two of the optimum [DJPSY]. When the parameter d is fixed, it is closed under the immersion order, and hence, solvable in polynomial time. There is an $O(n^{2d+1})$ algorithm for Multiway Cut that examines each of the $O(n^{2d})$ sets of d or fewer edges in the input graph and checks if one of these sets induces a satisfactory partition. When d is fixed, this algorithm runs in polynomial time. A linear-time algorithm, such as exists for MDGP, is not known for this fixed-parameter variant.

Minimum Degree Cut

Input: A graph G , some of whose vertices are distinguished, and an integer d .

Output: YES, if G has a partition into subsets of degree at most d , and NO, otherwise.

Minimum Degree Cut differs from Multiway Cut in that the parameter d now represents a local degree constraint rather than a global constraint. This problem can be cast as a restriction of MDGP, by replacing distinguished vertices with dense subgraphs and setting the parameter k to a sufficiently high value. Unlike the other problems we discuss, Minimum Degree Cut has provably polynomial complexity. We

next present a procedure, **mdcsolve**, that solves both the decision and search versions of the problem in polynomial time.

Procedure **mdcsolve** assumes the existence of a procedure, **find-min-cut**, to find the smallest set of edges that separate a given vertex from a set of other vertices. It follows from the well-known “max-flow min-cut” theorem [FF] that this can be accomplished using an algorithm for network flow. For instance, suppose we want to separate vertex v from the set of vertices U . We merely add a new vertex v' to the graph, add edges of infinite capacity between each vertex in U and v' , and invoke a network flow algorithm to find the minimum cut between v and v' . An array S is used in **mdcsolve** to maintain candidate subsets. Each entry in S is assumed to be empty initially.

Procedure mdcsolve

```

Input:   A graph  $G$ , some of whose vertices are distinguished, and an integer  $d$ .
Output: A partition  $P$  of  $G$  into subsets of degree at most  $d$ , if possible.
begin procedure
   $L :=$  the set of distinguished vertices in  $G$ ;
  for each vertex  $v \in L$ 
    do begin
       $E := \text{find-min-cut}(v, L - \{v\})$ ;
      if  $|E| \leq d$  then  $S[v] :=$  the vertices in the connected
                                component of  $G - E$  containing  $v$ ;
      for each vertex  $u \in V(G) - L$ 
        do begin
           $E := \text{find-min-cut}(u, L - \{v\})$ ;
          if  $|E| \leq d$  then  $S[u] :=$  the vertices in the connected
                                component of  $G - E$  containing  $u$ ;
        end
      end
  if  $\exists v \in V(G)$  such that  $S[v] = \emptyset$ 
    then output NO and quit;
  else for each pair of vertices  $u$  and  $v$  in  $G$ 
    do if the number of edges between  $S[u] \cap S[v]$  and  $S[u] - S[v]$  is smaller
          than the number of edges between  $S[u] \cap S[v]$  and  $S[v] - S[u]$ 
      then  $S[u] := S[u] - S[v]$ ;
      else  $S[v] := S[v] - S[u]$ ;
  output  $\{S[i] \mid S[i] \neq \emptyset\}$ ;
end procedure

```

The proof of correctness of **mdcsolve** uses the following technical lemma whose statement and proof are similar to those of the Locality Condition. For completeness, we include a proof here. Let v denote a vertex and S a set of vertices, in a graph G , some of whose vertices may be distinguished. For $d \geq 0$, we say that S is a *d-candidate subset* for v if S contains at most one distinguished vertex, $v \in S$ and $\delta(S) \leq d$.

Lemma 3.2 A graph G is a “yes” instance of Minimum Degree Cut(d) if and only if $\forall v \in V(G)$, there exists at least one d -candidate subset for v .

Proof Clearly, if there is a vertex for which a d -candidate subset does not exist, then G cannot be a “yes” instance. If a d -candidate subset exists for every vertex, then there is a collection of subsets (not necessarily disjoint), each of which contains at most one distinguished vertex and has degree at most d , whose union equals $V(G)$. Let C denote such a collection of subsets. We show that it is possible to convert C into a partition by deleting vertices from subsets without increasing the degree of any subset. This proves the lemma.

If all subsets in C are disjoint, then C is already a partition. So let S_1 and S_2 denote a pair of subsets with a non-empty intersection. Let m_1 denote the number of edges between $S_1 \cap S_2$ and $S_1 - S_2$, and m_2 the number of edges between $S_1 \cap S_2$ and $S_2 - S_1$. Since $S_1 - S_2$ and $S_2 - S_1$ are disjoint, $\delta(S_1 \cap S_2) \geq m_1 + m_2$. Assume, without loss of generality, that $m_2 \geq m_1$. Then $\delta(S_1) = \delta(S_1 - S_2) + \delta(S_1 \cap S_2) - 2m_1 \geq \delta(S_1 - S_2) + m_2 - m_1 \geq \delta(S_1 - S_2)$. Thus we may replace S_1 with $S_1 - S_2$ without increasing the degree or the number of distinguished vertices of any subset. By repeatedly eliminating common vertices until all subsets are disjoint, we obtain a partition. ■

Theorem 3.6 Let G denote a graph of order n , and let d denote a non-negative integer. Then the call to **mdcsolve**(G, d) correctly decides in $O(n^5)$ time whether

G is a “yes” instance of Minimum Degree Cut(d). If G is a “yes” instance, then a partition of G into subsets of degree at most d is returned, also in $O(n^5)$ time.

Proof Procedure **mdcsolve** has two distinct phases. In the first phase, the algorithm decides whether G is a “yes” instance, by attempting to find a d -candidate subset for each vertex. Failure is reported if and only if there is a vertex for which there is no such subset. This action is justified by Lemma 3.2. The complexity of finding d -candidate subsets is determined by the amount of time spent in **find-min-cut**. As mentioned before, **find-min-cut** can be implemented using a network flow algorithm, and therefore, runs in $O(n^3)$ time. Since **find-min-cut** is invoked $O(n^2)$ times, the first phase of **mdcsolve** takes $O(n^5)$ time.

Next, a partition of G is constructed by eliminating vertices from d -candidate subsets. The correctness of the procedure used for eliminating common vertices follows from the proof of Lemma 3.2. The complexity of this phase is dominated by the set operations (intersection and set difference) performed on the various d -candidate subsets. Since each subset contains at most n vertices, each operation takes $O(n)$ time. The number of iterations of the second loop is at most n^2 . Therefore, the total time spent in the second phase is $O(n^3)$. Overall, **mdcsolve** runs in $O(n^5)$ time. ■

Observe that although the definition of Multiway Cut imposes no constraints on the number of subsets, a valid partition need never contain a subset without a distinguished vertex in it. This is because any subset without a distinguished vertex can be absorbed into another without changing a “yes” instance to a “no” instance. So, in effect, the number of subsets is bounded by the number of distinguished vertices in the input graph. This is not true of Minimum Degree Cut: an optimal partition may need to contain arbitrarily many subsets with no distinguished vertices in them. Thus we are naturally led to the following version of Minimum Degree Cut, where there is an explicit bound on this number.

Minimum Degree Cut with Subset Bound (MDCSB)

Input: A graph G , some of whose vertices are distinguished, and integers d and p .

Output: YES, if G has a partition into p or fewer subsets of degree at most d ,
and NO, otherwise.

Unfortunately, as we shall show, the addition of parameter p transforms Minimum Degree Cut into an $\mathcal{NP}$ -hard problem. Let G denote a graph with p or fewer distinguished vertices. Then $mwc(G)$ denotes the minimum, over all partitions of G , of the number of edges between subsets, and $mdcsb(G, p)$ denotes the minimum, over all partitions of G into p or fewer subsets, of the maximum degree of a subset. Note that $mwc(G) \leq \frac{p}{2} \times mdcsb(G, p)$. This is because, in any partition of G , the total degree of the subsets, which equals twice the number of edges between subsets, is no more than the number of subsets times their maximum degree.

Lemma 3.3 Let G be a graph with three distinguished vertices and m edges. There is a supergraph G' of G , with the same vertex set as G and at most $4m$ edges, such that $mwc(G') = \frac{3}{2} \times mdcsb(G', 3)$.

Proof Let $\{S_1, S_2, S_3\}$ denote a partition of G that is optimal with respect to Multiway Cut; that is, the total number of edges between subsets of the partition equals $mwc(G)$. For $1 \leq i < j \leq 3$, add edges between the distinguished vertices in S_i and S_j until there are m edges between S_i and S_j . Let G' denote the resulting graph. Then $|E(G')| = |E(G)| + 3m - mwc(G) \leq 4m$. Note that each edge in $E(G') - E(G)$ has endpoints in different subsets in any partition of G' , and must, therefore, contribute to the number of edges between subsets. Thus $mwc(G') \geq mwc(G) + |E(G') - E(G)| = 3m$. On the other hand, since $\delta(S_1) = \delta(S_2) = \delta(S_3) = 2m$ in G' , $mdcsb(G', 3) \leq 2m$, implying that $mwc(G') \geq \frac{3}{2} \times mdcsb(G', 3)$. Since we know that $mwc(G') \leq \frac{3}{2} \times mdcsb(G', 3)$, the lemma follows. ■

Theorem 3.7 MDCSB is $\mathcal{NP}$ -hard.

Proof We exhibit a polynomial-time Turing reduction from Multiway Cut to MDCSB. The reduction is restricted to graphs with only three distinguished vertices. This suffices, since Multiway Cut is known to be $\mathcal{NP}$ -hard on such graphs [DJPSY].

Let G be a graph with three distinguished vertices and m edges. Let $\mathcal{B}(G)$ denote the set of all graphs generated by adding m or fewer edges between each pair of distinguished vertices in G . For $1 \leq i \leq |\mathcal{B}(G)|$, let G_i denote the i^{th} graph in $\mathcal{B}(G)$ and let e_i denote $|E(G_i) - E(G)|$. Then $\text{mwc}(G_i) = \text{mwc}(G) + e_i$, because, in any partition, an edge between distinguished vertices contributes a count of one to the number of edges between subsets. Since $\text{mwc}(G_i) \leq \frac{3}{2} \times \text{mdcsb}(G_i, 3)$, we have $\text{mwc}(G) \leq \frac{3}{2} \times \text{mdcsb}(G_i, 3) - e_i$. Note that the graph G' defined in Lemma 3.3 is an element of $\mathcal{B}(G)$. So there is some $k \leq |\mathcal{B}(G)|$, such that $G_k = G'$. Since $\text{mwc}(G') = \frac{3}{2} \times \text{mdcsb}(G', 3)$, $\text{mwc}(G) = \frac{3}{2} \times \text{mdcsb}(G_k, 3) - e_k$.

This observation implies that, for any d , $\text{mwc}(G) \leq d$ if and only if there is a graph $G_k \in \mathcal{B}(G)$ such that $\frac{3}{2} \times \text{mdcsb}(G_k, 3) - e_k \leq d$. Rearranging the terms in the latter inequality, we find that $\text{mwc}(G) \leq d$ if and only if there exists G_k such that $\text{mdcsb}(G_k, 3) \leq \frac{2(d+e_k)}{3}$. Thus we may determine whether $\text{mwc}(G) \leq d$ by making $|\mathcal{B}(G)|$ calls to a decision algorithm for MDCSB. To complete the proof, we show that $|\mathcal{B}(G)|$ is polynomial in m . Each graph in $\mathcal{B}(G)$ can be uniquely characterized by a three-tuple, each element of which denotes the number of edges added between a pair of distinguished vertices in G to obtain the graph (there are three ways of choosing a pair from the three distinguished vertices in G , hence the three-tuple). The value of each element in the three-tuple can range from 0 to m . Thus the total number of three-tuples is $(m+1)^3$, a polynomial function of m . ■

The above proof applies, *mutatis mutandis*, if the number of distinguished vertices in G is any fixed number greater than three. MDCSB stays $\mathcal{NP}$ -hard even on graphs in which all non-distinguished vertices have degree at most three. This is because Multiway Cut is $\mathcal{NP}$ -hard on such graphs, and because the transformation described

in our proof does not change the degree of any non-distinguished vertex.

When the parameters d and p are fixed, MDCSB is closed in the immersion order, and therefore, solvable in polynomial time. No direct polynomial-time method is known.

3.5 Refining the Complexity

Graphs that represent circuits often have bounded degree due to restrictions on fan-in/out. Recall that MDGP can be solved in polynomial time on such graphs. FPGA Minimization, on the other hand, remains $\mathcal{NP}$ -hard even on graphs of maximum degree four, as we prove here. We employ a reduction from MDCSB. To transform an instance of MDCSB to an instance of FPGA Minimization, we replace distinguished vertices with subgraphs isomorphic to grids. Before describing the transformation, we define grid graphs and establish some results relating the size and degree of vertex sets in these graphs.

The $m \times n$ *grid graph* is the simple graph G with $V(G) = \{(0, 0), (0, 1), \dots, (m - 1, n - 1)\}$ and $E(G) = \{\{(i, j), (k, l)\} \mid (i = k \text{ and } |j - l| = 1) \text{ or } (j = l \text{ and } |i - k| = 1)\}$. The vertices and edges of G can be visualized as the points and line segments of a grid. *Rows* and *columns* in G are sets of vertices with the same y and x coordinates, respectively. If $A \subseteq G$, then $\bar{A}$ denotes the set $V(G) - A$.

Since subgraphs induced by rows and columns are connected, if a row or column contains vertices from both A and $\bar{A}$, then it contains an edge between A and $\bar{A}$. This implies that the degree of A is no less than the number of rows and columns that contain vertices from both A and $\bar{A}$. The *minimal super-rectangle* of A is the smallest rectangle R with sides parallel to the axes such that all vertices in A lie either inside R or on R 's boundary. If A is a subset of a row or a column, then R will actually be one-dimensional. Note that each side of R must contain a vertex from A . If the subgraph induced by A is connected, then there must exist paths consisting entirely

of vertices in A between both pairs of opposite sides of R . Any row or column that intersects R intersects one of these paths, and hence, contains a vertex from A .

Lemma 3.4 Let A denote a set of vertices in a grid graph G , such that the subgraph induced by A is connected. Let R be the minimal super-rectangle of A , and let x and y denote the dimensions of R . If there is a row and a column in G that do not intersect R , then $\delta(A) \geq x + y$.

Proof Let X_r and X_c denote a row and a column, respectively, that do not intersect R . Then $X_r \subseteq \bar{A}$ and $X_c \subseteq \bar{A}$. Consider the $x + y$ rows and columns that intersect R . They all contain at least one vertex from A . Since every row in G intersects X_c and every column in G intersects X_r , they all contain at least one vertex from $\bar{A}$ as well. The lemma follows since $x + y$ rows and columns contain vertices from both A and $\bar{A}$. ■

Lemma 3.5 Let A be a set of $\frac{n^2}{2}$ or fewer vertices that induces a connected subgraph of an $n \times n$ grid graph G . Let R denote the minimal super-rectangle of A . If all rows or all columns in G intersect R , then $\delta(A) \geq n$.

Proof Suppose there a row X_r in G that does not intersect R . Then all columns in G intersect both R and X_r . Since $X_r \subseteq \bar{A}$, all n columns contain vertices from both A and $\bar{A}$, and the lemma follows. Similarly, the lemma holds if there is a column in G that does not intersect R . So assume that every row and every column intersects R . Then all rows and columns contain at least one vertex from A . Since $|A| \leq \frac{n^2}{2}$, at least $\frac{n}{2}$ rows and $\frac{n}{2}$ columns must also contain vertices from $\bar{A}$. Thus $\delta(A) \geq \frac{n}{2} + \frac{n}{2} = n$. ■

Lemma 3.6 Let A be a set of vertices in an $n \times n$ grid graph G . Then $\delta(A) \geq \text{minimum}(\sqrt{2|A|}, \sqrt{2n^2 - 2|A|})$.

Proof Assume that $|A| \leq \frac{n^2}{2}$; if not, we can work with $\bar{A}$. Since $|A| \leq n^2 - |A|$, we need only show that $\delta(A) \geq \sqrt{2|A|}$. The proof proceeds by induction on the

number of connected components of G_A , the subgraph induced by A . Let R denote the minimal super-rectangle of A . Let x and y denote the dimensions of R .

In the basis case G_A is a connected graph. If R intersects all rows or all columns in G , then, by Lemma 3.5, $\delta(A) \geq n \geq \sqrt{2|A|}$. Otherwise, by Lemma 3.4, $\delta(A) \geq x+y \geq 2\sqrt{xy} \geq 2\sqrt{|A|} > \sqrt{2|A|}$. If G_A is not connected, let A_1 denote the set of vertices in a connected component of G_A . By the induction hypothesis, $\delta(A_1) \geq \sqrt{2|A_1|}$ and $\delta(A - A_1) \geq \sqrt{2|A - A_1|}$. So $\delta(A) = \delta(A_1) + \delta(A - A_1) \geq \sqrt{2|A_1|} + \sqrt{2|A - A_1|} \geq \sqrt{2|A_1| + 2|A - A_1|} = \sqrt{2|A|}$. ■

Lemma 3.7 Let A be a set of $\frac{n^2}{2}$ or fewer vertices in an $n \times n$ grid graph G . Let X denote a row or a column in G . Then $|X \cap A| \leq \delta(A)$.

Proof The proof is analogous to that of Lemma 3.6, and uses induction on the number of connected components of G_A , the subgraph induced by A . Let R , with dimensions x and y , denote the minimal super-rectangle of A . Note that $|X \cap A| \leq n$ and $|X \cap A| \leq \text{maximum}\{x, y\}$.

In the basis case G_A is a connected graph. If R intersects all rows or all columns in G , then, by Lemma 3.5, $\delta(A) \geq n$. Otherwise, by Lemma 3.4, $\delta(A) \geq x+y$. In either case, $|X \cap A| \leq \delta(A)$. If G_A is not connected, let A_1 denote the set of vertices in a connected component of G_A . Then $\delta(A_1) \geq |X \cap A_1|$ and $\delta(A - A_1) \geq |X \cap (A - A_1)|$. So $\delta(A) = \delta(A_1) + \delta(A - A_1) \geq |X \cap A| + |X \cap (A - A_1)| = |X \cap A|$. ■

Let G be a graph of order n and size m . Let p denote the number of distinguished vertices in G . Construct a graph G' by replacing each distinguished vertex v in G by a $pm \times pm$ grid graph H_v and assigning the edges incident on v to vertices on the topmost row of H_v . The vertices in the top row are picked at random, subject to the condition that at most one edge is assigned to each vertex. Since a $pm \times pm$ grid graph has $pm > m$ vertices in each row, this condition can always be satisfied. We call G' a *grid expansion* of G . Note that the construction of G' can be accomplished

in polynomial time. Figure 3.2 shows a graph and one of its grid expansions. Vertices u and v are assumed to be distinguished.

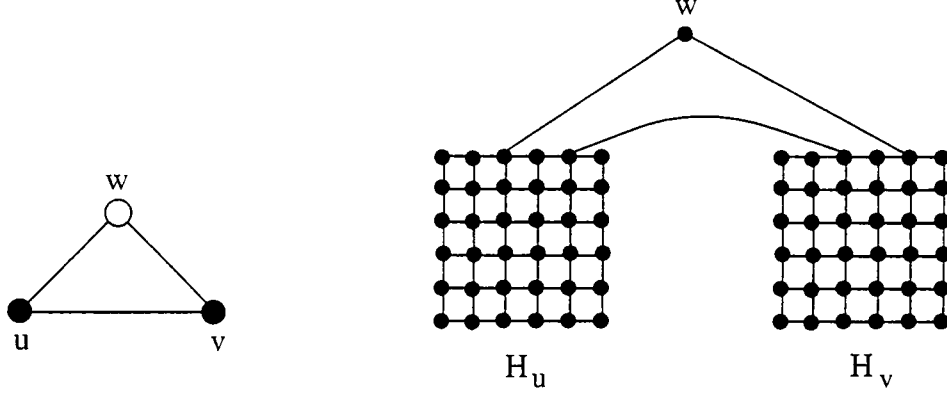


Figure 3.2: A grid expansion.

We prove that G is a “yes” instance of $\text{MDCSB}(d, p)$, if and only if G' is a “yes” instance of $\text{FPGA Minimization}(d, (p^2 + 1)m^2 + n, p)$. Before proceeding, we state and prove a structural lemma about partitions of G' .

Lemma 3.8 Let G denote a graph with n vertices and m edges. Let $\{v_1, \dots, v_p\}$ denote the set of distinguished vertices in G . Suppose the grid expansion G' of G is a “yes” instance of $\text{FPGA Minimization}(d, (p^2 + 1)m^2 + n, p)$ for some $d \leq m$. Then G' has a partition consisting of p subsets of degree at most d and size at most $(p^2 + 1)m^2 + n$, such that for each $i \in \{1, \dots, p\}$, there is a subset in the partition that contains all the vertices in H_{v_i} .

Proof Consider any partition of G' into p subsets of size at most $(p^2 + 1)m^2 + n$ and degree at most d . For $1 \leq i \leq p$, let S_i denote a subset in the partition that contains m^2 or more vertices from H_{v_i} ; S_i exists, since $\frac{|V(H_{v_i})|}{p} > m^2$. S_i must contain at least $(p^2 - 1)m^2$ vertices from H_{v_i} , or else, by Lemma 3.6, $\delta(S_i) \geq \text{minimum}(\sqrt{2m^2}, \sqrt{2p^2m^2 - 2m^2}) = \sqrt{2}m > d$, a contradiction. Since there is not enough room in a subset to accommodate $(p^2 - 1)m^2$ vertices each from two disjoint grid graphs, S_i and S_j are distinct, if $i \neq j$.

Consider the effect of moving $V(H_{v_i})$ into S_i , for all i . After the move, each subset contains fewer than $p^2m^2 + n$ vertices, satisfying the size constraint. To show that the degree constraint is not violated, we view the move as a sequence of p smaller moves, where the i^{th} move consists of moving $V(H_{v_i})$ into S_i , and prove that each move produces a partition that satisfies the degree constraint.

We first show that the i^{th} move does not increase $\delta(S_i)$. Let $A \subseteq V(H_{v_i})$ denote the set of vertices added to S_i by this move. After the move, let E_1 denote the set of edges between A and $S_i - A$; let E_2 denote the set of edges between A and $V(G') - S_i$. Proving that $\delta(S_i)$ does not increase is equivalent to showing that $|E_2| \leq |E_1|$. Since every edge in E_2 has a vertex in H_{v_i} and a vertex in $G' - H_{v_i}$ as endpoints, it follows from the construction of G' that $|E_2|$ is no more than the number of vertices in A that belong to the topmost row of H_{v_i} . Since $|A| \leq m^2 < \frac{|V(H_{v_i})|}{2}$, it then follows from Lemma 3.7 that $|E_2|$ is no more than the number of edges between A and $V(H_{v_i}) - A$, which is at most $|E_1|$.

Now consider a subset S_j , where $j \neq i$. Let B denote $V(H_{v_i}) \cap S_j$, before the i^{th} move. Let E_3 denote the set of edges between B and $S_j - B$, and E_4 the set of edges between B and $V(G') - S_j$. Since B is exactly the set of vertices moved from S_j to S_i , to prove that $\delta(S_j)$ does not increase, it suffices to show that $|E_3| \leq |E_4|$. This follows since $|E_3|$ is no more than the number of vertices in B that belong to a single row of H_{v_i} , which, by Lemma 3.7, is no more than the number of edges between B and $V(H_{v_i}) - B$, which is no bigger than $|E_4|$. ■

Theorem 3.8 For any fixed $p \geq 3$, FPGA Minimization is $\mathcal{NP}$ -hard, even on graphs of maximum degree four.

Proof Let G denote a graph with n vertices and m edges. Let $p \geq 3$ denote the number of distinguished vertices in G . Suppose G is a “yes” instance of $\text{MDCSB}(d, p)$. Given a partition $\{S_1, \dots, S_p\}$ of G into subsets of degree at most d , we can replace each distinguished vertex v_i by its grid graph H_{v_i} to obtain a partition $\{S'_1, \dots, S'_p\}$

of G' . For each i , $|S'_i| < p^2m^2 + |S_i| < (p^2 + 1)m^2 + n$, and $\delta(S'_i) \leq \delta(S_i)$. Thus G' is a “yes” instance of FPGA Minimization($d, (p^2 + 1)m^2 + n, p$).

Conversely, suppose G' is a “yes” instance of FPGA Minimization($d, (p^2 + 1)m^2 + n, p$). Let $\{S'_1, \dots, S'_p\}$ denote a partition of G' into subsets of size at most $(p^2 + 1)m^2 + n$ and degree at most d . Then, by Lemma 3.8, we may assume that for each grid graph H_{v_i} in G' , there is a subset S'_i that contains all the vertices in H_{v_i} . We construct a partition $\{S_1, \dots, S_p\}$ of G by replacing each grid graph H_{v_i} by the corresponding distinguished vertex v_i . Since $\delta(S_i) = \delta(S'_i)$ for each i , G is a “yes” instance of MDCSB(d, p).

Our reduction from MDCSB to FPGA Minimization is complete. If all non-distinguished vertices in G have degree three or less, then all vertices in G' have degree four or less. Since MDCSB is $\mathcal{NP}$ -hard on graphs with three or more distinguished vertices even if all non-distinguished vertices have maximum degree three (see Theorem 3.7 and the ensuing discussion), the result follows. ■

A similar approach can probably be used to show that FPGA Minimization is $\mathcal{NP}$ -hard on graphs of maximum degree three. Instead of grid graphs, one could use “walls,” which are structurally similar, but have maximum degree three. A formal proof, however, appears to be tedious.

3.6 Obstruction Sets

Since no practical direct algorithms are known for the fixed-parameter versions of the partitioning and cutting problems we have discussed, exploration of obstruction sets might be worthwhile. For MDGP and FPGA Minimization, both lower and upper bounds on the size of the obstruction sets are known [Pla]. The lower bounds are exponential in all parameters. For Multiway Cut and MDCSB, it is not even known if the obstruction sets are computable. We have been able to establish lower bounds on the sizes of these sets. These bounds indicate that the size of the obstruction sets

grows rapidly as the values of the input parameters increase.

Theorem 3.9 The size of the obstruction set to Multiway Cut(d) is at least exponential in d .

Proof Consider a tree T , whose vertices are all distinguished, with $d + 1$ edges. Clearly, T is a “no” instance of Multiway Cut(d). Taking a subgraph or lifting a pair of edges transforms T into a “yes” instance. T is thus a member of the obstruction set to Multiway Cut(d). So the obstruction set is no smaller than the number of trees with $d + 1$ edges, which is greater than 2^{d+1} [HP]. ■

A *star graph* consists of one or more pendant vertices adjacent to a central vertex. The central vertex may be adjacent to the other vertices by more than one edge each.

Theorem 3.10 The size of the obstruction set to MDCSB(d, p) is at least exponential in $\sqrt{d}$.

Proof Let G denote a star graph, whose vertices are all distinguished, with a central vertex of degree $d + 1$. G is a “no” instance of MDCSB(d). Since taking a subgraph or lifting a pair of edges reduces the degree of the central vertex and transforms G into a “yes” instance, G must be a member of the obstruction set to MDCSB(d, p). Therefore, the size of the obstruction set is no less than the number of star graphs with $d + 1$ edges.

To obtain a bound on this number, we establish a correspondence between star graphs with $d + 1$ edges and sets of positive integers that add up to $d + 1$. Given a set $\{d_1, \dots, d_n\}$ such that $d_1 + \dots + d_n = d + 1$, a unique star graph can be constructed with a central vertex v , n pendant vertices $v_1, \dots, v_n$ and d_i edges between v and v_i , for each i . Star graphs derived from distinct sets are non-isomorphic, as their degree sequences must be different. Thus the number of star graphs with $d + 1$ edges is no less than the number of sets of positive integers that add up to $d + 1$, which is exponential in $\sqrt{d}$ (see [Ros], page 222). ■

3.7 Approximations

It is straightforward to see that none of the $\mathcal{NP}$ -hard problems we have discussed permits polynomial-time absolute approximation algorithms. Consider MDGP, for example. Given a graph and an integer l , replacing each edge with l copies of itself gives us a graph that is a “yes” instance of MDGP(dl, k) if and only if the original graph is a “yes” instance of MDGP(d, k). By choosing a large enough l and invoking an absolute approximation algorithm for MDGP on the new graph, we can solve MDGP exactly on the original graph, violating Theorem 3.2. A similar argument shows that none of our problems is absolutely approximable in polynomial time (assuming $\mathcal{P} \neq \mathcal{NP}$).

The question of whether polynomial-time relative approximations exist remains open, except for Multiway Cut, for which a factor-of-two approximation algorithm is known. The algorithm for Multiway Cut is a simple greedy approach that is based on network flow [DJPSY]. We have been unable to obtain relative approximations using this method for the other problems studied in this chapter.

3.8 Discussion

When all input parameters are fixed, these problems are closed under the immersion order, and therefore, solvable in polynomial time. The degree of this polynomial, however, depends on the values of the parameters and could be huge when the parameters are large. Contrast this with the cutwidth minimization problem, whose fixed-parameter variant is immersion-closed and non-trivial, yet solvable constructively in linear time using well-quasi-ordering techniques [FL5]. This linear-time algorithm is derived from the observation that all “yes” instances have bounded treewidth. Unfortunately, there is no such bound for the problems we consider.

One might ask what sort of objective functions produce graph families that are not immersion-closed. One example relevant to circuit partitioning is the balance

criterion. That is, we attempt to minimize the ratio between the biggest and smallest subsets in a partition. It is easy to construct graphs with subgraphs that have a higher balance ratio than the graphs themselves. Thus the family of graphs with bounded balance ratio cannot be closed under any order that includes the taking of a subgraph as one of its defining operations. Of course, this precludes the family from being closed in the immersion order.

To place our results in perspective, note that it has been known for some time that a number of problems of practical interest correspond to closed graph families under the minor order¹. Such problems include Vertex Cover, Gate Matrix Layout and many others [FL5]. Unlike the minor order, the immersion order had few known applications until now. The partitioning and cutting problems we discussed in this chapter suggest that this order is worthy of further study.

¹ $H \leq G$ in the minor order if H can be obtained by a series of subgraph and edge contraction operations.

Chapter 4

Toward a Library of Tests

We have seen that the immersion order can be applied to a number of graph partitioning problems. A few other problems of relevance to circuit layout are also known to be amenable to this order. Consider, for example, the Cutwidth problem, which was defined in Section 2.3. The cutwidth metric is related to minimizing the area of a layout in which the circuit components are placed along a line (see [CNSD, FHKY, HPK] for examples of this style of layout).

Determining cutwidth is $\mathcal{NP}$ -complete in general. Nevertheless, for any arbitrary constant k , the family of graphs of cutwidth at most k is closed under immersion, and hence, recognizable in polynomial time. Multidimensional generalizations of cutwidth can likewise be solved in polynomial time using a collection of immersion tests. In fact, these problems can be solved in linear time, by taking advantage of the result that all “yes” instances have bounded treewidth [FL5].

The immersion test of [FL5] is powerful in that it is general, but impractical because of its complexity. Little is known about practical immersion tests for particular graphs, which are required if an approach based on obstruction-testing is to be feasible. This is probably due in part to the immersion order having been neglected in comparison to the better-known minor and topological orders, and in part to the fact that immersion tests appear harder to devise than tests in other orders. The

problems encountered in the previous chapter indicate that the immersion order is worthy of further study. Accordingly, we develop practical algorithms to perform two fundamental immersion tests.

Complete graphs are often among the smallest and densest obstructions, both for cutwidth and for the partitioning problems considered in Chapter 3. Testing for K_1 and K_2 are trivial. Detecting a K_3 is easy: K_3 is immersed in any graph of order three or more unless the graph is a tree in which no pair of multiple edges is incident on the same vertex. The first really difficult test is for K_4 . We design a practical linear-time K_4 test, that also finds a model of K_4 if any exist. The K_4 testing and model-finding algorithms were the result of a collaborative effort. Only my contribution is described in this dissertation. Some of the results presented here are mentioned without proof in [Rac]. For full details of the algorithms, refer to [BGLR].

Another fundamental test is for the k -bond, a graph consisting of just two vertices that are adjacent by k edges. The family of k -bonds, for various values of k , are the smallest obstructions to Cutwidth and to Multiway Cut. It is known that the presence of a k -bond in a graph of order n can be detected in $O(n^4)$ time using techniques based on network flow. We restrict attention to series-parallel graphs, on which the same techniques yield an $O(n^2)$ k -bond test. Our algorithm, on the other hand, does not use network flow and detects a k -bond in linear time.

Our experience with immersion testing suggests that edge-connectivity is significant in this regard. To illustrate, our K_4 test uses a fast algorithm that finds three-edge-connected components, to preprocess the input graph (see [BGLR] for details of this algorithm). We use a different approach than in [BGLR] to devise an algorithm that finds k -edge-connected vertex sets in series-parallel graphs. This algorithm runs in $O(kn)$ time on a graph of order n , improving on flow-based algorithms, which take $O(n^2)$ time.

4.1 A K_4 Test

If a graph contains a topological K_4 , then it also contains an immersed K_4 . A topological K_4 can be detected, and a model found, in linear time (see, for instance, [LG]). Thus we consider only those graphs with no topological K_4 . These are exactly the series-parallel graphs [Duf]. We concentrate on edge-disjoint paths, which are relevant because of the following alternate characterization of immersion containment: $H \leq_i G$ if and only if there exists an injection from $V(H)$ to $V(G)$ under which the edges of H are mapped to edge-disjoint paths in G between the images of the corresponding endpoints. Under such an injection, an image vertex is called a *corner* of H in G ; all image vertices and their associated paths are collectively called a *model* of H in G . Our algorithms exploit the edge connectivity of the input graph.

4.1.1 Three-Edge Connectivity

A *cut edge* of a connected graph G is an edge whose removal disconnects G . A pair of edges, neither of which is a cut edge, is said to form a *cut edge pair* if removing both of them disconnects G . Two vertices are *three-edge-connected* if there are at least three edge-disjoint paths between them. G is *three-edge-connected* if and only if it has no cut edges and no cut edge pairs.

A *three-edge-connected component* of G is a graph G' where $V(G') \subseteq V(G)$ is a maximal set of vertices that are pairwise three-edge-connected in G . $E(G')$ contains all edges induced by $V(G')$ plus a (possibly empty) set of *virtual edges* defined as follows: for $\{u, v\} \subseteq V(G')$, a virtual edge between u and v is added to $E(G')$ for each distinct $\{x, y\} \subseteq V - V'$ such that x is a neighbor of u , y is a neighbor of v , and the edges connecting u to x and v to y form a cut edge pair in G . Note that, due to the possible presence of virtual edges, a three-edge-connected component will not necessarily be a subgraph.

Lemma 4.1 If $K_4 \leq_i G$, then K_4 is immersed in some three-edge-connected component of G .

Proof Let a, b, c and d denote the corners of a K_4 model in G . These corners are joined (in G) by at least six edge-disjoint paths: $[ab]$, $[ac]$, $[ad]$, $[bc]$, $[bd]$ and $[cd]$. Thus a and b are connected by at least three edge-disjoint paths: $[ab]$, $[ac][cb]$ and $[ad][db]$. Maximality ensures that the three-edge-connected component containing a also contains b and, by symmetry, c and d . Let G_a denote this component. If $[ab]$ contains edges not in G_a , then $[ab]$ can be written as $[au]e_1[xy]e_2[vb]$, where $\{e_1, e_2\}$ is a cut edge pair in G , with e_1 connecting u and x , and e_2 connecting y and v . There must be a virtual edge e_3 connecting u and v in G_a . Thus a and b are connected within G_a by $[au]e_3[vb]$, which is edge disjoint from the other five paths of the model. By symmetry, all pairs of corners are so connected within G_a . ■

The proof of Lemma 4.1 can be generalized to any three-edge-connected graph immersed in G . In order to make use of the result of Lemma 4.1, we need to show that three-edge-connected components of series-parallel graphs are themselves series-parallel. As mentioned earlier, series-parallel graphs can be characterized as those that exclude a topological K_4 . Topological containment can be defined as a restricted form of immersion containment, with lifting permitted only at vertices of degree two. Alternately, topological containment can be viewed as an injection, but with vertex-disjoint rather than edge-disjoint paths.

Lemma 4.2 Each three-edge-connected component of a series-parallel graph is series-parallel.

Proof The proof is straightforward, by noting that virtual edges introduce no additional vertex-disjoint paths. ■

Lemma 4.3 Let G' denote the graph obtained from G by deleting edges until no pair of vertices is connected by more than four edges. Then $K_4 \leq_i G$ if and only if $K_4 \leq_i G'$.

Proof If $K_4 \not\leq_i G$, then $K_4 \not\leq_i G'$, since G' is a subgraph of G . If $K_4 \leq_i G$, let a, b, c and d denote the corners of a K_4 model in G . Suppose five or more edges connecting two vertices u and v are contained in this model. Without loss of generality, assume the images of the K_4 edges are simple paths. For some pair of corners, say a and b , all five paths with an endpoint at a or b ($[ab], [ac], [ad], [bc], [bd]$) contain an edge between u and v . Either u is a corner, or it can be made a corner by replacing a with u and deleting the three subpaths of the form $[au]$. Similarly, either v is a corner or b can be replaced with v . G therefore contains a K_4 model with corners u, v, w and x , where $\{w, x\} \subset \{a, b, c, d\}$. At most one of $[uw], [vw]$ must contain an edge between u and v ; at most one of $[ux], [vx]$ must contain an edge between u and v . Thus, of the six edge-disjoint paths of this model, at least two need not contain any of the edges between u and v , and all but four edges between u and v can be eliminated. This construction is iterated until a model is obtained in which no two vertices are connected by more than four edges. This model must be present in G' . ■

In the remainder of this section, we assume that there are at most four edges between any pair of vertices in a graph. We note that if a simple graph H is series-parallel, then $|E(H)| \leq 2|V(H)| - 3$. From this bound and Lemma 4.3, we know that all graphs of interest have at most a linear number of edges.

4.1.2 The Decision Algorithm

We first verify that the input graph is series-parallel and use the methods in [BGLR] to break it into three-edge-connected components. We then search each three-edge-connected component separately for an immersed K_4 . In a three-edge-connected graph, suppose v is a vertex with exactly two neighbors, u and w , and suppose there is only one edge between v and w . (Thus there are at least two edges between u and v by three-edge-connectivity.) We say that v is *pruned* if all but two edges between u and v are deleted. We say a graph is pruned if each vertex fitting the profile of v is pruned.

Procedure K_4 -test

```
Input:   A series-parallel graph  $G$ .
Output:  Yes, if  $K_4 \leq_i G$ , No otherwise.
begin procedure
  for each three-edge-connected component of  $G$ 
    do for each vertex  $v$  with exactly one neighbor
      do delete all but three edges incident on  $v$ ;
    if there is a cut point  $v$  with  $\delta(v) \geq 7$  then output Yes and halt;
    for each biconnected component  $B$  with  $|V(B)| \geq 4$ 
      do begin
        prune  $B$ ;
        if there is a vertex  $v$  in  $B$  with  $\delta(v) \geq 5$  then output Yes and halt;
      end
    output No and halt;
end procedure
```

4.1.3 The Correctness of K_4 -test

The correctness of K_4 -test relies on a number of lemmas, which follow. Before proceeding, we make a few useful observations.

Observation 4.1 If $H' \leq_i H$, and if M' is a K_4 model in H' , then in H there is a K_4 model with the same corners as M' .

Observation 4.1 follows from noting that edges in H' map to edge-disjoint paths in H , and that in H a suitable K_4 model can be found merely by replacing the edges of M' with their image paths in H .

The next observation is a well-known property of series-parallel graphs (see [KT], for example, for a proof).

Observation 4.2 Every biconnected series-parallel graph with four or more vertices contains at least two non-adjacent vertices with exactly two neighbors.

Suppose v is a vertex with exactly two neighbors, x and y . We say that v is *shorted* if we lift all pairs of edges vx and vy and delete any remaining edges incident on v along with v itself.

Observation 4.3 Shorting preserves biconnectivity, three-edge-connectivity and series-parallelness.

Observation 4.3 holds because shorting a vertex does not change the number of vertex-disjoint or edge-disjoint paths between any pair of remaining vertices.

Observation 4.4 A biconnected component of a three-edge-connected graph is three-edge-connected.

Observation 4.4 follows from noting that edge-disjoint paths may as well be made simple and that, whenever a pair of vertices lies in the same biconnected component, all vertices along simple paths connecting them in the original graph must also lie in this component.

Lemma 4.4 Let G denote a graph in which a vertex, v , has exactly one neighbor. Let G' be obtained from G by deleting all but three edges incident on v . Then $K_4 \leq_i G$ if and only if $K_4 \leq_i G'$.

Proof If $K_4 \leq_i G$, then G contains a K_4 model in which edge images are simple paths. Since v cannot be an intermediate vertex in a simple path, at most three of the edges incident on v are needed in this model. Thus $K_4 \leq_i G'$. If $K_4 \not\leq_i G$, then $K_4 \not\leq_i G'$, since G' is a subgraph of G . ■

Lemma 4.5 Let G be three-edge-connected. Let u and v denote non-cut point vertices in G . Let w denote any other vertex in G . Then there exist three mutually edge-disjoint paths, each connecting w with either u or v , such that at most two of these paths contain u , and at most two contain v .

Proof The paths we seek to identify are illustrated in Figure 4.1, where the dashed lines denote edge-disjoint paths that do not contain u or v as an intermediate vertex. Consider three mutually edge-disjoint paths P_1 , P_2 and P_3 , each between w and $\{u, v\}$.

These paths exist because G is three-edge-connected. Assume all three contain, say, u . Hence all three may as well be simple and end at u . Consider now some path P between w and v that does not contain u (P exists since u is not a cut point). P may contain vertices and edges in P_1 , P_2 and P_3 . Let y be the last vertex in P (counting from w) that is also in P_1 or P_2 or P_3 . Without loss of generality, assume y is in P_1 . We can construct a path P' from w to v , by taking P_1 until we reach y , and using P from there on. Thus P' , P_2 and P_3 are the desired edge-disjoint paths, with P' not containing u . ■

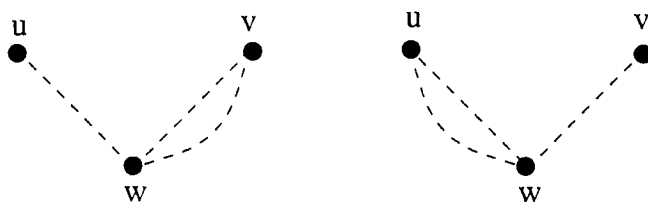


Figure 4.1: Edge-disjoint paths in a three-edge-connected graph.

Figure 4.2 depicts a graph we will discuss frequently, henceforth termed graph M .

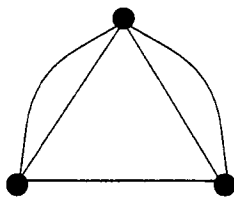


Figure 4.2: The graph M .

Lemma 4.6 Let G be three-edge-connected. Let v denote a non-cut point vertex in G with degree at least four, let u and w be neighbors of v , and suppose u and v are connected by at least two edges. Then G contains an M model, with corners u , v and w , and with v the image of M 's degree-four vertex.

Proof We restrict our attention to G' , the biconnected component of G containing v . (G' is three-edge-connected by Observation 4.4. Since v is not a cut point, its

neighborhood is unchanged in G' .) From Lemma 4.1, we know that there are three mutually edge-disjoint paths from w to $\{u, v\}$ such that at most two of these paths contain u and at most two contain v . Since v and w are adjacent, one of these paths is an edge. If one of the other paths contains v , the lemma holds. So suppose neither contains v . See Figure 4.3(a). To complete an M model, we must find an edge-disjoint path $[vw]$. If there are three or more edges between u and v , we can construct this path by combining an edge between v and u , and one of the paths $[uw]$. So assume there are only two edges between u and v . Let x denote a neighbor of v other than u or w . Since G' is biconnected, there is a path $[xw]$ that does not contain any of the edges incident on v . Let y denote the first vertex on this path (counting from x) common to either of the two paths $[uw]$. We combine the edge between v and x with the paths $[xy]$ and $[yw]$ to get the desired path $[vw]$. See Figure 4.3(b). ■



Figure 4.3: Graphs used in the proof of Lemma 4.6.

Lemma 4.7 Let G be three-edge-connected and series-parallel, with at least three vertices. Let v denote a vertex in G with degree at least four. Then G contains an M model, with v the image of M 's degree-four vertex, and corner u adjacent to v and corner w adjacent to u or v .

Proof Suppose v has only one neighbor, which must be u . Then there are four edges between u and v . Let w denote an arbitrary neighbor of u , besides v . Since G is three-edge-connected, and since u is a cut point, the graph depicted in Figure 4.4(a) is immersed in G , satisfying the statement of the lemma. Suppose v has two or more

neighbors, and v is a cut point. Let u and w denote arbitrary neighbors of v . Now the graph in Figure 4.4(b) is immersed in G , again satisfying the statement of the lemma. Finally, suppose v has two or more neighbors, and v is not a cut point. For this case, we prove something slightly stronger, namely, that an M model exists with v the image of M 's degree-4 vertex, and corners u and w both adjacent to v .

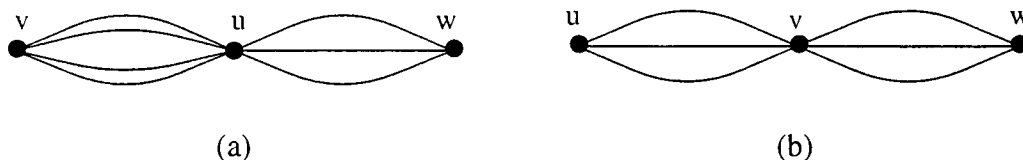


Figure 4.4: Graphs used in the proof of Lemma 4.7.

We proceed by contradiction, and let H denote a counterexample. Without loss of generality, we assume H is minimal. That is, no counterexample exists with fewer than $|V(H)|$ vertices and, for this number of vertices, no counterexample exists with fewer than $|E(H)|$ edges. H must be biconnected, else the biconnected component containing v provides a smaller counterexample. Similarly, no three-edge-connected, series-parallel graph containing v and all its incident edges can be properly immersed in H , else such a graph would again contradict minimality. This implies that any vertex with exactly two neighbors must be adjacent to v (else the vertex could be shorted). So there must be some vertex, x , that is adjacent to v and that has exactly one other neighbor, y . By Lemma 4.6, we know that there is exactly one edge between v and x . Let e_1 denote this edge. Three-edge-connectivity requires that there are two or more edges between x and y . Let e_2 and e_3 denote two of these edges. Now consider H' , obtained from H by shorting x . H' satisfies the conditions of the lemma and so (by the minimality of H) contains an M model, with v the image of the degree-4 vertex in M , and corners u and w both adjacent to v . The only edges in H' not in H are the ones between v and y , implying that y plays the role of u (or, by symmetry in this case, w). But this means that, in H , we can replace the edge-disjoint paths

$[vy]$, $[vy]$ and $[uy]$ with e_1 , $[vy]e_2$ and $[uy]e_3$ respectively, giving us an M model with corners v , x and w , a contradiction to the presumed existence of a counterexample.

■

Lemma 4.8 Let G be three-edge-connected and series-parallel, and let all vertices in G with exactly one neighbor have degree three. Suppose G has a cut point v with degree seven or more. Then there is a K_4 model in G with corners u , v , w and x , where u and x are adjacent to v and w is adjacent to u or v .

Proof Let $C_1, \dots, C_k$ denote the connected components of $G - \{v\}$. Let A_i denote C_i augmented with a copy of v and the edges it induces. Each A_i is three-edge-connected, and thus contains a model of the triple-edge shown in Figure 4.5(a), with any pair of vertices serving as the corners. Without loss of generality, assume A_1 contains the least number of edges incident on v , and let H denote $G - C_1$. It follows that $\delta(v) \geq 4$ in H and that H has at least three vertices. Thus, by Lemma 4.7, there is an M model in H with v the image of the degree-4 vertex in M , and with corner u adjacent to v and corner w adjacent to u or v . This M model can be combined with a model of the triple-edge in A_1 to form in G a model of the graph shown in Figure 4.5(b), which contains the desired K_4 model. ■

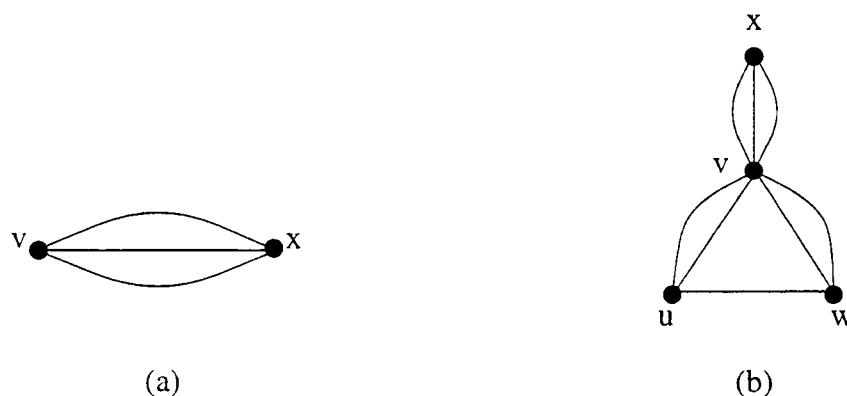


Figure 4.5: Graphs used in the proof of Lemma 4.8.

Lemma 4.9 Suppose G has no cut point with degree exceeding six. Then $K_4 \leq_i G$ if and only if K_4 is immersed in a biconnected component of G .

Proof If a biconnected component of G contains K_4 , then so does G , because a biconnected component is a subgraph. To prove the converse, consider a K_4 model in G with the K_4 edges mapped to simple paths. Let u, w, x and y denote the corners of this model, and suppose there is a cut point v that separates them. We know that v cannot be one of the corners, else $\delta(v) \geq 7$ (see Figure 4.6(a)). Nor can v separate two corners from the others, else $\delta(v) \geq 8$ (see Figure 4.6(b)). So it must be that v separates just one corner, say y , from the others (see Figure 4.6(c)). Thus the edge-disjoint paths $[uy]$, $[wy]$ and $[xy]$ all pass through v , and we can construct another K_4 model in which v replaces y as a corner. By iterating this replacement, we eventually get a K_4 model all of whose corners (and paths) are in the same biconnected component. ■

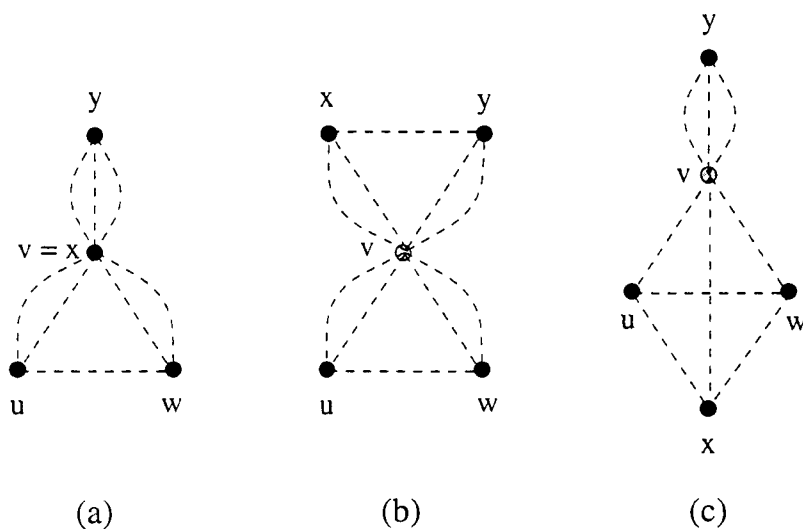


Figure 4.6: Models of K_4 that span a cut-point.

Lemma 4.10 Let v denote a vertex with exactly two neighbors, u and w , and suppose there is exactly one edge between v and w . Then u and v can be corners of a given K_4 model only if $\delta(u) \geq \delta(v) + 2$.

Proof Let x and y denote the other corners of this model. Paths $[ux]$ and $[uy]$ need not contain uv . Either $[vx]$ or $[vy]$ has to pass through u . Thus at least three edges are incident on u in addition to the edges between u and v (see Figure 4.7), and the lemma follows. ■

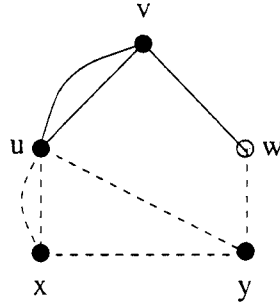


Figure 4.7: A model of K_4 with corners u , v , x and y .

Lemma 4.11 If G is series-parallel and of maximum degree four, then $K_4 \not\leq_i G$.

Proof Suppose otherwise, and let H denote a minimal counterexample. H must be three-edge-connected by Lemma 4.1. H must also be biconnected, since a cut point in a three-edge-connected graph has degree at least six. Thus, by Observation 4.2, H contains a vertex, v , with exactly two neighbors, u and w . It must be that v is needed as a corner in every K_4 model, else we can short it, contradicting minimality. So $\delta(v) = 3$ and we assume, without loss of generality, that v is connected to u by two edges and to w by one edge. We now fix the remaining corners of some K_4 model. Vertex u cannot be one of these corners, by Lemma 4.10. But now it is easy to see that u can replace v in this model, contradicting the fact that v must be a corner. ■

We henceforth use the term *candidate graph* to denote a biconnected, three-edge-connected, series-parallel graph with four or more vertices.

Lemma 4.12 In a candidate graph, G , suppose vertex v has exactly two neighbors, u and w , and suppose there are more edges between u and v than between v and w . If $\delta(u) - \delta(v) \geq 2$, then $K_4 \leq_i G$.

Proof Suppose otherwise, and let H denote a minimal counterexample. Let $x \neq v$ denote another vertex with exactly two neighbors. There must be at least two edges between x and u , else we can short x , contradicting minimality. Consider the effect of shorting v , producing the graph H' . Since $\delta(u) \geq 4$ in H' , we know from Lemma 4.6 that the M model illustrated in Figure 4.8(a) is immersed in H' . But this means that the graph shown in Figure 4.8(b), which contains K_4 , is immersed in H , thereby contradicting the assumption that H is a counterexample. ■

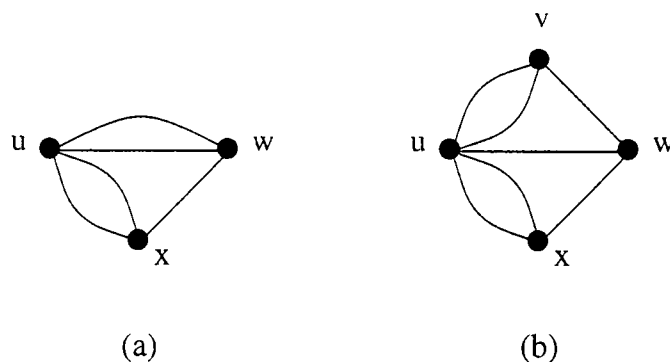


Figure 4.8: Graphs used in the proofs of Lemmas 4.12 and 4.14.

Recall pruning, as defined in Section 4.1.2.

Lemma 4.13 In a candidate graph, G , suppose vertex v has exactly two neighbors, u and w , and suppose there is exactly one edge between v and w . Letting G' denote the graph resulting from pruning v , $K_4 \leq_i G$ if and only if $K_4 \leq_i G'$.

Proof If $K_4 \leq_i G'$, then $K_4 \leq_i G$ as well, because $G' \subseteq G$. Suppose $K_4 \leq_i G$. If G contains a K_4 model in which v is not a corner, then so does G' , since pruning is irrelevant (at most one of the images of the K_4 edges in this model can pass through v .) So suppose v is a corner in every K_4 model in G . Vertex u must also be a corner in all these models, else we could replace v with u , forming a model in which v is not a corner. Now, by Lemma 4.10, $\delta(u) \geq \delta(v) + 2$, a property unchanged by pruning. Thus, by Lemma 4.12, $K_4 \leq_i G'$. ■

Lemma 4.14 In a pruned candidate graph, G , suppose vertex v has exactly two neighbors, u and w , and suppose there are three or more edges between u and v , and two or more edges between v and w . Then there is a K_4 model in G with corners u , v , w and x , where $x \notin \{v, w\}$ is a neighbor of u .

Proof The biconnectivity of G ensures that u has some neighbor other than v (and possibly w) to play the role of x . If v and x are u 's only neighbors, then there must be two or more edges between u and x (G has been pruned and yet there are at least three edges between u and v). Thus in G' , the graph that results from shorting v , $\delta(u) \geq 4$. We conclude from Lemma 4.6 that the M model illustrated in Figure 4.8(a) is immersed in G' , and the graph shown in Figure 4.8(b), which contains K_4 , is immersed in G . ■

Lemma 4.15 In a candidate graph, G , suppose vertex v has exactly two neighbors, u and w , suppose v is connected to both u and w by at least two edges each, and suppose u and w are adjacent. Then there is a K_4 model in G with corners u , v , w and x , where $x \notin \{v, w\}$ is a neighbor of u .

Proof As in the last lemma, such an x must exist. We apply Lemma 4.5, with w playing the role of v and x playing the role of w . Thus at least one of the graphs shown in Figure 4.9, both of which contain K_4 , is immersed in G . ■

Lemma 4.16 Let G denote a pruned candidate graph. $K_4 \leq_i G$ if and only if G has a vertex of degree five or more.

Proof We know from Lemma 4.11 that a candidate graph of maximum degree four contains no K_4 . To prove the converse, we proceed by contradiction and assume H denotes a minimal pruned candidate graph, with at least one vertex of degree five or more, but with no immersed K_4 . It is easy to verify that $|V(H)| \geq 5$, a necessary property because we will use shorting to contradict minimality, and a candidate graph

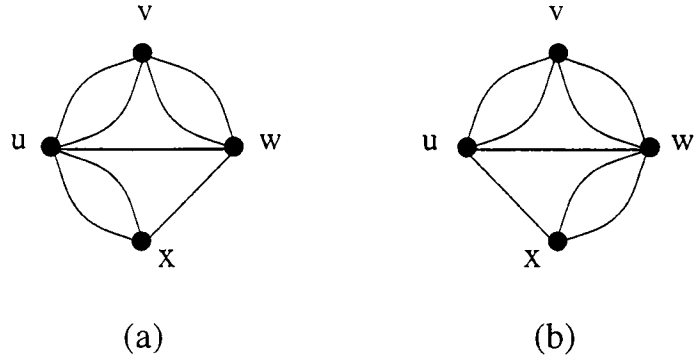


Figure 4.9: Graphs used in the proof of Lemma 4.15.

requires at least four vertices. Let v denote a vertex in H with exactly two neighbors, u and w , and assume there are at least as many edges between u and v as there are between v and w . Lemma 4.14 guarantees that $\delta(v) < 5$. If $\delta(v) = 4$, Lemma 4.15 and the fact that H is pruned ensure that u and w are not adjacent. But now we can short v , obtaining a pruned candidate graph that contradicts minimality. So $\delta(v) = 3$ and, by Lemma 4.12, $\delta(u) \leq 4$. Biconnectivity requires that there is at most one edge between u and w . But now we can again short v to obtain a pruned candidate graph, contradicting the presumed minimality of H . ■

This completes the proof of the correctness of K_4 -test. To implement K_4 -test efficiently, we store the input as a doubly-linked adjacency list (see Section 1.2 for details). Three-edge-connected components can be found using the methods in [BGLR] or by using an algorithm we present later in this chapter. Cut points and biconnected components can be found using depth-first search. All these steps take only linear time. Thus we have the following result.

Theorem 4.1 Algorithm K_4 -test correctly decides in linear time whether K_4 is immersed in an arbitrary series-parallel graph.

4.1.4 Finding a Model

Once the presence of K_4 has been detected in a graph, our method to identify a K_4 model proceeds in two steps. Algorithm **corners** is first invoked to modify the input graph until an appropriate set of corners is isolated. Then algorithm **paths** is used to find the images of the K_4 edges.

Algorithm **corners** marks vertices in the input graph as part of the corner-finding process. All vertices are assumed to be unmarked initially. Algorithm **corners** also maintains a list for every edge, to store the sequence of edges that may have been eliminated by shorting. Each list is assumed to contain only the edge itself initially.

We now address the correctness of **corners**. Suppose G contains a cut point v of degree at least seven. Lemmas 4.6 and 4.7 tell us how to find the corners of an M model. From this, Lemma 4.8 tells us how to find the corners of a K_4 model, as long as either (1) $G - \{v\}$ has three or more connected components or (2) there is an augmented component in which $\delta(v) \geq 4$ and at least two edges exist between u and v . If neither of these conditions is initially satisfied, condition (2) is easily forced with a series of vertex deletions and shorting operations.

So suppose no cut point of degree seven or more exists, and consider some biconnected component containing an immersed K_4 . This component must also contain a vertex with exactly two neighbors (Observation 4.2) and a vertex of degree five or more (Lemma 4.16). In this event, we employ Lemmas 4.14, 4.15, and 4.16, plus Lemma 4.17, which follows.

Lemma 4.17 In a candidate graph, G , suppose vertex v has exactly two neighbors, u and w , and suppose there are two edges between u and v , one edge between v and w , and $\delta(u) \geq 5$. Let x denote a neighbor of u other than v or w . If either there are two or more edges between u and x , or there is an edge between u and w , then there is a K_4 model in G with u, v, w and x as corners.

Proof In G' , the graph resulting from shorting v , $\delta(u) \geq 4$, and u is connected to

Algorithm corners

Input: A three-edge-connected series-parallel graph G containing an immersed K_4 .

Output: The four corners u, v, w, x , of a K_4 model in G .

begin procedure

for each vertex with only one neighbor **do** delete all but three incident edges;

if G has a cut point v of degree seven or more

then if $G - \{v\}$ has three or more connected components

then $u, w, x :=$ neighbors of v in G , each in a different
 connected component of $G - \{v\}$; **halt**;

else begin

$C_1, C_2 :=$ the connected components of $G - \{v\}$;

$A_1 := C_1$ augmented with v and the edges induced by v ;

$A_2 := C_2$ augmented with v and the edges induced by v ;

if $\delta(v) \geq 4$ in A_1 **then** $A := A_1, B := A_2$; **else** $A := A_2, B := A_1$;

while v is not connected to a vertex in A by more than one edge

do if there is a vertex in A with only one neighbor

then delete this vertex and its incident edges;

else short a vertex in A with only two neighbors;

$u :=$ a vertex in A connected to v by two or more edges;

if v has a neighbor in A besides u

then $w :=$ one of these neighbors;

else $w :=$ a neighbor of u in A besides v ;

$x :=$ a neighbor of v in B ; **halt**;

end

 discard all vertices and edges except those in a biconnected component containing K_4 ;

 prune all vertices with exactly two neighbors;

while true

do begin

$v :=$ an unmarked vertex with exactly two neighbors u and w , with
 at least as many edges between u and v as between v and w ;

if $\delta(v) \geq 5$, or $\delta(v) = 4$ and u and w are adjacent

then $x :=$ any neighbor of u besides v or w ; **halt**;

if $\delta(v) = 4$ or $\delta(u) = 4$

then short v ; start next iteration of while loop;

if u and w are adjacent

then $x :=$ any neighbor of u besides v and w ; **halt**;

if $\exists a \neq v$, with two or more edges between u and a

then $x := a$; **halt**;

if two vertices have degree five or more

then short v ; **else** mark v ;

end

end procedure

either x or w by at least two edges. Then by Lemma 4.6, there is an M model in G' with corners u , w and x , and with u the image of M 's degree-four vertex. Thus the graph in Figure 4.8(b), which contains the desired K_4 model, is immersed in G . ■

If an immersed K_4 cannot yet be identified, then a vertex, v , with exactly two neighbors is shorted as long as the resulting graph retains at least one vertex of degree at least five. Accordingly, if one of v 's neighbors, u , is the only vertex of degree at least five, there are exactly two edges between u and v , and no other vertex is connected to u by more than one edge, then v cannot be shorted. It suffices in this case to mark v as having been visited, since at most one vertex can be so marked and another candidate for v is always available.

In each iteration, **corners** deletes, shorts or marks some vertex. Handling any of these operations and updating the appropriate edge list requires only a constant number of steps if the input graph is represented as a doubly-linked adjacency list. Thus **corners** runs in linear time.

After the corners of a K_4 model have been found, we proceed to find the images of the K_4 edges using algorithm **paths** (see page 105). Algorithm **paths** uses the property that k edge-disjoint paths exist between a pair of vertices if and only if a network flow of value k is possible between them.

We address the use and complexity of **paths**. In the following figures, paths that are mere edges are shown as solid lines. These edges are temporarily deleted so that **paths** can be employed to find additional paths with multiple edges, depicted with dashed lines. To illustrate, consider the case in which the K_4 model spans a cut-point v and $G - \{v\}$ has three or more connected components. See Figure 4.10. Three calls are made to **paths**, each with v playing the role of s and k set to two. (The first call uses $u = t_1 = t_2$; the second uses $w = t_1 = t_2$; the third uses $x = t_1 = t_2$.) If $G - \{v\}$ has two connected components, two calls suffice. See Figures 4.11 and 4.12. If the K_4 model is in a single biconnected component, one call is enough. See Figure 4.13.

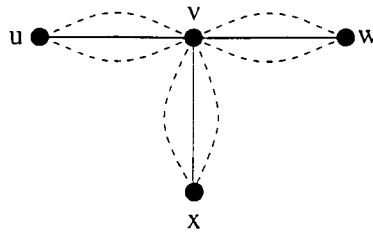


Figure 4.10: Paths to be found if $G - \{v\}$ has three or more connected components.

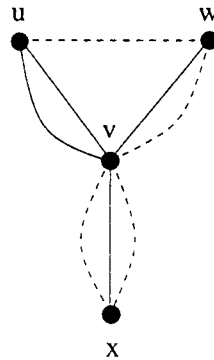


Figure 4.11: Paths to be found if v has at least two neighbors in u 's component.



Figure 4.12: Paths to be found if v has no neighbor besides u in u 's component.

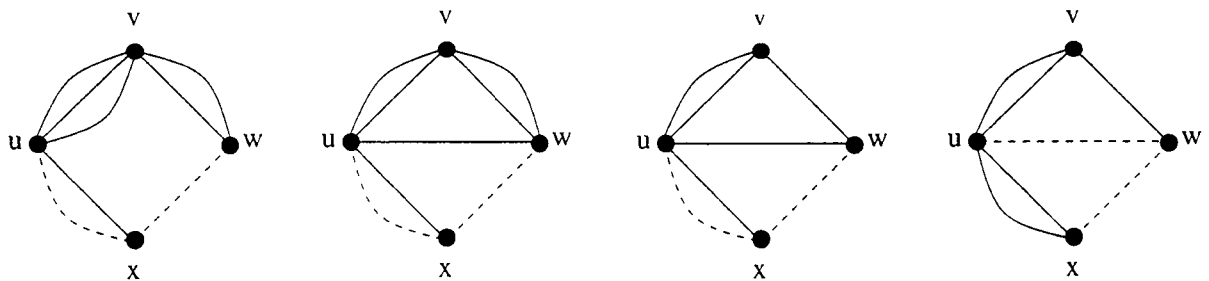


Figure 4.13: Paths to be found if the K_4 model lies in a biconnected component.

Algorithm **paths**

Input: a graph G and vertices $s, t_1, \dots, t_k$.

Output: edge-disjoint paths $p_1, \dots, p_k$, with p_i connecting s to t_i , if such paths exist.

begin procedure

$G' :=$ a digraph with $V(G') = V(G)$ and $E(G') = \{(u, v) \mid u \text{ and } v \text{ are adjacent in } G\}$;

for each edge (u, v) in G'

do $\text{capacity}((u, v)) :=$ number of edges between u and v in G ;

add to G' a vertex t and edges $(t_1, t), \dots, (t_k, t)$, each of capacity one;

find a flow of value k from s to t , if such a flow exists;

if there is no such flow **then** halt;

for each edge (u, v) in G'

do if both (u, v) and (v, u) have positive flow values

then begin

$\text{flow}((u, v)) := \text{maximum}\{0, \text{flow}((u, v)) - \text{flow}((v, u))\}$;

$\text{flow}((v, u)) := \text{maximum}\{0, \text{flow}((v, u)) - \text{flow}((u, v))\}$;

end

discard from G' all edges with zero flow;

for $i = 1$ to k

do begin

$p'_i :=$ a path in G' from s to t_i ;

$p_i :=$ a corresponding path in G from s to t_i ;

 decrement in G' the flow along each edge in p'_i by one;

 delete from G each edge in p_i ;

end

output $p_1, \dots, p_k$;

end procedure

Recall that the input to **paths** has at most a linear number of edges. Thus it takes only linear time to construct G' and to read off paths (using, for example, a shortest paths algorithm) after a flow of value k has been found. The running time of **paths** is therefore dominated by the algorithm for finding network flows. So we employ a flow method such as Ford-Fulkerson [FF], which runs in linear time as long as k is bounded by an integer constant and all edge-capacities are integers, as is the case here.

In summary, to find a K_4 model we invoke **corners** once and **paths** at most three times. The entire model-finding process is accomplished in linear time.

Theorem 4.2 Algorithms **corners** and **paths** correctly isolate a K_4 model if K_4 is

immersed in an arbitrary series-parallel graph.

4.1.5 Experimental Results

We implemented our algorithms in C and ran them on a SUN SPARCstation 20. Representative results are listed in Table 4.1. Each execution time shown is in seconds, and was obtained by averaging the times observed in a dozen runs. The graphs employed are (pseudo) random, generated by fixing the number of vertices and then randomly adding edges until the desired average degree was reached. Each edge was added only after verifying that its addition maintained series-parallelness, since a quick test for a topological (and hence an immersed) K_4 suffices to eliminate non-series-parallel graphs.

It is clear that from these results that our algorithms are practical, not just asymptotically optimal. They take only seconds to process graphs with thousands of vertices. The running time of the detection algorithm is affected mainly by the size of the input graph. One might suspect that the distribution of edges over vertices could also have an effect, but we sampled several edge-probability distributions and found no noticeable differences. On the other hand, the model-finding algorithm does appear to take slightly longer on graphs in which we have forced corners to be connected only by long paths. Even on such contrived instances, finding a model takes no more than twice the average time for random graphs of similar size.

4.1.6 Parallelization

It is not difficult to devise a parallel version of K_4 -test. Deciding whether a graph is series-parallel can be done in $O(\log^2 n + \log m)$ time with $O(m + n)$ processors [He]. The triconnected components algorithm of [FRT], modified slightly to find three-edge-connected components [Ram], runs in $O(\log n)$ time with $O(n \log \log n / \log n)$ processors. Biconnected components can be found in $O(\log n)$ time on a CRCW PRAM with $O((m + n)\alpha(m, n) / \log n)$ processors [FRT], where $\alpha(m, n)$ denotes the

Table 4.1: Running Times of K_4 Testing and Model-Finding Algorithms

average degree	number of vertices	detection time	percent with immersed K_4	model-finding time
1.0	200	0.01	0	N/A
	500	0.03	0	N/A
	1000	0.07	0	N/A
	2000	0.14	0	N/A
	5000	0.35	0	N/A
	10000	0.70	0	N/A
1.25	200	0.02	0	N/A
	500	0.05	0	N/A
	1000	0.09	0	N/A
	2000	0.16	0	N/A
	5000	0.41	17	0.72
	10000	0.80	42	1.42
1.5	200	0.02	8	0.03
	500	0.05	33	0.08
	1000	0.10	50	0.15
	2000	0.20	58	0.32
	5000	0.54	83	0.83
	10000	1.09	92	1.66
1.75	200	0.02	25	0.04
	500	0.05	67	0.09
	1000	0.11	83	0.19
	2000	0.24	100	0.37
	5000	0.61	100	0.88
	10000	1.19	100	1.75
2.0	200	0.02	67	0.05
	500	0.05	75	0.11
	1000	0.12	100	0.21
	2000	0.23	100	0.40
	5000	0.60	100	1.01
	10000	1.20	100	2.05
2.25	200	0.03	100	0.05
	500	0.07	100	0.11
	1000	0.13	100	0.22
	2000	0.26	100	0.43
	5000	0.64	100	1.13
	10000	1.27	100	2.24

inverse of Ackermann's function. It is straightforward to parallelize the rest of Algorithm **K_4 -test** to run in constant time with $O(n)$ processors.

Thus, in principle, it is possible to determine whether a graph has an immersed K_4 in $O(\log^2 n)$ time with $O(n)$ processors on the CRCW PRAM model. We did not implement this scheme because many of the algorithms mentioned are highly impractical. The problem of devising a practical parallel decision method remains open, as does the problem of finding even an asymptotically efficient parallel model-finding algorithm.

4.2 A k -bond Test

Vertices u and v in a graph G are said to be *k -edge-connected* if there are k or more edge-disjoint paths between them. It is clear that the k -bond is immersed in G if and only if some pair of vertices is k -edge-connected.

To decide if two vertices are k -edge-connected, it suffices to check if a network flow of value k or more exists between them. Let n denote the order of G . A naive k -bond immersion test might invoke a network flow algorithm $O(n^2)$ times, once for each pair of vertices in G . By making use of the fact that k -edge-connectivity is transitive, it is possible to reduce the number of calls to $O(n)$ [Guf]. Since the flow between a pair of vertices can be determined in $O(n^3)$ time [MPM], a k -bond test on general graphs takes $O(n^4)$ time. This can be further improved to $O(n^3 \log n)$ by observing that potential "no" instances can have no more than a linear number of distinct edges [Mad] and then applying the network flow algorithm of [GT].

We focus on testing for the k -bond on a restricted class of graphs, namely series-parallel graphs. The flow between a pair of vertices in a series-parallel graph can be determined in $O(n)$ time. Thus a flow-based k -bond immersion test works in $O(n^2)$ time. We present a simpler algorithm, **kbond-test**, that takes only $O(n)$ time. Observe that the k -bond is immersed in a graph if and only if it is immersed

in a biconnected component. Since biconnected components can be found in linear time, we assume that the input to **kbond-test** is biconnected.

Vertices u and v in G are *identified* by replacing them with a new vertex w , and assigning all edges incident on u and v to w .

Algorithm kbond-test

Input: A biconnected series-parallel graph G of order two or more and $k \in \mathbb{Z}^+$.
Output: Yes, if the k -bond is immersed in G , and No otherwise.
begin procedure
 while $|V(G)| > 2$
 do begin
 $v :=$ a vertex in G with exactly two neighbors;
 $u, w :=$ the two neighbors of v , with at least as many
 edges between u and v as between v and w ;
 if $k > \delta(v)$ **then** identify u and v ; **else** identify v and w ;
 end
 if $|E(G)| \geq k$ **then** output Yes; **else** output No;
 end procedure

4.2.1 Correctness and Complexity

We address the correctness and complexity of **kbond-test**.

Lemma 4.18 Let v denote a vertex in G , with exactly two neighbors u and w . Suppose there are at least as many edges between u and v as there are between v and w . Let G' denote the graph obtained by identifying u and v . For any $k > \delta(v)$, the k -bond is immersed in G if and only if it is immersed in G' .

Proof Since $k > \delta(v)$, v cannot be k -edge-connected to any other vertex in G . Observe that the operation of identifying u and v is identical to the operation of shorting v . Since shorting preserves the number of edge-disjoint paths between each pair of remaining vertices, any two vertices are k -edge-connected in G if and only if they are k -edge-connected in G' . ■

Lemma 4.19 Let v denote a vertex in G , with exactly two neighbors u and w . Suppose there are at least as many edges between u and v as there are between v and w . Let G' denote the graph obtained by identifying v and w . For any $k \leq \delta(v)$, if the k -bond is immersed in G , then it is immersed in G' .

Proof Let x and y denote a pair of k -edge-connected vertices in G . If both x and y are distinct from v , then the extra edges between u and w in G' ensure that x and y are k -edge-connected in G' as well. So assume, without loss of generality, that $y = v$. Let m_1 denote the number of edges between u and v , and m_2 the number between v and w .

Suppose $x \in \{u, w\}$ (see Figures 4.14 (a) and (b)). Consider the number of edge-disjoint paths between u and w in $G - \{v\}$. If $x = u$, then this is at least $k - m_1$. If $x = w$, then this is at least $k - m_2$. Since $m_1 \geq m_2$, u and w are connected by at least $k - m_1$ edge-disjoint paths in $G - \{v\}$. All these paths exist in G' as well. There are m_1 more edges between u and w in G' than in $G - \{v\}$. Thus, in G' , the number of edge-disjoint paths between u and w is at least $k - m_1 + m_1 \geq k$.

Now suppose $x \notin \{u, w\}$ (see Figure 4.14 (c)). There must be at least k edge-disjoint paths between x and $\{u, w\}$ in $G - \{v\}$, and hence in G' . Suppose l of these paths terminate at u . Then at least $k - l$ terminate at w . Since there are at least $m_1 \geq \frac{k}{2}$ edges between u and w in G' , the number of edge-disjoint paths between w and x in G' is at least $(k - l) + \min\{l, m_1\} = \min\{k, k + m_1 - l\} \geq$

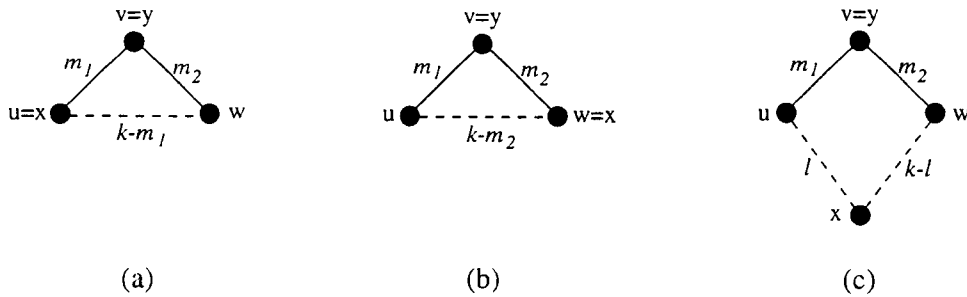


Figure 4.14: Possible k -bond models in G .

$\min\{k, \frac{3k}{2} - l\}$. Similarly, the number of edge-disjoint paths between u and x in G' is at least $l + \min\{k - l, m_1\} = \min\{k, m_1 + l\} \geq \min\{k, \frac{k}{2} + l\}$. For any value of l , either $\frac{3k}{2} - l \geq k$ or $\frac{k}{2} + l \geq k$. Thus one of u and w is k -edge-connected to x in G' .

In all three cases, some two vertices in G' are k -edge-connected. So the k -bond is immersed in G' . ■

Now we prove the converse of Lemma 4.19.

Lemma 4.20 Let v denote a vertex in G , with exactly two neighbors u and w . Suppose there are at least as many edges between u and v as there are between v and w . Let G' denote the graph obtained by identifying v and w . For any $k \leq \delta(v)$, if the k -bond is immersed in G' , then it is immersed in G .

Proof Let m_1 denote the number of edges between u and v , m_2 the number between v and w and m_3 the number between u and w (all in G). Let x and y denote a pair of vertices that are k -edge-connected in G' . Consider a set of k edge-disjoint paths between x and y . Let l denote the number of these paths that contain one of the edges between u and w . Delete all but l of the edges between u and w in G' to obtain a graph G'' . Since x and y are k -edge-connected in G'' , the k -bond is immersed in it. If $l \leq m_2 + m_3$, then $G'' \leq_i G$, and therefore, the k -bond must be immersed in G .

So suppose $l > m_2 + m_3$. Since there are at most $m_1 + m_3$ edges between u and w in G' , it is also true that $l \leq m_1 + m_3$. Consider the l edge-disjoint paths in G'' between x and y that contain one of the edges between u and w . Each of these paths can be decomposed into three subpaths: a path from x to $\{u, w\}$, an edge between u and w , and a path from $\{u, w\}$ to y (if $x \in \{u, w\}$ or $y \in \{u, w\}$, one or two of these subpaths will be empty; see Figure 4.15 for the various possibilities). Thus in G'' , there are l paths between $\{u, w\}$ and x , and l paths between $\{u, w\}$ and y , all of which are mutually edge-disjoint. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ denote these two sets of paths. Since x and y are k -edge-connected, there are $k - l$ paths between them that are edge-disjoint with all the paths in $\mathcal{P}_1$ and $\mathcal{P}_2$. Let $\mathcal{P}_3$ denote this set of paths. Then we may

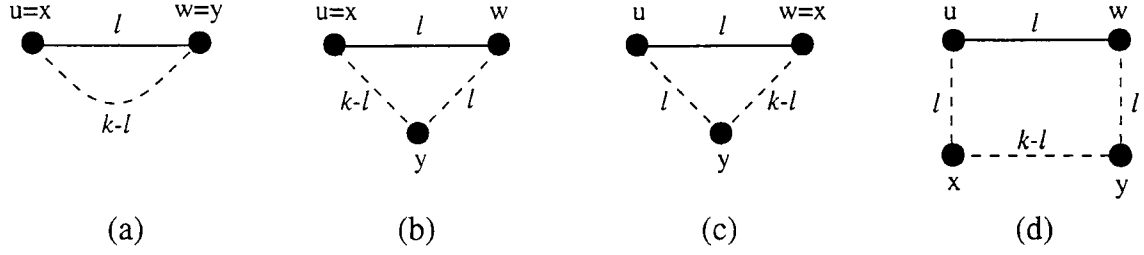


Figure 4.15: Possible k -bond models in G' .

obtain a path between u and w by composing a path from $\mathcal{P}_1$, one from $\mathcal{P}_3$, and one from $\mathcal{P}_2$. This implies that there are at least $\text{minimum}\{l, k-l\}$ edge-disjoint paths between u and w in G'' , none of which contains the edges between u and w . All of these paths exist in $G - \{v\}$, in addition to the m_3 edges between u and w . The total number of paths between u and w in $G - \{v\}$ is at least $m_3 + \text{minimum}\{l, k-l\} \geq \text{minimum}\{m_3 + (m_2 + m_3), m_3 + k - (m_1 + m_3)\} \geq \text{minimum}\{m_2, k - m_1\} = k - m_1$. Each of these paths can be composed with an edge between w and v to form a path between u and v . There are at least $\text{minimum}\{m_2, k - m_1\} = k - m_1$ such paths between u and v in G . Since u and v are adjacent by m_1 edges in G , the total number of edge-disjoint paths between u and v in G is at least $m_1 + k - m_1 = k$. Thus the k -bond is immersed in G . ■

Theorem 4.3 Given a biconnected series-parallel graph G of order two or more and a positive integer k as input, **kbond-test** correctly detects if the k -bond is immersed in G .

Proof Let v, u , and w be as defined in **kbond-test**. If $\delta(v) < k$, then u and v are identified. By Lemma 4.18, the k -bond is immersed in the resulting graph if and only if it is immersed in G . If $\delta(v) \geq k$, then v and w are identified. By Lemmas 4.19 and 4.20, the k -bond is immersed in the resulting graph if and only if it is immersed in G . To complete the proof, we note that if G has order two, then the k -bond is immersed in G if and only if the two vertices in G are adjacent by k or more edges; that is, if

and only if $|E(G)| \geq k$. ■

We represent G in doubly-linked adjacency list format. Vertices to play the role of v can be maintained using a queue. Enqueueing and dequeueing takes only constant time. Identification of two vertices is equivalent to merging their lists of neighbors and then deleting one of the two vertices. This can be accomplished in constant time as well. Thus **kbond-test** consists of a linear number of constant-time steps, and consequently, works in linear time.

4.3 Finding k -edge-connected Sets

A k -edge-connected set in a graph G is a maximal set of vertices that are pairwise k -edge-connected. The set of all k -edge-connected sets in G is denoted $EC(k, G)$. Given the k -edge-connected sets in each biconnected component of G , it is easy to build $EC(k, G)$. One merely needs iteratively to replace pairs of sets containing the same cut point with their union, until all sets are disjoint. We present a procedure, **find-ec**, to construct $EC(k, G)$, when G is biconnected and series-parallel. Procedure **find-ec** is based on the following lemma.

Lemma 4.21 Let v denote a vertex in a graph G , with exactly two neighbors u and w . Suppose the number of edges between u and v is no less than the number between v and w . Let x denote an arbitrary vertex in G . For any k , if v and x are k -edge-connected, then so are v and u .

Proof Consider a set of k edge-disjoint paths between x and v . Let l denote the number of these that contain one of the edges between u and v . Then at least $k - l$ of the paths contain one of the edges between w and v . Let m_1 denote the number of edges between u and v , and m_2 the number of edges between v and w . The following relations hold: $m_1 \geq l$, $m_2 \geq k - l$ and $m_1 \geq k - l$. This implies that the number of edge-disjoint paths between v and u is at least $m_1 + \text{minimum}\{m_2, k - l, l\} =$

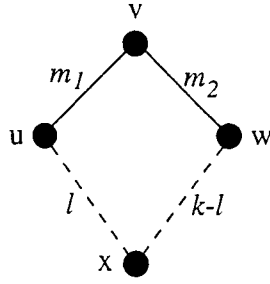


Figure 4.16: Edge-disjoint paths between u and v .

$m_1 + \text{minimum}\{k-l, l\} \geq \text{minimum}\{l+k-l, k-l+l\} = k$. (See Figure 4.16. Solid lines represent paths that are mere edges; paths that may contain more than one edge are shown as dashed lines.) Thus u and v are k -edge-connected. ■

The preceding result suggests a simple algorithm for determining $EC(k, G)$, that proceeds as follows: First find vertices to play the role of v , u and w . If u and v are not k -edge-connected, then short v . Otherwise, add v and all vertices known to be k -edge-connected to it, to the set of vertices known to be k -edge-connected to u , and then short v .

Of course this algorithm, though simple, has quadratic complexity, because it requires $n-1$ calls to a subroutine that checks if a given pair of vertices is k -edge-connected. We can improve on this using the following lemma.

Lemma 4.22 Let v denote a vertex in a graph G , with exactly two neighbors u and w . Let m_1 denote the number of edges between u and v , and m_2 the number of edges between v and w . Let $l = k + m_2 - m_1$, where k is an arbitrary positive integer. Then u and v are k -edge-connected if and only if $\delta(v) \geq k$ and either $m_1 \geq k$ or u and w are l -edge-connected.

Proof Suppose u and v are k -edge-connected. Then $\delta(v) \geq k$. If $m_1 < k$, consider a set of k edge-disjoint paths between u and v . Of these, at least $k - m_1$ contain one of the edges between v and w . Thus there are $k - m_1$ or more edge-disjoint paths between u and w in $G - \{v\}$. These paths, along with the m_2 paths between u and

w obtained by concatenating an edge between u and v and an edge between v and w , give us $k + m_2 - m_1 = l$ edge-disjoint paths between u and w .

Conversely, suppose $\delta(v) \geq k$ and either $m_1 \geq k$ or u and w are l -edge-connected. Of course, if $m_1 \geq k$, then u and v are k -edge-connected. Otherwise, of the l or more edge-disjoint paths between u and w , at most m_2 can contain v as an intermediate vertex. So there are at least $k - m_1$ edge-disjoint paths between u and w that do not contain v . Concatenating any of these paths with an edge between w and v gives us a path between u and v . Since $\delta(v) \geq k$, $m_2 = \delta(v) - m_1 \geq k - m_1$. Therefore, there are at least $\text{minimum}\{m_2, k - m_1\} = k - m_1$ such paths between u and v . These paths, along with the m_1 edges between u and v , give us k edge-disjoint paths between u and v . ■

Thus the problem of deciding if u and v are k -edge-connected reduces to that of deciding if u and w are l -edge-connected. Our algorithm to construct $EC(k, G)$, using the results of Lemmas 4.21 and 4.22, is called **find-ec**. In addition to taking G and k as input, it assumes that $EC(2, G), \dots, EC(k - 1, G)$ are available. These sets are required in order to make use of the result of Lemma 4.22. In practice, each $EC(i, G)$ for $2 \leq i \leq k - 1$ would itself be obtained using **find-ec**. Since G is biconnected, $EC(2, G) = \{V(G)\}$. Now $EC(3, G)$ can be constructed with a call to **find-ec**($G, 3$). Once $EC(3, G)$ is known, $EC(4, G)$ can be constructed with another call, and so on. In order to construct $EC(k, G)$, given G and k , $k - 2$ calls are required.

4.3.1 Correctness and Complexity

We address the correctness and complexity of **find-ec**.

Theorem 4.4 Let G denote a biconnected series-parallel graph of order two or more and let $k \geq 3$ be an integer. Then the call to **find-ec**(G, k) returns the set of k -edge-connected sets in G .

Procedure find-ec

```

Input:   A biconnected series-parallel graph  $G$  of order two or more and  $k \geq 3$ .
Output: The set  $S$ , where  $S = EC(k, G)$ .
begin procedure
  if  $|V(G)| = 2$ 
    then begin
       $x, y :=$  the two vertices in  $G$ ;
      if  $|E(G)| \geq k$  then  $S := \{\{x, y\}\}$ ; else  $S := \{\{x\}, \{y\}\}$ ;
      end
    else begin
       $v :=$  a vertex in  $G$  with exactly two neighbors;
       $u, w :=$  the neighbors of  $v$ , with the number of edges between  $u$  and  $v$ 
        no less than the number of edges between  $v$  and  $w$ ;
       $m_1 :=$  the number of edges between  $u$  and  $v$ ;
       $m_2 :=$  the number of edges between  $v$  and  $w$ ;
       $l := k + m_2 - m_1$ ;
       $S' := \text{find-ec}(G', k)$ , where  $G'$  is the graph obtained by shorting  $v$ ;
       $S'(u) :=$  the set in  $S'$  that contains  $u$ ;
      if  $\delta_G(v) \geq k$  and ( $k \leq m_1$  or  $u$  and  $w$  are  $l$ -edge-connected )
        then  $S(u) := S'(u) \cup \{v\}$ ;
       $S := S' - \{S'(u)\} \cup S(u)$ ;
      end
    output  $S$ ;
  end procedure

```

Proof The proof proceeds by induction on the order of G . In the basis case, when G contains just two vertices, it is clear from the description of **find-ec** that the theorem holds.

So suppose G has order three or more. Let $v, u, w, l, m_1, G', S', S'(u), S(u)$ and S be as defined in **find-ec**. Observation 4.2 guarantees that a vertex to play the role of v exists in G . By the induction hypothesis, S' is the set of k -edge-connected sets in G' . Since shorting does not change the number of edge-disjoint paths between remaining vertices, two vertices in G' are j -edge-connected, for any j , if and only if they are j -edge-connected in G . In particular, this implies that all sets in S' , with one possible exception, are valid k -edge-connected sets in G . The one exception is due to the fact that vertex v may need to be added to one of the sets in S' . It follows from Lemma 4.21 that the only set to which v can be added is $S'(u)$, the set that contains

u . By Lemma 4.22, v is k -edge-connected to u if and only if $\delta(v) \geq k$, and either $k \leq m_1$ or u and w are l -edge-connected. Thus v is added to $S(u)$ in **find-ec** if and only if u and v are k -edge-connected, implying that S is the set of k -edge-connected sets in G . ■

To determine the complexity of **find-ec**, we examine the operations performed in each call. By storing G in doubly-linked adjacency list format and employing a queue to maintain candidates for shorting, v can be found and shorted in constant time. Finding the set $S'(u)$ containing u and adding v to $S'(u)$ can be performed in constant time by representing the set of k -edge-connected sets as an array. The array is indexed by the vertex set of G . All entries are set to zero initially, and changed as vertices are shorted. If u and v are found to be k -edge-connected, then v 's entry is set equal to that of u , indicating that u and v belong to the same k -edge-connected set. Otherwise, v 's entry is set equal to one more than the maximum of all current entries, indicating that v belongs to a k -edge-connected set by itself. Note that there is no need to scan the array to find the maximum, as it can be stored separately and updated. Finally, it may be necessary to check if u and w are l -edge-connected. This can be done in constant time by storing the set of all l -edge-connected sets as an array.

Since the order of G decreases by one after each call to **find-ec**, there are at most a linear number of calls. So, given $EC(2, G), \dots, EC(k-1, G)$, a linear number of constant-time steps suffices to construct $EC(k, G)$. Of course, if only G and k are given, we must iteratively construct $EC(2, G), \dots, EC(k-1, G)$. Thus we obtain the following result.

Theorem 4.5 If G is a biconnected series-parallel graph of order n , and $k \geq 3$, then $EC(k, G)$ can be constructed in $O(kn)$ time.

4.4 Discussion

Fast immersion tests are of interest, because some problems that arise in practice may be amenable to an approach that tests for the presence of immersed forbidden graphs. Examples include cutwidth and its generalizations, and the partitioning problems discussed in the previous chapter. But there is one other issue that must be addressed before this approach can be truly practical. For most immersion-closed problems, the number of obstructions explodes dramatically, as the values of input parameters are increased. Thus, even if each individual test is fast, any algorithm that tests for the presence of every graph in the forbidden set is bound to be infeasible, except perhaps for small parameter values.

One possible escape route is to abandon the idea of exhaustive testing. Instead, one might focus on testing for a few “critical” obstructions, which would be most likely to occur in practice. Of course, an algorithm of this type is necessarily inaccurate, in the sense that its output is sometimes incorrect. Experiments using such an algorithm, albeit on a problem that is closed in a different order, indicates that this approach may have promise [GLR]. The evidence is far from conclusive, however; in particular, not much is known about critical obstructions for immersion-closed problems.

Another idea is to test for several obstructions as a unit, rather than separately. For instance, the Hopcroft-Tarjan planarity test [HT] can be viewed as checking simultaneously for the presence of a topological K_5 or $K_{3,3}$ [Wil]. Checking for the presence of each graph separately turns out to be a much harder task. It is not known whether there are immersion obstructions to practical problems that are easier to test together than apart.

Chapter 5

Conclusions

5.1 Summary of Results

We have devised approximation algorithms to determine the pathwidth of several classes of graphs. For k -chordal, outerplanar and series-parallel graphs, we described how to construct tree decompositions that can be converted into path decompositions of small width. The path decompositions produced are guaranteed to have width no more than k , three and six times the optimal, for each respective class. We then showed that path decompositions can be obtained not just from tree decompositions, but from decompositions whose underlying graphs may be structurally more complex than trees. This was demonstrated by an approximation algorithm for the pathwidth of Halin graphs, that makes use of an outerplanar graph decomposition. This algorithm has a performance ratio of eighteen. We noted that graph decompositions can also be used to construct tree decompositions and to bound the treewidth of certain classes of graphs.

We have also examined graph partitioning problems that are relevant to layout using multiple FPGAs. We analyzed the complexity of partitioning variants involving degree constraints, which model the pin-count limitations of FPGAs. We showed that these variants, with one exception, are $\mathcal{NP}$ -hard. For the lone “easy” problem, we

developed a polynomial-time algorithm based on network flow. We then considered the fixed-parameter versions of the problems, and found that they are closed under the immersion order. Since closed problems can be solved by testing for the presence of obstructions, we undertook a preliminary investigation of the obstruction sets. We found that the obstruction sets explode in size as parameter values are increased.

A possible approach to tackling immersion-closed problems is to test for a few selected obstructions. This approach requires fast tests for particular obstructions. Accordingly, we devised two practical immersion tests. Our test for K_4 can detect the presence of an immersed K_4 , and find a model if any exist, in linear time. Our test for the k -bond detects its presence in series-parallel graphs in linear time, but cannot find a model. We also devised an algorithm to find k -edge-connected sets in series-parallel graphs in $O(kn)$ time, where n denotes the order of the input graph. This algorithm may be useful as a preprocessing step for fast immersion tests.

5.2 Open Problems

We conjecture that our method of deriving path decompositions of small width from tree decompositions can be extended to all graphs of bounded treewidth. At the moment, the question of whether this is possible, let alone practical, is unanswered. An answer in the affirmative could lead to practical relative approximation algorithms for graphs of treewidth three and four, k -outerplanar graphs, graphs of disk-dimension four and several other classes for which bounded-width tree decompositions can be constructed efficiently. It may also be possible to use the notion of graph decompositions to construct tree decompositions of small width. This issue needs further study. A long-standing open problem concerns the existence of a polynomial-time relative approximation algorithm for the pathwidth of planar graphs. Since planar graphs have unbounded treewidth, constructing their tree decompositions and converting these into path decompositions will not, in general, produce a relative approxima-

tion. Nevertheless, our approach may be viable for this class of graphs, if graph decompositions are used instead of tree decompositions.

The partitioning and cutting problems we have discussed are in an early stage of investigation. Although a few heuristics have been designed for them, they have not been studied from a theoretical perspective in any great detail. Multiway Cut is an exception. Efficient algorithms on planar graphs and an approximation algorithm on general graphs that achieves a performance ratio of two are known [DJPSY]. For Minimum Degree Cut, we presented an $O(n^5)$ exact algorithm that is based on network flow. Recent advances in network flow algorithms [GR] suggest that this estimate of the complexity is pessimistic. Beyond improvements provided by faster flow algorithms, we suspect that it is possible substantially to reduce the number of calls made by our algorithm to a flow routine. It is not known whether efficient approximations are possible for the remaining problems. The behavior of these problems on many interesting classes of graphs is also unknown. Some preliminary results are available for trees [Pla], but not for more complicated classes, such as planar graphs or graphs of bounded treewidth. Since the fixed-parameter versions are closed under immersion, it seems worthwhile to explore the obstruction sets, attempt to identify critical obstructions, and design fast tests for them. Although it is as yet unclear whether this approach can ever be made practical, it cannot be ruled out without further knowledge about the obstructions. An alternative to the rather ad hoc method of designing fast immersion tests for particular graphs is to devise a faster general immersion test. This looks to be a daunting task, however, because of the intimate connection between the immersion order and disjoint paths, which is a notoriously hard problem.

Bibliography

Bibliography

- [ACP] S. Arnborg, D. G. Corneil and A. Proskurowski, “Complexity of Finding Embeddings in a k -tree,” *SIAM Journal of Algebraic and Discrete Methods* 8 (1987), 277–284.
- [AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1974.
- [ALS] S. Arnborg, J. Lagergren and D. Seese, “Easy Problems for Tree-Decomposable Graphs,” *Journal of Algorithms* 8 (1991), 308–340.
- [BFL] D. J. Brown, M. R. Fellows and M. A. Langston. “Polynomial-Time Self-Reducibility: Theoretical Motivations and Practical Results,” *International Journal of Computer Mathematics* 31 (1989), 1–9.
- [BGHK] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson and T. Kloks, “Approximating Treewidth, Pathwidth, Frontsize and Shortest Elimination Tree,” *Journal of Algorithms* 18 (1995), 238–255.
- [BGLR] H. D. Booth, R. Govindan, M. A. Langston and S. Ramachandramurthi, “Fast Algorithms for K_4 Immersion Testing,” Accepted for Publication in *Journal of Algorithms*. (A preliminary version of this paper was released as a technical report.)

- [BK] H. L. Bodlaender and T. Kloks, "Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs," *Journal of Algorithms* 25 (1996), 358–402.
- [Bod1] H. L. Bodlaender, "Some Classes of Graphs with Bounded Treewidth," *Bulletin of the EATCS* 36 (1988), 116–126.
- [Bod2] H. L. Bodlaender, "Dynamic Programming on Graphs with Bounded Treewidth," *Lecture Notes in Computer Science* 317 (1988), 105–118.
- [Bod3] H. L. Bodlaender, "A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth," *Proceedings, 25th Annual ACM Symposium on Theory of Computing* (1993), 226–234.
- [Bol] B. Bollobás, *Extremal Graph Theory*, Academic Press, New York, 1978.
- [Bon] J. A. Bondy, "Pancyclic graphs: recent results," in *Infinite and Finite Sets: to Paul Erdős on his 60th birthday*, A. Hajnal, R. Rado and V. T. Sós, Editors, North-Holland Publishing Company, Amsterdam (1975), 181–187.
- [BR] J. S. Blair and S. S. Ravi, "On Finding Maximum Transmitter-Receiver Matchings in Chordal Networks," *Proceedings, 27th Annual Allerton Conference on Communication, Control and Computing* (1989), 72–81.
- [CDF] K. Cattell, M. J. Dinneen and M. R. Fellows, "A Simple Linear Time Algorithm for Finding Path Decompositions of Small Width," *Information Processing Letters* 57 (1996), 197–203.
- [CLCDL] Nan-Chi Chou, Lung-Tien Liu, Chung-Kuan Cheng, Wei-Jin Dai and Rodney Lindelof, "Local Ratio Cut and Set Covering Partitioning for Huge Logic Emulation Systems," *IEEE Transactions on CAD* 14:9 (1995), 1085–1091.

- [CNSD] H. Cai, S. Note, P. Six and H. De Man, "A Data Path Layout Assembler for High Performance DSP Circuits," *Proceedings, 27th Design Automation Conference* (1990), 306-311.
- [DeM] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, 1994.
- [DJPSY] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour and M. Yannakakis, "The Complexity of Multiway Cuts," *Proceedings, 24th Annual ACM Symposium on Theory of Computing* (1992), 241-251.
- [DKL] N. Deo, M. S. Krishnamoorthy and M. A. Langston, "Exact and Approximate Solutions for the Gate Matrix Layout Problem," *IEEE Transactions on CAD* 6 (1987), 79-84.
- [Duf] R. J. Duffin, "Topology of Series-Parallel Networks," *Journal of Mathematical Analysis and Applications* 10 (1965), 303-318.
- [EST] J. A. Ellis, I. H. Sudborough and J. S. Turner, "The Vertex Separation and Search Number of a Graph," *Information and Computation* 113 (1994), 50-79.
- [ET] S. Even and R. E. Tarjan, "Computing an *st*-numbering," *Theoretical Computer Science* 2 (1976), 339-344.
- [FF] L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1974.
- [FG] D. R. Fulkerson and O. A. Gross, "Incidence Matrices and Interval Graphs," *Pacific Journal of Mathematics* 15 (1964), 539-548.
- [FHKY] T. Fujii, H. Horikawa, T. Kikuno and N. Yoshida, "A Heuristic Algorithm for Gate Assignment in One-Dimensional Array Approach," *IEEE Transactions on CAD* 6 (1987), 159-164.

- [FL1] M. R. Fellows and M. A. Langston, "Nonconstructive Advances in Polynomial-Time Complexity," *Information Processing Letters* 26 (1987/88), 157-162.
- [FL2] M. R. Fellows and M. A. Langston, "Nonconstructive Tools for Proving Polynomial-Time Decidability," *Journal of the ACM* 35:3 (1988), 727-739.
- [FL3] M. R. Fellows and M. A. Langston, "An Analogue of the Myhill-Nerode Theorem and its Use in Computing Finite-Basis Characterizations," *Proceedings, 30th Annual Symposium on Foundations of Computer Science* (1989), 520-525.
- [FL4] M. R. Fellows and M. A. Langston, "Fast Search Algorithms for Layout Permutation Problems," *International Journal of Computer Aided VLSI Design* 3 (1991), 325-341.
- [FL5] M. R. Fellows and M. A. Langston, "On Well-Partial-Order Theory and its Application to Combinatorial Problems of VLSI Design," *SIAM Journal on Discrete Mathematics* 5:1 (1992), 117-126.
- [FL6] M. R. Fellows and M. A. Langston, "On Search, Decision and the Efficiency of Polynomial-Time Algorithms," *Journal of Computer and System Sciences* (to appear). (A preliminary version of this paper can be found in *Proceedings, 21st Annual ACM Symposium on Theory of Computing* (1989), 501-512.)
- [FRT] D. Fussell, V. Ramachandran, R. Thurimella, "Finding Triconnected Components by Local Replacements," *Proceedings, 16th International Colloquium on Automata, Languages and Programming* 372 (1989), 379-393.
- [GJ] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$ -Completeness*, W. H. Freeman and Company, New York, 1979.
- [GLR] R. Govindan, M. A. Langston and S. Ramachandramurthi, "A Practical Approach to Layout Optimization," *Proceedings of the Sixth International Conference on VLSI Design* (1993), 222-225.

- [GLY] R. Govindan, M. A. Langston and X. Yan, "Approximating the Pathwidth of Outerplanar Graphs," Accepted for Publication in *Information Processing Letters*.
- [GR] A. V. Goldberg and S. Rao, "Beyond the Flow Decomposition Barrier," *Proceedings, 38th Annual Symposium on Foundations of Computer Science* (1997), 2-11.
- [GT] A. V. Goldberg and R. E. Tarjan, "A New Approach to the Maximum-Flow Problem," *Journal of the ACM* 35:4 (1988), 921-940.
- [Guf] D. Gusfield, "Very Simple Methods for All Pairs Network Flow Analysis," *SIAM Journal on Computing* 19:1 (1990), 143-155.
- [Gus] J. Gustedt, "On the Pathwidth of Chordal Graphs," *Discrete Applied Mathematics* 45:3 (1993), 233-248.
- [Har] F. Harary, *Graph Theory*, Addison-Wesley Publishing Company, Reading, MA, 1969.
- [He] X. He, "Efficient Parallel Algorithms for Series Parallel Graphs," *Journal of Algorithms* 12 (1991), 409-430.
- [HP] F. Harary, E. M. Palmer, *Graphical Enumeration*, Academic Press, New York and London, 1973.
- [HPK] Y-S. Hong, K-H. Park and M. Kim, "Heuristic Algorithms for Ordering the Columns in One-Dimensional Logic Arrays," *IEEE Transactions on CAD* 8 (1989), 547-562.
- [HT] J. E. Hopcroft and R. E. Tarjan, "Efficient Planarity Testing," *Journal of the ACM* 21:4 (1974), 549-568.

- [KBK] R. Kuznar, F. Brglez and K. Kozminski, "Cost Minimization of Partitions into Multiple Devices," *Proceedings of the 30th ACM/IEEE DAC* (1993), 315–320.
- [KF] T. Kashiwabara and T. Fujisawa, " $\mathcal{NP}$ -Completeness of the Problem of Finding a Minimum-Clique-Number Interval Graph Containing a Given Graph as a Subgraph," *Proceedings of the International Symposium on Circuits and Systems* (1979), 657–660.
- [Kin1] N. G. Kinnersley, "The Vertex Separation of a Graph Equals its Path-Width," *Information Processing Letters* 42 (1992), 345–350.
- [Kin2] N. G. Kinnersley, "Immersion Order Obstruction Sets," Technical Report TR-93-3, Department of Computer Science, University of Kansas, Lawrence, KS, 1993.
- [KK] N. G. Kinnersley and W. M. Kinnersley, "An Efficient Polynomial-Time Algorithm for Three-Track Gate Matrix Layout," *The Computer Journal* 37:5 (1997), 449–462.
- [KeL] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal* 49 (1970), 291–307.
- [KiL] N. G. Kinnersley and M. A. Langston, "Obstruction Set Isolation for the Gate Matrix Layout Problem," *Discrete Applied Mathematics* 54 (1994), 169–213.
- [Klo] T. Kloks, "Treewidth," Ph. D. Thesis, University of Utrecht, The Netherlands, 1993.
- [KoT] A. Kornai and Z. Tuza, "Narrowness, Pathwidth, and their Application in Natural Language Processing," *Discrete Applied Mathematics* 36 (1992), 87–92.
- [KP] L. M. Kirousis and C. H. Papadimitriou, "Interval Graphs and Searching," *Discrete Mathematics* 55 (1985), 181–184.

- [KS] S. Khuller and B. Schieber, "Efficient Parallel Algorithms for Testing k -connectivity and Finding Disjoint s - t Paths in Graphs, " *SIAM Journal on Computing* 20:4 (1991), 352-375.
- [KT] E. Korach and A. Tal, "General Vertex Disjoint Paths in Series-Parallel Graphs," *Discrete Applied Mathematics* 41 (1993), 147-164.
- [Lan] M. A. Langston, "WQO-Based Methods," *International Workshop on Combinatorial Methods for Circuit Design*, Schloss Dagstuhl, Germany, October, 1993.
- [Len] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, Chichester, England, 1990.
- [LG] P. C. Liu and R. C. Geldmacher, "An $O(\max(m, n))$ Algorithm for Finding a Subgraph Homeomorphic to K_4 ," *Congressus Numerantium* 29 (1980), 597-609.
- [Li] J-T. Li, "Algorithms for Gate Matrix Layout," *Proceedings, International Symposium on Circuits and Systems* (1983), 1013-1016.
- [LP] M. A. Langston and B. C. Plaut, "On Algorithmic Applications of the Immersion Order," *Discrete Mathematics* (to appear).
- [LR] M. A. Langston and S. Ramachandramurthi, "Dense Layouts for Series-Parallel Circuits," *Proceedings, First Great Lakes Symposium on VLSI* (1991), 14-17.
- [Mad] W. Mader, "Homomorphiesätze für Graphen," *Mathematische Annalen* 178 (1968), 154-168.
- [MaS] F. S. Makedon and I. H. Sudborough, "On Minimizing Width in Linear Layouts," *Discrete Applied Mathematics* 23 (1989), 243-265.

- [Möh] R. H. Möhring, "Graph Problems Related to Gate Matrix Layout and PLA Folding," *Computational Graph Theory, Computing Supplementum 7* (1990) 17–51.
- [MoS] B. Monien and I. H. Sudborough, "Min-Cut is $\mathcal{NP}$ -Complete for Edge Weighted Trees," *Lecture Notes in Computer Science* 226 (1986), 265–274.
- [MPM] V. M. Malhotra, M. Pramodh-Kumar and S. N. Maheshwari, "An $O(V^3)$ Algorithm for Finding Maximum Flows in Networks," *Information Processing Letters* 7 (1978), 277–278.
- [Nas] C. Nash-Williams, "On Well-Quasi-Ordering Infinite Trees," *Proceedings, Cambridge Philosophical Society* 61 (1965) 697–720.
- [Pla] B. C. Plaut, private communication.
- [Rac] S. Ramachandramurthi, "Algorithms for VLSI Layout Based on Graph Width Metrics," Ph. D. Thesis, Department of Computer Science, University of Tennessee, Knoxville, TN, 1994.
- [Ram] V. Ramachandran, private communication.
- [RS1] N. Robertson and P. D. Seymour, "Graph Minors I. Excluding a Forest," *Journal of Combinatorial Theory, Series B* 35 (1983), 39–61.
- [RS2] N. Robertson and P. D. Seymour, "Graph Minors II. Algorithmic Aspects of Treewidth," *Journal of Algorithms* 7 (1986), 309–322.
- [RS4] N. Robertson and P. D. Seymour, "Graph Minors IV. Tree-Width and Well-Quasi-Ordering," *Journal of Combinatorial Theory, Series B* 48 (1990), 227–254.
- [RS5] N. Robertson and P. D. Seymour, "Graph Minors V. Excluding a Planar Graph," *Journal of Combinatorial Theory, Series B* 41 (1986), 92–114.

- [RS13] N. Robertson and P. D. Seymour, "Graph Minors XIII. The Disjoint Paths Problem," *Journal of Combinatorial Theory, Series B* 63 (1995), 65–110.
- [RS20] N. Robertson and P. D. Seymour, "Graph Minors XX. Wagner's Conjecture," manuscript (1996).
- [RS23] N. Robertson and P. D. Seymour, "Graph Minors XXIII. Nash-Williams' Conjecture," manuscript (1996).
- [RT] D. J. Rose and R. E. Tarjan, "Algorithmic Aspects of Vertex Elimination," *Proceedings, 7th Annual ACM Symposium on Theory of Computing* (1975), 245–254.
- [Ros] H. E. Rose, *A Course in Number Theory*, second edition, Clarendon Press, Oxford, 1994.
- [VTL] J. Valdes, R.E. Tarjan and E. Lawler, "The Recognition of Series-Parallel Digraphs," *SIAM Journal on Computing* 11 (1982), 298–313.
- [WHW] O. Wing, S. Huang and R. Wang, "Gate Matrix Layout," *IEEE Transactions on CAD* 4 (1985), 220–231 .
- [WK] N. Woo and J. Kim, "An Efficient Method of Partitioning Circuits for Multiple-FPGA Implementation," 30th *ACM/IEEE Design Automation Conference* (1993), 202–207.
- [Wil] S. G. Williamson, "Depth-First Search and Kuratowski Subgraphs," *Journal of the ACM* 31:4 (1984), 681–693.
- [Wim] T. V. Wimer, "Linear Algorithms on k -terminal Graphs," Ph. D. Thesis, Department of Computer Science, Clemson University, Clemson, SA, 1987.
- [Yan] X. Yan, "A Relative Approximation Algorithm for Computing Pathwidth," Master's Thesis, Department of Computer Science, Washington State University, Pullman, WA, 1989.

Appendix

Glossary

biconnected component: maximal biconnected subgraph

biconnected graph: graph that has no cut points

chordal graph: graph in which every induced cycle of length four or more contains a chord

clique: completely connected graph

clique number: order of the largest clique subgraph of a graph

connected component: maximal connected subgraph

corner: image of a vertex under a mapping that demonstrates immersion containment

cut point: vertex whose deletion disconnects a graph

cutwidth (of graph G): minimum, over all linear layouts of G , of the maximum number of edges cut by a line drawn between consecutive vertices; denoted $cw(G)$

degree (of vertex v): number of edges incident on v ; denoted $\delta(v)$

degree (of vertex set S): number of edges with exactly one endpoint in S ; denoted $\delta(S)$

edge-connectivity (of a pair of vertices): minimum number of edges that must be deleted to disconnect the two vertices

edge-connectivity (of a graph): minimum number of edges that must be deleted to disconnect the graph

edge covering property: property of path and tree decompositions that ensures that the endpoints of every edge are contained in some subset

edge-disjoint paths: paths that do not share any edges

FPGA minimization: partitioning problem for which the input is a graph G and integers d , k and p , and the question to be answered is whether G has a partition into p or fewer subsets of degree at most d and size at most k

graph decomposition (of graph G): a pair (H, Y) , where H is a graph and $Y = \{Y_i \mid i \in V(H)\}$ is a collection of subsets of $V(G)$ such that

- for each edge $e \in E(G)$, some Y_i contains both endpoints of e , and
- for all $i, j, k \in V(H)$, the set $\{Y_i \mid v \in Y_i\}$ is either singleton or induces a connected subgraph of H

grid graph: simple graph G with $V(G) = \{(0, 0), (0, 1), \dots, (m - 1, n - 1)\}$ and $E(G) = \{(i, j), (k, l)\} \mid (i = k \text{ and } |j - l| = 1) \text{ or } (j = l \text{ and } |i - k| = 1)\}$, where m and n are positive integers

halin graph: planar graph obtained by taking a planar embedding of a tree with no degree-two vertices and connecting all leaf nodes in a cycle

immersion: graph H is immersed in graph G (denoted $H \leq_i G$) if H can be obtained from G by deleting vertices and edges, and lifting pairs of adjacent edges. Equivalently, H is immersed in G if there exists an injection from $V(H)$ to $V(G)$

such that the images of adjacent vertices in H are connected by edge-disjoint paths in G

intersection property: property of path, tree or graph decompositions that ensures that the set of all subsets containing a particular vertex induces a connected subgraph of the underlying graph of the decomposition

internal edge: in an outerplanar layout, an edge that is not on the external face

internal neighbors: vertices that are adjacent by an internal edge

K_4 : complete graph on four vertices

k -bond: graph consisting of two vertices that are adjacent by k edges

k -edge-connected set: maximal set of vertices that are pairwise k -edge-connected

lifting: operation of replacing a pair of edges that share an endpoint by an edge between the endpoints that are not common to the pair

$mdcsb(G, p)$: the minimum, over all partitions of graph G into p or fewer subsets, of the maximum degree of a subset

mine graph: graph consisting of a central clique of order k , with a pendant vertex attached to each vertex in the clique by an edge of multiplicity m

minimum degree cut (MDC): partitioning problem for which the input is a graph G , some of whose vertices are distinguished, and an integer d , and the question to be answered is whether G has a partition into subsets of degree at most d

minimum degree cut with subset bound (MDCSB): partitioning problem for which the input is a graph G , some of whose vertices are distinguished, and integers d and p , and the question to be answered is whether G has a partition into p or fewer subsets of degree at most d

minimum degree graph partition (MDGP): partitioning problem for which the input is a graph G and integers d and k , and the question to be answered is whether G has a partition into subsets of degree at most d and size at most k

model: minimal subgraph whose vertices and paths are images of vertices and edges under a mapping that demonstrates either immersion or topological containment

multiway cut: partitioning problem for which the input is a graph G , some of whose vertices are distinguished, and an integer d , and the question to be answered is whether G has a partition in which the total number of edges between subsets is at most d

$mwc(G)$: the minimum, over all partitions of graph G , of the number of edges between subsets

obstruction: minimal forbidden graph

outerplanar decomposition: graph decomposition whose underlying graph is outerplanar

outerplanar graph: graph that has a planar embedding with all vertices lying on a single face

parallel composition: combination of two graphs by identifying source with source and sink with sink

parallel node: node x in series-parallel decomposition tree D is a parallel node if $G(D_x)$ is the result of a parallel composition

partition (of graph G): collection of disjoint subsets of $V(G)$ whose union is $V(G)$, such that each subset contains at most one distinguished vertex

path decomposition (of graph G): a sequence $X_1, \dots, X_r$ of subsets of $V(G)$ such that

- for each edge $e \in E(G)$, some X_i contains both endpoints of e , and
- for $1 \leq i \leq j \leq k \leq r$, $X_i \cap X_k \subseteq X_j$

pathwidth (of graph G): smallest width of all path decompositions of G ; denoted $pw(G)$

perfect elimination ordering (of graph G): an ordering $(v_1, \dots, v_n)$ of $V(G)$ such that, for $1 \leq i \leq n$, v_i is a simplicial vertex in the subgraph of G induced by $\{v_i, \dots, v_n\}$

series composition: combination of two graphs by identifying one's source with the other's sink

series node: node x in series-parallel decomposition tree D is a series node if $G(D_x)$ is the result of a series composition

series-parallel decomposition tree: directed binary tree that indicates how a 2TSP graph is built up using series and parallel compositions; the 2TSP graph encoded by series-parallel decomposition tree D is denoted $G(D)$

series-parallel graph: graph that contains no topological K_4 ; equivalently, graph whose biconnected components are 2TSP

series-tree (of series node x): subtree induced by x and all series nodes y such that there is a path consisting only of series nodes from x to y

simplicial vertex: vertex whose neighborhood induces a clique

topological containment: graph H is topologically contained in graph G (denoted $H \leq_t G$) if there exists an injection from $V(H)$ to $V(G)$ such that the images of adjacent vertices in H are connected by vertex-disjoint paths in G

tree decomposition (of graph G): a pair (T, Y) , where T is a tree and $Y = \{Y_i \mid i \in V(T)\}$ is a collection of subsets of $V(G)$ such that

- for each edge $e \in E(G)$, some Y_i contains both endpoints of e , and
- for all $i, j, k \in V(T)$, if j is on the path between i and k in T , $Y_i \cap Y_k \subseteq Y_j$

treewidth (of graph G): smallest width of all tree decompositions of G ; denoted $tw(G)$

two-terminal series-parallel (2TSP) graph: graph that can be obtained from copies of K_2 using series and parallel compositions

underlying graph: if (H, Y) is a graph decomposition, then H is the underlying graph of the decomposition

virtual edge: edge present in a three-edge-connected component, but not in the original graph

w -extensible tree decomposition (of path P with endpoints w and x): a width-two tree decomposition (T, Y) such that $T = P - \{w, x\}$ and for $i \in V(T)$, $Y_i = \{x, i, j\}$, where j is the neighbor of i on w 's side

weak edge covering property: property of graph decompositions that ensures that the endpoints of every edge are contained in either a single subset or adjacent subsets

width: one less than the cardinality of the largest subset in a path, tree or graph decomposition

Vita

Rajeev Govindan was born on November 9, 1966 in Trivandrum, India. After graduating from high school, he attended the Indian Institute of Technology at Madras, obtaining a Bachelor of Technology degree in computer science in 1988. His first year as a graduate student was spent at the Washington State University in Pullman, Washington. He transferred to the University of Tennessee in the fall of 1989, to pursue a doctorate in computer science.