Layered Graph Drawing

Matthew Suderman

School of Computer Science
McGill University, Montréal

October, 2005

A thesis submitted to McGill University in partial fulfilment
of the requirements of the degree of Doctor of Philosophy.

Your file Votre référence
ISBN: 978-0-494-21700-9
Our file Notre référence
ISBN: 978-0-494-21700-9

Contents

Abstract

A layered graph drawing is a two-dimensional drawing of a combinatorial graph in which the vertices lie on a given set of horizontal lines. Such drawings are used in application domains such as software engineering, bioinformatics, and VLSI design. In addition to being layered, drawings in these applications may also satisfy other constraints, for example bounds on the number of edge crossings. The problems related to obtaining these drawings are almost always $\mathcal{NP}$-hard, so, in this thesis, we investigate restricted versions of these problems in order to find efficient algorithmic solutions that can be used in practice.

As a first very drastic restriction, we consider layered drawings that are planar. Even with this restriction, however, the resulting problems can still be $\mathcal{NP}$-hard. In addition to proving one such hardness result, we do succeed in deriving efficient algorithms for two problems. In both cases, we correct previously published results that claimed extremely simple and efficient algorithmic solutions to these problems. Our solutions, though efficient as well, show that the truth about these problems is significantly more complex than the published results would suggest.

We also study non-planar layered drawings, particularly drawings obtained by crossing minimization and minimum planarization. Though the corresponding problems are $\mathcal{NP}$-hard, they become tractable when the value to be minimized is upper-bounded by a constant. This approach to obtaining tractable problems is formalized in a theory called parameterized complexity, and the resulting tractable problems and algorithmic solutions are said to be fixed-parameter tractable ($\mathcal{FPT}$). Though relatively new, this theory has attracted a rapidly growing body of theoretical results. Indeed, we derive original $\mathcal{FPT}$ algorithms with the best-known asymptotic running times for planarization in two layer drawings.

Because parameterized complexity is so new, little is known about its implications to the practice of graph drawing. Consequently, we have implemented a few $\mathcal{FPT}$ algorithms and compared them experimentally with previously implemented approaches, especially integer linear programming (ILP). Our experiments show that the performance of our $\mathcal{FPT}$ planarization algorithms are competitive with current ILP algorithms, but that, for crossing minimization, current ILP algorithms remain the clear winners.

Résumé

Le problème du dessin de graphe en couches consiste à dessiner dans le plan euclidien un graphe combinatoire de façon à ce que chaque sommet du graphe se retrouve sur l'une ou l'autre d'un ensemble spécifié de lignes horizontales. Ce problème a des applications dans des domaines tel le génie logiciel, la bio-informatique ou la conception VLSI. Outre la représentation par couches, certaines applications imposent parfois des contraintes supplémentaires au dessin du graphe, par exemple une limite au nombre d'intersections d'arêtes. Les problèmes de ce genre sont le plus souvent NP-difficiles; la présente thèse a pour objectif de trouver des solutions algorithmiques efficaces, et par conséquent utiles en pratique, à des instances restreintes de ces problèmes.

Nous examinons en premier lieu le problème du dessin en couches de graphes planaires. Même restreints à cette classe de graphes, les problèmes évoqués ci-haut demeurent souvent NP-difficiles. Nous établissons un tel résultat de NP-difficulté; nous donnons également des algorithmes efficaces pour deux autres problèmes, rectifiant ainsi des résultats publiés précédemment qui disaient fournir pour ces deux problèmes des solutions algorithmiques efficaces et élégantes. Nos solutions mettent en lumière une complexité de ces deux problèmes insoupçonnée desdites publications précédentes.

Nous examinons également le dessin en couches de graphes non planaires, en particulier les dessins obtenus par minimisation de planarisation ou d'intersections d'arêtes. Bien que les problèmes correspondants soient NP-difficiles, ils admettent des algorithmes efficaces (deviennent tractables) s'il existe une constante qui soit une borne supérieure à la valeur devant être minimisée. Cette approche génératrice de problèmes tractables découle du concept plus formel de la complexité paramétrée (FPT), et les solutions algorithmiques correspondantes sont dites 'tractables par paramètre fixe'. Un nombre grandissant de résultats théoriques sont publiés à propos de ce concept malgré qu'il soit relativement nouveau; quant à nous, nous présentons de nouveaux algorithmes pour la planarisation en deux couches d'un graphe, algorithmes au meilleur temps d'exécution asymptotique connu.

La complexité paramétrée est si récente que l'on sait peu sur son utilité pratique pour le dessin de graphes. Nous avons donc implémenté quelques algorithmes tractables par

paramètre fixe dans le but de les comparer à des stratégies existantes, en particulier la programmation linéaire en nombres entiers (ILP). Nos tests démontrent une certaine compétitivité entre les deux stratégies, cependant, pour ce qui est de la minimisation d'intersections d'arêtes, les algorithmes ILP existants demeurent nettement supérieurs.

Declaration

This thesis contains no material which has been accepted in whole, or in part, for any other degree or diploma. Except for results whose authors are cited where first mentioned, Chapters 3-8 constitute an original contribution to knowledge.

Assistance has been received only as mentioned in the following:

- The problem studied in Chapter 3 was originally discussed at the *2001 International Workshop on Graph Drawing and Fixed Parameter Tractability* organized by my supervisor Sue Whitesides; however, only minor progress was made toward solving the problem at the workshop.
- The work presented in Chapter 4 was initiated by David Wood when he suggested that I attempt to reconstruct a proof of a claimed result.
- Chapter 5 is joint work with Emilio Di Giacomo, Walter Didimo and Giuseppe Liotta.
- Chapter 8 is joint work with my supervisor although I was solely responsible for implementing the algorithms, running the experiments and handling the data.

An extended abstract of Chapter 4 appears in [89], the contents of Chapter 5 appear in [43], and preliminary results of Chapter 8 appear in [90].

Acknowledgements

I would first of all like to thank Sue Whitesides. Both personally and professionally, Sue has much to offer students and colleagues, and, as my supervisor, she has been very generous to me. Not only did she introduce me to the topics addressed in this thesis, but she gave me many opportunities to experience first-class research, by inviting me to her workshops, arranging research visits with her colleagues, and making it possible for me to attend conferences. Sue has always impressed me with her ability to perceive the deeper significance of ideas. More than once have I discovered what seemed to me to be a little curiosity, only to realize in a conversation with Sue that the curiosity was in fact the key to something surprisingly significant.

I would like to thank Thomas Shermer for teaching his graduate course on Graph Drawing. One of the four assignments in the course was, of all things, to implement the Hopcroft and Tarjan planarity testing algorithm! As difficult as the assignment was, it sparked my interest in drawing graphs and ultimately led to the completion of this thesis.

Thanks to my colleagues Vida Dujmović and David Wood for both calm and lively days spent proving theorems. David amazes me with his ability to comprehend the ideas that come tumbling out of my mouth and give them clear and concise expression. Vida ensures that problem sessions are stimulating, the way she pounces on ideas and instantly spots holes in "proofs."

I am especially grateful to Sue for introducing me to Giuseppe (Beppe) Liotta. Each week that I have spent visiting and working with Beppe, Emilio Di Giacomo and Walter Didimo in Perugia has been memorable and productive. As a result of the visits, I have come to consider all three of them as my friends.

Many of the ideas presented in this thesis are based on work by Michael Fellows and Frances Rosamond. I was fortunate enough to have Mike and Fran invite me to spend a month with them at the University of Newcastle. During that time, I experienced first-hand their lifestyle built around mathematics, computer science, ... and of course surfing. I also had the pleasure of meeting their students Peter Shaw and Elena Prieto and spending time with them trying to cover points with lines.

Thanks to Michel Langlois for translating my abstract into French. I definitely could not have done that without you!

I am grateful for financial support from a scholarship for doctoral studies from FCAR and from the grant of Sue Whitesides. I was not a starving student!

Thank you Dad for showing me how to work hard and to think deeply about things. Thank you Mom for passing your insatiable curiosity on to me.

And finally, to my wife Leslie, thanks for putting up with all the late nights of clickity-clack on the keyboard and reminding me when it was 3AM. You are my best friend. Let's have a picnic.

List of Figures

List of Tables

Chapter 1

Introduction

To Leslie—*you show me that sanity is not found in reasoning but in loving.*

A *combinatorial graph* consists of a set of elements called *vertices* and a set of connections between pairs of vertices called *edges*. In spite of this simple definition, combinatorial graphs are important theoretical tools for modelling complex types of data. For example, software engineers model large software systems as graphs whose vertices correspond to system components and whose edges correspond to component dependencies. They use these graph models to visualize and test the structural quality of systems (see, e.g. [65]). Another example is the World Wide Web (WWW), which has grown so large that new tools are needed to use it to find information efficiently. As a result, research into how to effectively visualize and navigate the structure of the World Wide Web has become a popular research topic (see, e.g. [35]), where the structure is often modelled as a graph whose vertices correspond to Web pages and whose edges correspond to hyperlinks between pages.

Once obtained, graph models are used for a variety of purposes that often require drawings of the graph satisfying certain physical constraints such as bounded area or volume, or a minimum distance between pairs of vertices. The discipline of Graph Drawing is concerned, in part, with all problems relating to mathematically characterizing those graphs that can be drawn subject to certain constraints, to finding efficient algorithms for automatically testing whether or not a given graph satisfies a given characterization and, if it does, to obtaining such a drawing.

In this thesis, we investigate *layered* or *hierarchical graph drawings*, that is drawings in the plane whose vertices are drawn on one or more horizontal lines called layers. Layered graph drawings were first introduced by Tomii, Kambayashi and Yajima [93], Carpano [11], and Sugiyama, Tagawa and Toda [91]. They are used in many applications [76] including visualization [6,60], DNA mapping [96], phylogenetic tree comparison [29], and row-based VLSI layout [86].

Layered drawings come in many different flavours. In nearly all of the drawings that we consider in this thesis, edges are drawn as straight-line segments between the vertices that

they connect. Within this broad category of drawings, we differentiate between planar and non-planar drawings. A drawing is a *planar* drawing if no pair of distinct edges intersects except at shared end-vertices; otherwise, the drawing is *non-planar*. For example, Figure 1.1 illustrates a planar and a non-planar drawing of the same graph. The only difference between the two drawings is the position of the darkened vertex and its incident edges, and the fact that the non-planar drawing uses fewer layers.

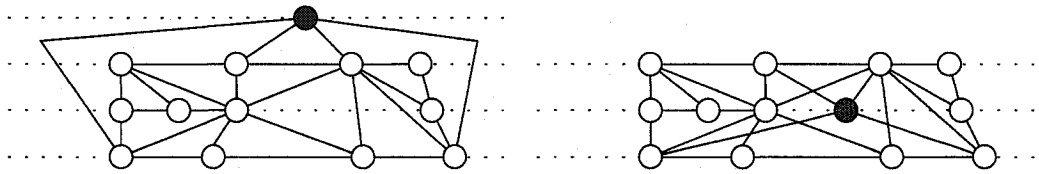


Figure 1.1: A planar layered drawing (a) and a non-planar layered drawing (b) of the same graph.

In the next two sections, we discuss layered drawings in more detail and define several problems related to obtaining these drawings.

1.1 Planar Drawings

We motivate the details related to planar layered drawings using an application from DNA mapping. For our purposes, a *DNA molecule* is simply composed of a long chain of nucleotide pairs. In DNA mapping, biologists use restriction enzymes to identify important locations in DNA molecules. A *restriction enzyme* interacts with a DNA molecule by cutting it into smaller molecule fragments at specific locations. To identify these locations, biologists obtain copies of a DNA molecule and two restriction enzymes. Let us suppose that one enzyme is denoted by A and and the other by B. They then use enzyme A to cut one of the molecule copies into fragments $a_1, a_2, \ldots, a_m$, and then use enzyme B to cut another copy of the molecule into fragments $b_1, b_2, \ldots, b_n$. Through further tests, biologists are able to discover which pairs of fragments contain the same parts of the original DNA molecule; in other words, they know which fragments *overlap*. From this information, they wish to determine the order of the cuts created by A and B along the molecule, or, equivalently, the order of the fragments as they appeared in the original molecule. For an example, see Figure 1.2.

Waterman and Griggs [96] reduce this problem to a layered graph drawing problem on two layers. They construct a graph model in which each fragment a_i, $1 \leq i \leq m$,

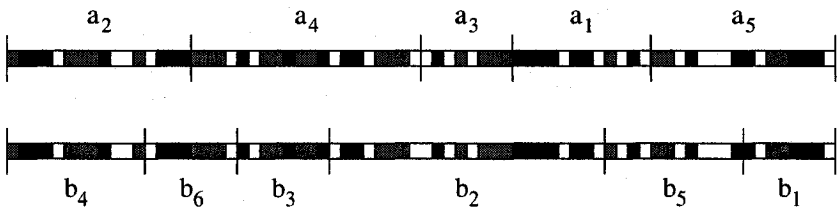


Figure 1.2: The fragments created by enzymes A and B as they appear in the original molecule.

and each fragment b_i, $1 \leq i \leq n$, corresponds to a unique vertex, and there is an edge between each pair of vertices whose corresponding fragments overlap. Figure 1.3 shows the graph corresponding to the mapping information given in Figure 1.2. They derive several

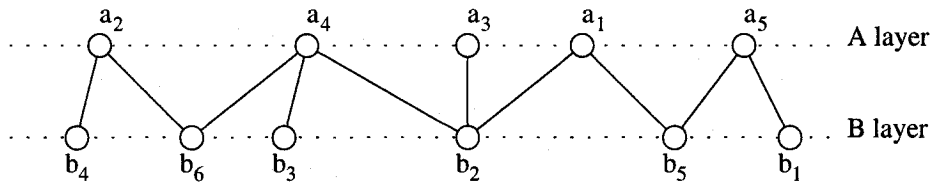


Figure 1.3: The graph corresponding to the fragment overlap information given in Figure 1.2.

properties of such graphs. For example, they show that there is at most one sequence of edges that one can follow when traversing the graph from one vertex to another. In graph-theoretic terms, this implies that the graph is *acyclic*. They also show that each vertex is connected by edges to at most two other vertices that are not *leaves*. A leaf vertex is a vertex that is connected to exactly one other vertex by an edge. Using these properties, they provide an efficient algorithm for obtaining a planar drawing of the graph in which the vertices corresponding to fragments a_i, $1 \leq i \leq m$, lie on one layer, and the vertices corresponding to fragments b_i, $1 \leq i \leq n$, lie on a second layer. They show that the linear order of the vertices on each layer in the drawing corresponds to the order of their corresponding DNA fragments in the original DNA molecule.

It is interesting to note that Waterman and Griggs, without realizing it, actually describe *proper 2-layer planar* graph drawings, drawings that were defined and studied by Harary and Schwenk [45] as far back as 1972. The reason for the term 'proper' is to differentiate these drawings from other layered drawings in which the end-vertices of a single edge may lie on the same layer or lie on non-adjacent layers. We say that these other types of drawings are *unconstrained*. We formally define these and other types of layered drawings in Chapter 2.

Though this DNA mapping application does not illustrate it, other applications areas generalize drawings on two layers to drawings on a larger number of layers. In these contexts, it is usually necessary to determine whether or not a graph has a planar drawing on a given number of layers, and, if it does have such a drawing, to obtain the drawing. Hence, for proper drawings we have the PROPER k-LAYER PLANAR problem: [1]

> **Given:** A graph G and an integer $k \geq 0$.
> **Question:** Is G proper k-layer planar? i.e. does G have a planar drawing in which the vertices lie on k horizontal lines and the edges are drawn as straight-line segments between end-vertices on adjacent layers?

Much has been published about drawing graphs *after* their vertices have already been assigned to layers; examples include the well-known work of Sugiyama, Tagawa and Toda [91] and polynomial-time algorithms of Junger *et al.* [56, 55] and Healy and Kuusik [48]. However, surprisingly little is known about problems like PROPER k-LAYER PLANAR where the vertices have not been pre-assigned to layers. Aside from the $\mathcal{NP}$-completeness result of Heath and Rosenberg [51], [2] Dujmović *et al.* [24] derive the only other results related to this problem. Though they obtain polynomial-time solutions when the number of layers is bounded by a constant, the running times are impractically large for even simple graphs.

More is known when the number of layers is equal to 1, 2 or 3. Of course, for $k = 1$, the PROPER k-LAYER PLANAR problem is trivial and, for $k = 2$, we have already discussed the efficient algorithm of Waterman and Griggs [96]. For $k = 3$, Fößmeier and Kaufmann [40] claim to have a linear-time solution; however, as we will discuss in Chapter 3, their algorithm contains fatal flaws. Cornelsen, Schank and Wagner [17] have published the only result for unconstrained planar drawings on two and three layers. They derive linear-time algorithms for determining whether or not a given graph has unconstrained 2 and 3-layer planar drawings.

In the drawings that we have considered so far, edges are drawn as straight-line segments between their end-vertices. Most applications favour such a restriction because it is easy to see which pairs of vertices are connected by edges. It is natural, however, to consider drawings that relax this restriction, allowing edges to be drawn with a single bend, in other

[1] Similar problem definitions exist for other types of planar layered drawings; however, we omit them here in order to simplify the discussion.

[2] Heath and Rosenberg [51] prove that the $\mathcal{NP}$-completeness of determining whether or not a given graph has a planar layered drawing in which the end-vertices of each edge lie on adjacent layers. They call such drawings *leveled-planar* whereas we call them *proper, planar layered* drawings. This complexity result, however, does not appear to immediately extend to minimizing layers in planar layered drawings that are not necessarily proper. Consequently, the complexity of the problem for these drawings remains open.

words, as polylines consisting of at most two line-segments. Thus, we have the k-LAYER, 1-BEND PLANAR problem:

> **Given:** A graph G and an integer $k \geq 0$.
> **Question:** Is there a k-layer, 1-bend planar drawing of G? i.e. does G have a planar drawing on k layers in which each edge is drawn as a polyline consisting of at most two line-segments?

Even for $k = 1$, this problem is $\mathcal{NP}$-complete [61]. In Chapter 5, we extend this hardness result by showing that the problem is $\mathcal{NP}$-hard for each fixed $k \geq 2$. In other words, it is extremely unlikely that we will ever obtain efficient solutions to this problem.

1.2 Non-Planar Drawings

Sometimes, it is impossible to draw a graph without having a few crossing edges. The application of DNA mapping mentioned above is no exception because of imperfections in the experimental methods for determining whether or not two DNA molecule fragments overlap. Consequently, the resulting graph may be missing a few edges or incorrectly contain a few additional edges. Though missing edges do not prevent us from obtaining 2-layer planar drawings, additional edges may impose crossings. To obtain a reasonable ordering of the DNA fragments under these conditions, we obtain a 2-layer drawing of the graph as before; however, in this case, we either obtain a drawing that minimizes the number of edge crossings, or else we attempt to remove a minimum number of edges such that the resulting graph has a planar drawing. The decision version of the first problem is called 2-SIDED CROSSING MINIMIZATION:

> **Given:** A bipartite graph $G = (V_0, V_1; E)$ and an integer $k \geq 0$.
> **Question:** Is there a 2-layer drawing of G with at most k edge crossings in which the vertices of V_0 lie on one layer and the vertices of V_1 lie on the other layer?

The decision version of the second problem is called 2-LAYER PLANARIZATION:

> **Given:** A graph G and an integer $k \geq 0$.
> **Question:** Is there a set of edges $S \subseteq E$ of size at most k such that $G - S$ has a proper 2-layer planar drawing?

Solving these problems, of course, does not guarantee a correct ordering for the original DNA mapping problem, but, when the experimental errors are small (as is usually the case), the obtained orderings closely resemble the correct orderings.

Both 2-SIDED CROSSING MINIMIZATION and 2-LAYER PLANARIZATION are $\mathcal{NP}$-complete [42, 32, 93]. In fact, Eades and Wormald [33] show that crossing minimization remains $\mathcal{NP}$-complete even when we fix the order of the vertices on one of the layers. This restricted version of the problem is called 1-SIDED CROSSING MINIMIZATION. Muñoz *et al.* [73] shows further that this restricted problem remains $\mathcal{NP}$-complete even when the vertices on the ordered layer have degree equal to one, and the vertices on the other layer have degree at most four.

The 2-LAYER PLANARIZATION problem similarly remains $\mathcal{NP}$-complete even when the order of the vertices on one of the layers is fixed [32, 93]. This restricted version of the problem is called 1-LAYER PLANARIZATION. Eades and Whitesides [32] show further that 2-LAYER PLANARIZATION remains $\mathcal{NP}$-complete even when the vertices on one layer have degree two and the vertices on the other layer have degree at most three. They also show that 1-LAYER PLANARIZATION remains $\mathcal{NP}$-complete when each vertex on the fixed-order layer has degree one, and each vertex on the other layer has degree at most two; that is, the graph is composed of component paths of length at most two. These problems are, however, solvable in polynomial-time when the order of the vertices on *both* layers is fixed [32, 77, 95, 5].

In experiments with branch-and-cut algorithms for crossing minimization, Jünger and Mutzel [58] show that the 2-SIDED CROSSING MINIMIZATION problem may be more difficult to solve in practice than the 1-SIDED CROSSING MINIMIZATION problem. More specifically, their results show that their algorithms can efficiently and optimally solve instances of 1-SIDED CROSSING MINIMIZATION with up to about 60 vertices on the free layer; on the other hand, their algorithm for 2-SIDED CROSSING MINIMIZATION can efficiently handle at most 15 vertices per layer. Further investigations are necessary, then, in order to find more practical solutions to 2-SIDED CROSSING MINIMIZATION, if such solutions exist.

In similar experiments using branch-and-cut algorithms for planarization, Mutzel [74, 75] describes an algorithm for the 2-LAYER PLANARIZATION problem that handles graphs with up to about 100 vertices per layer and 200 total edges. Though these results are quite strong, the algorithm does demonstrate weakness when applied to graphs with edge-to-vertex ratio around 1.25. Mutzel's 1-LAYER PLANARIZATION algorithm is even more efficient, efficient enough to be used in most practical situations to obtain exact solutions [77].

As with planar drawings on layers, we may also consider non-planar drawings on more than two layers. These drawings have applications in visualization [6, 60], and row-based VLSI layout [86]. Of course, these problems are as difficult as their 2-layer versions so, in

practice, we do not normally expect to obtain anything more than approximate solutions.

The most famous approximation approach is due to Sugiyama, Tagawa and Toda [91] which reduces the problem to a series of 1-SIDED CROSSING MINIMIZATION or 1-LAYER PLANARIZATION problems. The vertices are first assigned to layers and then an algorithmic solution to 1-SIDED CROSSING MINIMIZATION or 1-LAYER PLANARIZATION is applied to adjacent pairs of layers, permuting the vertices on one layer while holding the other layer fixed. Usually, this algorithm is applied in a sweeping fashion, beginning with the top two layers and then, moving down layer-by-layer to the bottom two layers, and then in a similar manner back up to the top. These sweeps are repeated until some stopping criteria is satisfied, at which point, a drawing is obtained.

Jünger *et al.* [54] do describe branch-and-bound and branch-and-cut algorithms that minimize crossings globally once vertices have been assigned to layers. Though potentially producing drawings with fewer crossings than algorithms based on Sugiyama's approach, the algorithms are not yet efficient enough to be used in practice. They include experiments only for two and three layer drawings. Healy and Kuusik [47, 46] obtain slight improvements by adding additional constraints based on cycles. In preliminary experiments[3], their algorithms have slightly slower running times but use about half as many branch-and-bound nodes.

1.3 Practical Approaches to $\mathcal{NP}$-hard Problems

Most of the problems that we have just surveyed are $\mathcal{NP}$-hard; consequently, it is very unlikely that any of them can be solved efficiently in general. Indeed, in our survey so far, we have described experiments with state-of-the-art implementations that are often not efficient enough to be used in practice. Since practical solutions are needed, we now survey several approaches to obtaining such solutions to these difficult problems. In particular, we survey algorithms that handle restricted versions of problems, heuristic algorithms, approximation algorithms, branch-and-cut algorithms, and fixed-parameter tractable algorithms.

There are other approaches to solving $\mathcal{NP}$-hard problems such as randomized algorithms, algorithms with running times of the form $\mathcal{O}(\alpha^n)$ where n is the size of the problem instance and α is just slightly larger than 1, and genetic algorithms. We omit these because we are not aware that these approaches have been successfully applied to the problems we consider. Investigating the usefulness of these approaches is a possible area for further research.

[3]The experiments are reported in [47]. They consider 10 different graphs, each on 8 layers with 12 vertices per layer and 110 edges total. One of the graphs requires a minimum of 31 edge crossings.

1.3.1 Problem Restriction

In many applications, it is not necessary to solve a graph problem in full generality. As an example, consider the *software include graph* used by software engineers to model program file dependencies (see Figure 1.4). In these models, each vertex corresponds to a unique

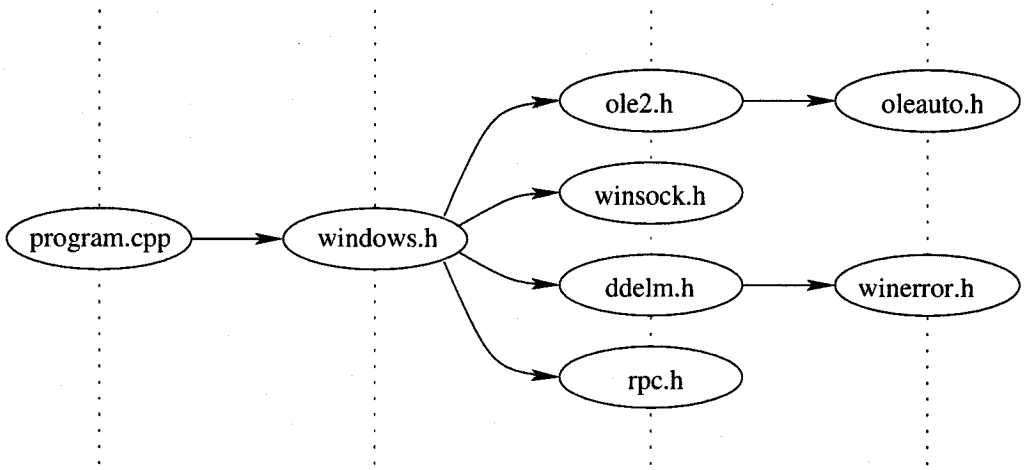


Figure 1.4: A layered drawing of an include graph corresponding to a very simple Windows program.

file and each edge to a case where one file "includes" information from another file. The resulting graphs are called *directed graphs* because each edge has an oriented direction to indicate which of the two files "includes" the information and which file provides the information. According to Lakos [65], these graphs should never contain a directed cycle, for otherwise, this would imply a cyclic dependency in the program files. Furthermore, the most natural way for software engineers to visualize these graphs is as layered drawings in which each edge is oriented from a vertex on a lower layer to a vertex on a higher layer. We note that this is a restriction of the layered drawings mentioned earlier, and, unlike the many of them, this restricted problem has a well-known linear-time algorithm for minimizing the number of required layers in the drawing [6].

1.3.2 Heuristics

A very common approach to handling $\mathcal{NP}$-hard problems is to exchange optimal or exact solutions in favor of 'reasonably good' solutions, usually by applying on one or more heuristics to the problem. A *heuristic* is simply a rule of thumb, often a greedy choice, that can be made quickly.

For example, the Sugiyama framework for layered drawings that we mentioned earlier uses two main heuristics. First of all, to truly minimize edge crossings in a layered drawing of a graph, assigning a layer to each vertex and ordering of the vertices on each layer must be performed in a globally optimal way, and therefore simultaneously. However, to simplify things, the Sugiyama framework separates these two problems into different steps. Secondly, once vertices are assigned to layers, the resulting edge crossings can be minimized only by choosing an ordering of the vertices on each layer, once again, in a globally optimal way, and therefore for all layers simultaneously. However, as described earlier, the Sugiyama framework chooses layer orderings by iteratively considering two layers at a time.

In spite of these simplifications, the crossing minimization problem, called 1-SIDED CROSSING MINIMIZATION, that forms the basis of the Sugiyama framework is also $\mathcal{NP}$-hard. Consequently, practical algorithms within the Sugiyama framework often use heuristics to solve even these problems, including the *barycenter heuristic* of Sugiyama, Tagawa and Toda [91], and the *median heuristic* of Eades and Wormald [33]. In both of these heuristics, the order of the vertices on the free layer, the layer whose order is not fixed, is obtained according to some "averaging" criteria. More specifically, let $w_1, w_2, \ldots, w_p$ be the vertices on the fixed layer, in the order that they appear on that layer, that are each connected to the same vertex v by an edge. In the barycenter heuristic, the x-coordinate $\text{pos}(v)$ of v is set equal to the average x-coordinate of its neighbors $w_1, w_2, \ldots, w_p$:

$$\text{pos}(v) = \frac{1}{p} \sum_{i=1}^{p} \text{pos}(w_i).$$

In the median heuristic, the x-coordinate of v is set equal to the median x-coordinate of its neighbors:

$$\text{pos}(v) = \frac{1}{2}(\text{pos}(w_{\lfloor \frac{p}{2} \rfloor}) + \text{pos}(w_{\lceil \frac{p}{2} \rceil})).$$

Surprisingly, these two simple heuristics consistently outperform other more complex heuristics in experiments [67,58], including the stochastic [22], greedy-insert [30], greedy-switch [30], split [30], and assign [12] heuristics.

Because the barycenter and median heuristics have proved to be so successful, new approaches based on these two heuristics have been proposed. Martí and Laguna [68] describe experiments with a new heuristic called GRASP, which is based on the barycenter heuristic, and show that it outperforms all previous heuristics on sparse graphs. They also include a Tabu Search algorithm in their experiments and show that, in exchange for a longer running time than most other heuristics, the algorithm often finds optimal solutions

when the input graphs have high edge-density.

Stallman *et al.* [88] also report on experiments with several variations of the barycenter and median heuristics. These variations consist of a preprocessing step that obtains an initial ordering of the vertices before applying the barycenter and median heuristics. Their experiments show that these variations are very effective when applied to highly structured graphs such as trees.

1.3.3 Approximation Algorithms

Approximation algorithms are closely related to heuristic algorithms in that they do not guarantee an optimal solution; however, approximation algorithms do come with bounds on their worst-case behavior. It turns out that both the barycenter and median heuristics are approximation algorithms for 1-SIDED CROSSING MINIMIZATION, with barycenter solutions at most $\Theta(\sqrt{n})$ times the minimum [66] and median solutions incredibly at most 3 times the minimum [33]. In other words, the barycenter heuristic is a $\Theta(\sqrt{n})$-approximation and the median heuristic is a 3-approximation algorithm. Yamaguchi and Sugimoto [98] obtain a slightly better approximation with their greedy heuristic. When the maximum degree of the graph is at most four, their algorithm is a 2-approximation, and, as the maximum degree increases, the approximation ratio monotonically increases to a maximum of 3. Recently, Nagamochi [78] has obtained a 1.47-approximation.

For 2-SIDED CROSSING MINIMIZATION, we do not know of any approximation algorithms; this is further evidence that this problem may be significantly more difficult to solve than 1-SIDED CROSSING MINIMIZATION.

1.3.4 Branch-and-Cut

Many problems can be expressed as linear programming problems similar to the following:

maximize $200x_1 + 500x_2$
subject to the following constraints:

$$\begin{aligned} 60x_1 + 40x_2 &\leq 2400 \\ 50x_1 + 50x_2 &\leq 2500 \\ x_1, x_2 &\geq 0 \end{aligned}$$

More generally, a *linear programming problem* has as input a linear function to minimize or maximize subject to a set of linear constraints. A *linear function* f has the form $f(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \ldots + a_nx_n$ where each a_i is a constant rational number and each x_i is a variable. A *linear constraint* has the form $f(x_1, x_2, \ldots, x_n) \leq b$ where b is

a constant rational number and f is a linear function. It is well-known that such problems can be solved in polynomial time and efficiently in practice as well.

However, the problems that we consider cannot be expressed as linear programming problems unless we are able to constrain some of the variables to take only integer values. Programs with these types of constraints are called *(mixed) integer linear programs*. For example, Jünger *et al.* [54] express 2-SIDED CROSSING MINIMIZATION as the following integer linear program:

minimize $\sum_{(v_i,w_j),(v_k,w_l)\in E} c_{ijkl}$
subject to the following constraints:

$$\begin{aligned}
x_{ik} + y_{lj} - c_{ijkl} &\leq 1 \\
x_{ki} + y_{jl} - c_{ijkl} &\leq 1 \quad (v_i,w_j),(v_k,w_l) \in E \text{ and } i<k, j\neq l \\
x_{ij} + x_{jk} + x_{ki} &\leq 2 \quad v_i,v_j,v_k \in V (i<j<k) \\
y_{ij} + y_{jk} + y_{ki} &\leq 2 \quad w_i,w_j,w_k \in V (i<j<k) \\
x_{ij} + x_{ji} &= 1 \quad v_i,v_j \in V (i<j) \\
y_{ij} + y_{ji} &= 1 \quad w_i,w_j \in V (i<j) \\
x_{ij}, y_{ij}, c_{ijkl} &\in \{0,1\}
\end{aligned}$$

In this program, there is a variable x_{ij} for each pair of vertices v_i and v_j on the first layer, such that $x_{ij} = 1$ if v_i is before v_j on the layer, and $x_{ij} = 0$ otherwise. There is similarly a variable y_{ij} for each pair of vertices w_i and w_j on the second layer. Then, for each pair of edges (v_i, w_j) and (v_k, w_l) such that $i < k$ and $j \neq l$, there is a variable c_{ijkl} that is equal to 1 if the edges cross and equal to 0 otherwise. In the program, we wish to minimize the number of crossings $\sum_{(v_i,w_j),(v_k,w_l)\in E} c_{ijkl}$. The first two constraints ensure that $c_{ijkl} = 1$ whenever edges (v_i, w_j) and (v_k, w_l) cross, that is, whenever $x_{ik} = y_{lj} = 1$ (see Figure 1.5(a)) or $x_{ki} = y_{jl} = 1$ (see Figure 1.5(b)). The next two constraints ensure that the vertex

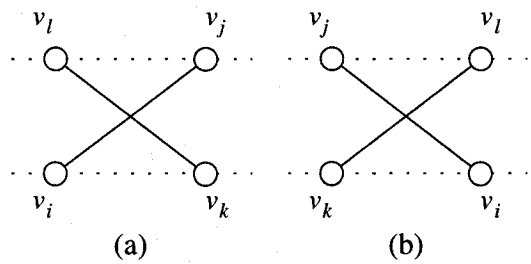


Figure 1.5: Edge crossings corresponding to variable values in an integer linear program: (a) $x_{ik} = y_{lj} = 1$ and (b) $x_{ki} = y_{jl} = 1$.

orderings implied by the variables are transitive. The final two constraints ensure that each pair of vertices is ordered.

Integer linear programs can be solved using *branch-and-bound* algorithms. As the name suggests, branch-and-bound algorithms consist of "branching" and "bounding". The "branching" part involves selecting a variable with integer constraints and using it to split the problem into two subproblems. More specifically, if the variable must take a integer value between lower bound L and upper bound U, then we might generate a subproblem in which the variable must take an integer between L and $\lfloor (U + L)/2 \rfloor$ and another subproblem in which the variable must take an integer between $\lfloor (U + L)/2 \rfloor + 1$ and U. The algorithm then "branches", solving each subproblem separately.

The "bounding" part usually involves solving the *relaxation* of the integer linear programming problem; in other words, the problem is solved while allowing all variables to take non-integer values. As mentioned earlier, the relaxation can be solved efficiently. We observe that such a solution is a lower bound on the solution for the original problem if it is a minimization problem, and an upper bound on the solution for the original problem if it is a maximization problem. In some cases, the relaxation solution satisfies the integer constraints of the original problem. In that case, the solution is an optimal solution for the subproblem corresponding to the current branching and a feasible solution (i.e. satisfies all constraints) for the original problem. It is not necessarily an optimal solution to the original problem because, as mentioned above, each branching creates subproblems by adding new constraints.

As a branch-and-bound algorithm runs, the feasible solutions that it discovers incrementally improve until an optimal solution is discovered. As a result, if we can find an initial feasible solution to the problem before beginning the branch-and-bound algorithm using some heuristic, then the branch-and-bound algorithm acts as an *anytime algorithm*; in other words, at any point in the execution of the algorithm, we can stop it and ask for the best feasible solution that it has computed. On the other hand, if we allow the algorithm to finish, then the best feasible solution is also optimal.

One way to increase the efficiency of branch-and-bound algorithms is to add additional constraints to the integer linear program as the branch-and-bound algorithm runs. Though these added constraints do not affect the optimal solution of the program, they may help the algorithm converge to an optimal solution more efficiently. For example, in any proper 2-layer drawing of a simple cycle, there are at least $n/2 - 1$ edges crossings, where n is the number of edges in the cycle. Therefore, for each simple cycle C in the graph, we could add the following constraint to the program:

$$\sum_{(v_i,w_j),(v_k,w_l) \in E(C)} c_{ijkl} \geq n/2 - 1,$$

where $E(C)$ is the set of edges in C and n is the size of $E(C)$. Unfortunately, generating such a constraint for each cycle in a typical graph would result in far too many constraints for any linear programming system to handle. Therefore, we add these types of constraints dynamically whenever they are violated by some intermediate relaxation solution. This variation of branch-and-bound is called *branch-and-cut*.

We recall that branch-and-cut algorithms are used in the experiments of Jünger and Mutzel [58], Jünger *et al.* [54], and Healy and Kuusik [47,46], as well as in the planarization experiments of Mutzel [74,75] mentioned earlier. There we noted that the branch-and-cut algorithms were efficient enough to solve 1-LAYER PLANARIZATION in most practical applications, efficient enough to solve 1-SIDED CROSSING MINIMIZATION and 2-LAYER PLANARIZATION in many practical applications, but too inefficient to solve anything but very small instances of 2-SIDED CROSSING MINIMIZATION. Branch-and-cut algorithms depend on many factors, including the types of cuts used and how they are applied, so further investigations are necessary to determine whether or not all of these problems can be solved efficiently using branch-and-cut algorithms.

1.3.5 Parameterized Complexity

The motivation behind the theory of parameterized complexity is that most applications do not require efficient solutions for all instances of difficult problems. This theory formalizes this idea by studying parameterized decision problems. A *parameterized decision problem* consists of a classical decision problem, usually $\mathcal{NP}$-hard, together with a parameter, often an integer, that relates to the problem input in some way. The hope is that, for some bounded range of parameter values, the parameterized decision problem has efficient solutions. For instance, let us consider a parameterized version of 2-SIDED CROSSING MINIMIZATION where the parameter is the number of edge crossings k that are permitted. Notice that, if this parameterized problem can be solved efficiently for small values of k, then it is likely to be an efficient solution to the DNA mapping problem (described on page 2) because the number of experimental errors, and therefore the number of imposed edge crossings, is normally very small. Similarly, a parameterized version of 2-LAYER PLANARIZATION could have as its parameter the maximum number of edges k that may be removed to planarize the graph. In this case as well, a solution to the problem that is efficient for small values of k is likely to be an efficient solution to the DNA mapping problem as well.

In parameterized complexity, as in classical complexity, a problem is *tractable* if it can be solved in polynomial time. In the language of parameterized complexity, such a problem is said to be *fixed-parameter tractable*, or in the class $\mathcal{FPT}$. More formally, a

parameterized problem with parameter k and input size n belongs to $\mathcal{FPT}$ if it can be solved in $\mathcal{O}(f(k) \cdot n^{\alpha})$ time, for some function f that is independent of n, and a constant $\alpha > 0$. An algorithm whose running time proves that a problem belongs to $\mathcal{FPT}$ is said to be an $\mathcal{FPT}$ algorithm.

Although nearly half of the parameterized versions of the $\mathcal{NP}$-complete problems in [41] belong to $\mathcal{FPT}$ [82], many appear to be outside this class. In classical complexity theory, $\mathcal{NP}$-hard problems are those that are not likely to have polynomial-time solutions. Parameterized complexity has an analogous class of problems, $\mathcal{W}[1]$-*hard* problems, that are not likely to belong to $\mathcal{FPT}$. In fact, the theory organizes problems into a whole hierarchy of hardness classes $\mathcal{FPT} \subseteq \mathcal{W}[1] \subseteq \mathcal{W}[2] \subseteq \ldots$ and supplies appropriate reducibility and completeness notations for each.

The value of any theory is determined by how well it models and explains current observations and how well it predicts future observations. Classical complexity theory has proved to be a valuable theory. For example, it has so far accurately predicted that $\mathcal{NP}$-hard problems cannot be be efficiently solved in general. Unfortunately, these predictions have not always proved to be useful in practice. For example, consider the well-known problems VERTEX COVER and INDEPENDENT SET. These two problems are very similar; in fact, they are duals of one other: if G is a graph with vertex set V, then S is a vertex cover for G if and only if $V \setminus S$ is an independent set for G. Not surprisingly, in classical complexity, both of these problems belong to the same complexity class, the class of $\mathcal{NP}$-complete problems. In addition to this, neither problem admits a polynomial-time approximation scheme [52, 53]. Both theoretical and experimental evidence suggests that VERTEX COVER is the easier problem to solve. For example, the following is a simple 2-approximation algorithm for solving VERTEX COVER: select an edge in the graph, add the end-vertices of the edge to the cover, remove the vertices from the graph, and then repeat the previous three steps until the graph contains no vertices. More sophisticated techniques have been used to obtain an approximation within a factor of $2 - \frac{\log\log|V|}{2\log|V|}$ times the optimal [4, 72]. On the other hand, INDEPENDENT SET is not likely to be approximated within $|V|^{1/2-\epsilon}$ times the optimal for any $\epsilon > 0$ [53]. On the experimental side, Cheetham *et al.* [14] report that their parallel system consistently solves the VERTEX COVER problem for graphs with covers of size up to 400 in at most 1.5 hours. Experiments with INDEPENDENT SET have not been nearly as successful. One of the fastest algorithms is based on branch-and-bound [94].[4] This algorithm regularly finds independent sets of size up to

[4]The actual results of Tomita and Seki [94] are for finding the maximum clique. It is well-known that the size of the maximum clique in a graph is equal to the maximum independent set in the dual of the graph, the graph obtained by connecting vertex pairs with edges if and only if they are not connected by edges in the original graph. Thus, an algorithm that finds maximum cliques can be easily adapted to find maximum

30-40 vertices, though, in some cases, it takes 4-5 hours to find an independent set of size as small as 10. [5]

Thus, as this evidence appears to suggest, these two problems should fall into a different complexity classes in a finer grained complexity theory. Indeed, this is the case for parameterized complexity because VERTEX COVER is in $\mathcal{FPT}$, and INDEPENDENT SET is in the harder $\mathcal{W}[1]$ class [21].

1.3.5.1 Toolkit

An important benefit of parameterized complexity to practice is that it provides a growing, systematic toolkit of techniques for obtaining $\mathcal{FPT}$ algorithms. The benefit of this toolkit is not that its techniques are entirely novel because, in fact, many of the basic techniques are used in other approaches to obtain efficient algorithms. Instead, the benefit is that in parameterized complexity, its toolkit of techniques is being used systematically to obtain $\mathcal{FPT}$ algorithms.

Not all of the techniques in the toolkit are intended to obtain $\mathcal{FPT}$ algorithms that are efficient in practice. Some common examples include color-coding (hashing) [21], well-quasi-ordering [21], and monadic second-order logic with bounded treewidth [20]. Though definitely not efficient in practice, finding $\mathcal{FPT}$ algorithms using these techniques is not completely meaningless from a practical point of view. For example, the first $\mathcal{FPT}$ algorithms for VERTEX COVER and GRAPH GENUS were based on these techniques but increasingly more efficient algorithms have since been obtained [71, 21]. Our concern in this thesis is with efficient algorithms so this is all we will say about these techniques.

Other techniques that may lead to efficient algorithms include kernelization [21], bounded search trees [21], bounded treewidth [21], crown decompositions [1], extremal structure such coordinatized kernels [36], and catalyzation [36]. In the remainder of this section, we describe kernelization, bounded search tree and bounded treewidth in more detail and survey their uses in layered graph drawing. We omit further discussion of the other techniques because they have not yet been successfully applied to layered graph drawing.

Kernelization. The most commonly used technique is called *kernelization*, the process of transforming problem instances into equivalent simpler instances. Of course, kernelization

independent sets by simply preprocessing the input graph to obtain its dual.

[5] In some rare cases, the algorithm Tomita and Seki [94] finds independent sets of size up to 150; however, these graphs have extremely low edge-density, smaller than 0.1. Graphs with such small edge-density tend to have very large independent sets in comparison to the number of vertices in the graph; therefore, they tend to have very small minimum vertex covers. In other words, we would expect to be able to use the algorithm of Cheetham *et al.* [14] to quickly find a minimum vertex cover and from that obtain a maximum independent set. Thus, even these rare cases do not support the hypothesis that VERTEX COVER and INDEPENDENT SET are equally difficult.

is not an invention of parameterized complexity. Years before anyone had dreamt of the name "parameterized complexity", Nemhauser and Trotter [79] discovered a clever way to use linear programming to greatly reduce the number of variables in an integer linear program for solving the VERTEX COVER problem.

However, in parameterized complexity, kernelization has become a standard technique that is used in conjunction with one or more parameters. We illustrate using the algorithm of Dujmović and Whitesides [28] for the parameterized version of 1-SIDED CROSSING MINIMIZATION whose parameter k is the number of permitted edge crossings. In their algorithm, they kernelize by assigning a so-called "natural ordering" to certain pairs of vertices on the unordered layer. A pair of vertices u and v has a *natural ordering* if the neighbors of u on the ordered layer all appear before or all appear after the neighbors of v on the ordered layer. In the first case, the natural ordering is u before v, and, in the second case, the natural ordering is v before u. They show that every optimal solution to the problem satisfies the natural orderings. This kernelizing step is easily completed in $\mathcal{O}(|G|^3)$ time, after which the algorithm determines whether or not the remaining pairs of vertices, those without natural orderings, can be assigned an ordering while creating at most k edge-crossings. Though this step requires exponential time, it is a function of k rather than of the size of the graph. This is because each pair of vertices without a natural ordering is responsible for at least one edge-crossing in any solution. Thus, if there are more than k pairs of vertices without natural orderings, then every ordering of the vertices creates more than k edge-crossings so the algorithm can simply return false; otherwise, the algorithm enumerates all possible orderings, at most 2^k, of these pairs to determine if one of them induces at most k edge-crossings. Thus, we have obtained an algorithm that runs in $\mathcal{O}(k^2 \cdot 2^k + |G|^3)$ time, proving that the problem belongs to $\mathcal{FPT}$. We will show below how to improve this running time using a heuristic and bounded search trees.

For another graph drawing problem, upward planarity testing, Chan [13] also uses kernelization to obtain a solution that runs in $\mathcal{O}(k! \cdot 8^k \cdot n^3 + 2^{3 \cdot 2^l} \cdot k^{3 \cdot 2^l} \cdot k! \cdot 8^k \cdot n)$ time, where k is a problem parameter bounding the number of triconnected components in the input graph, l is a second problem parameter bounding the number of cut-vertices in the graph, and n is the number of vertices in the graph. The algorithm is based on a result showing that the number of planar embeddings of a graph is bounded by a function of the number of triconnected components in the graph. Thus, the algorithm works by enumerating all planar embeddings and determining whether or not one of embeddings is upward planar.

Healy and Lynch [50] improve on this result, also using kernelization, to obtain a solution that runs in $O(2^k \cdot k! \cdot n^2)$ time. They also describe another algorithm with a running time of $O(n^2 + \Delta^4 \cdot (2\Delta + 1)!)$ where $\Delta = |E| - |V|$.

These examples show positive results where a kernelizing procedure leads directly to an $\mathcal{FPT}$ algorithm. As we mentioned earlier, however, not all parameterized problems belong to $\mathcal{FPT}$. Even for these problem kernelization can be used to reduce problem instances to obtain something manageable in certain applications, see e.g. [19].

Bounded Search Trees. The second technique uses *bounded search trees*. Similar to branch-and-bound algorithms, bounded search tree algorithms organize the search space of the problem instance as a rooted tree, but, unlike branch-and-bound, uses the problem parameter to bound the size of the tree. For example, the exhaustive enumeration step in the algorithm of Dujmović and Whitesides [28] described in the previous section could be organized as a bounded search tree. At each node in the tree, we associate a subproblem of the original problem instance. To the root node, we associate the original problem instance. We create two children for a node by selecting an unordered pair of vertices and fixing their order in two possible ways, one for each child. The children inherit not only these fixed orderings but also all orderings fixed by their ancestors. Thus, a search tree node has no children if all pairs are ordered by its ancestors in the search tree. Since there are at most k unordered pairs, the height of the tree is at most k so that size of the tree is at most $2^k + 1$.

Organizing the search space in this way, it is possible to further reduce the size of the tree by observing that we are only interested in vertex orderings that create at most k edge crossings. Thus, if fixing the order of a pair of vertices u and v results in only 1 additional edge crossing regardless of the ordering that we choose, then we can actually ignore this pair in the search tree. In other words, a node has no children when the only pairs of vertices unordered by the node's ancestors are like u and v. No further branching is needed at this node because every ordering of u and v creates exactly one additional edge crossing. Using this observation, Dujmović and Whitesides [28] reduce the size of the search tree from $\mathcal{O}(2^k)$ to $\mathcal{O}(a^k)$, where $a = \frac{1+\sqrt{5}}{2} \approx 1.618$. Using more complicated heuristics, Dujmović, Fernau and Kaufmann [26] further reduce the size to $\mathcal{O}(a^k)$ where $a \approx 1.4656$. We describe this algorithm in Chapter 7.

Bounded search tree algorithms have also been used to solve planarization problems. The first $\mathcal{FPT}$ algorithms for 1-LAYER PLANARIZATION and 2-LAYER PLANARIZATION had running times of $\mathcal{O}(3^k \cdot |G|)$ and $\big(6^k \cdot |G|)$, respectively [23]. The result for 2-LAYER PLANARIZATION was then slightly improved in the journal version of that paper [25] to $\mathcal{O}(k \cdot 6^k + |G|)$. Very recently, Fernau [39] has combined heuristics from solutions to the 3-HITTING SET and 6-HITTING SET problems to reduce the running times for 1-LAYER PLANARIZATION and 2-LAYER PLANARIZATION down to $\mathcal{O}(k^3 \cdot 2.5616^k + |G|^2)$ and $\mathcal{O}(k^2 \cdot 5.1926^k + |G|)$, respectively.

Bounded Treewidth. A third technique uses *bounded treewidth*. Very roughly, *treewidth* is a measure of how closely a graph resembles a tree. It is well-known that many intractable problems become tractable when restricted to graphs with bounded treewidth [83]. In the language of parameterized complexity, we say that the problems, when parameterized by taking the treewidth of the input graph as a parameter, belong to $\mathcal{FPT}$.

One approach for obtaining $\mathcal{FPT}$ algorithms for problems parameterized by treewidth uses dynamic programming on tree decompositions. If a graph G has treewidth k, then there exists a tree decomposition of G of width k. Very roughly, this tree decomposition is a tree that resembles G, where each tree vertex corresponds to a set of at most $k+1$ vertices in G. The problem is then solved by first obtaining a tree decomposition of width k and then applying a dynamic programming solution to the problem on the tree decomposition. There is an $\mathcal{FPT}$ algorithm that determines whether a graph has treewidth k, and, if it does, the algorithm obtains a tree decomposition of width k [84]. Thus, to prove that the problem is in $\mathcal{FPT}$, we need only find a $\mathcal{FPT}$ dynamic programming solution to the problem. Bodlaender [9] gives a more detailed overview of the different techniques one can use to obtain $\mathcal{FPT}$ algorithms when treewidth is bounded.

This technique was successfully applied by Dujmović *et al.* [24] to obtain $\mathcal{FPT}$ algorithms for the (h,k)-LAYER CROSSING MINIMIZATION problem:

> **Given:** A graph G and integers $h, k \geq 0$.
> **Question:** Is there an h-layer drawing of G containing at most k edge crossings in which edges are drawn as straight-line segments between their end-vertices?

and for the (h,k)-LAYER PLANARIZATION problem:

> **Given:** A graph G and integers $h, k \geq 0$.
> **Question:** Is there a set of edges S of size at most k in G such that $G - S$ has an h-layer planar drawing?

Dujmović *et al.* [24] also obtain $\mathcal{FPT}$ algorithms for variations of both these problems in which the drawings must be proper, i.e. edges must be drawn as straight-line segments between end-vertices on adjacent layers.

Though the algorithms are $\mathcal{FPT}$, their importance is purely theoretical because their running times are $\mathcal{O}(2^{32(h+2k)^3} \cdot |G|)$. There are two reasons for the poor running time. The first is that the dynamic programming step in their algorithms runs in something of the order $\mathcal{O}((h+k)^{h+k})$, and the second is that the $\mathcal{FPT}$ algorithms for computing minimum tree decompositions have even worse running times. In spite of this, this algorithm may eventually prove to be useful if we can modify it to obtain some kind of approximation

algorithm. Currently, there are efficient approximation algorithms for computing tree decompositions [63], and these have been used, for example, to solve several cases of partial constraint satisfaction problems that had never before been solved [64].

1.4 Contributions and Organization of the Thesis

In the next chapter, **Chapter 2**, we present formal definitions and previous results that are used throughout the paper. The main body of the thesis is divided into two parts.

Part I Planar drawings.

Chapter 3 Proper three-layer drawings.

We correct an oversimplified result in the literature by characterizing proper 3-layer planar drawings and, from the characterization, obtain a corresponding efficient linear-time recognition and drawing algorithm. Based on the complexity of our characterization, however, we conclude that we may need to consider other approaches in order to consider solving similar problems for four layers.

Chapter 4 Tree drawings.

We study the relationship of pathwidth (a specialization of treewidth) to the number of layers required in planar layered drawings of trees under four different drawing models. For each model, we prove optimal upper and lower bounds on the number of layers with respect to pathwidth, and obtain efficient linear-time algorithms for obtaining drawings satisfying these bounds.

Our results correct a previously published result about pathwidth and layered drawings of trees.

Chapter 5 One-bend drawings.

We study layered drawings in which each edge may be drawn with at most one bend. We obtain a simple characterization of graphs that have such drawings on two layers. We also prove that the problem of testing whether or not a graph has such a drawing on any fixed number of layers is $\mathcal{NP}$-hard.

Part II Non-planar drawings.

Chapter 6 Biplanarization algorithms.

We derive new algorithms for the 1-LAYER PLANARIZATION and 2-LAYER PLANARIZATION problems. Our algorithms with running times of $\mathcal{O}(2^k + |G|)$ and $\mathcal{O}(3.562^k + |G|)$, respectively, are currently the fastest algorithms known

for this problem. The previous best running times were by Fernau [39] and had running times of $\mathcal{O}(k^3 \cdot 2.5616^k + |G|^2)$ and $\mathcal{O}(k^2 \cdot 5.1926^k + |G|)$, respectively. We also describe a new divide-and-conquer heuristic for traversing bounded search trees. We show how to incorporate this heuristic into our 2-LAYER PLANARIZATION algorithms without increasing their asymptotic running times.

Chapter 7 One-sided crossing minimization and applications.

We describe the most recent 1-SIDED CROSSING MINIMIZATION algorithm [26] and our implementation of it. We also show how to incorporate the divide-and-conquer heuristic described in Chapter 6 into the algorithm of [26] while only slightly increasing the asymptotic running time. We end the chapter by showing how to modify the algorithm of [26] to obtain $\mathcal{FPT}$ algorithms for two problems from bioinformatics.

Chapter 8 Experiments with $\mathcal{FPT}$ algorithms.

We describe our experiments with $\mathcal{FPT}$ algorithms for 2-LAYER PLANARIZATION and 1-SIDED CROSSING MINIMIZATION as described in Chapters 6 and 7. Our experiments show that, with respect to the 2-LAYER PLANARIZATION problem, our implementations of $\mathcal{FPT}$ algorithms are competitive with and in many cases better than current branch-and-cut algorithms. On the other hand, with respect to the 1-SIDED CROSSING MINIMIZATION problem, current branch-and-cut algorithms remain the clear winners.

We conclude the thesis in **Chapter 9**, by presenting a short review of the thesis highlights and describing topics for future research.

Chapter 2

Preliminaries

In this chapter we give notation, definitions, and fundamental results that are used throughout the thesis.

2.1 Set Notation

We assume that the reader is familiar with standard set-notation, so in this section we only define notation that may not be so standard. If a is an element in S, then we use $S - a$ to denote the subset of S containing all elements of S except element a. Similarly, if S' is a subset of S, then we use $S \setminus S'$ to denote the subset of S containing all elements of S except for those in S'. Conversely, $S + a$ denotes the set containing a and the elements of S, and $S \cup S'$ denotes the set containing the elements of S and S'.

2.2 Graphs

Given a graph $G = (V, E)$, we use $V(G) = V$ to denote the vertices of G and $E(G) = E$ to denote its edges. An edge $e \in E(G)$ corresponds to a pair of vertices $u, v \in V(G)$ and is denoted (u, v) or (v, u). Given $e = (u, v)$, we say that e is *incident* on u and v, that u and v are *adjacent* to one another and to e, that u and v are the *end-vertices* of e, and that u and v are *neighbors*. The graphs in this thesis are *undirected* so the notation (u, v) and (v, u) denote the same edge. The *degree* of a vertex v in G, denoted $\deg_G(v)$ (or just $\deg(v)$ when the context is clear), is equal to the number of edges incident on v in G. For a vertex v, we use $\mathrm{Adj}(v)$ to denote the set of vertices adjacent to v and $\mathrm{Inc}(v)$ to denote the set of edges incident on v.

A *subgraph* H of G is a graph for which $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. The subgraph H of G *induced* by a set of vertices $S \subseteq V(G)$ is such that $V(H) = S$ and $E(H) = \{(u, v) \mid u, v \in S, (u, v) \in E(G)\}$. A graph G is *bipartite* if there exist subsets $A, B \subseteq V(G)$ such that $V(G) = A \cup B$, $A \cap B = \emptyset$, and each edge in G has one end-vertex

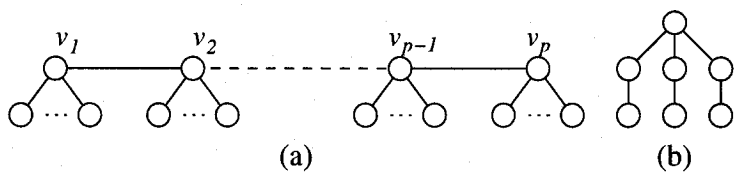


Figure 2.1: (a) Caterpillar, (b) 2-Claw

in A and the other in B. Vertex sets A and B are called *bipartition classes* of G, and we write $G = (A, B; E)$ where $E = E(G)$.

Following our set-notation above, we use $G - v$ to denote, for some vertex $v \in V(G)$, the induced subgraph of G on vertex-set $V(G) - v$, and we use $G - e$ to denote, for some edge $e \in E(G)$, the subgraph H of G with $V(H) = V(G)$ and $E(H) = E(G) - e$. Similarly, given a set of vertices $S \subseteq V(G)$, $G \setminus S$ denotes the induced subgraph of G on vertex set $V(G) \setminus S$. If, on the other hand, S is a set of edges from $E(G)$, then $G \setminus S$ denotes the subgraph H of G with $V(H) = V(G)$ and $E(H) = E(G) \setminus S$.

A vertex is called a *leaf* if it is adjacent to exactly one other vertex in the graph. We use $\deg'_G(v)$ (or $\deg'(v)$ when the context is clear) to denote the number of non-leaf neighbors of a vertex v in a graph G. Any graph that can be transformed into a path by removing all its leaves is a *caterpillar*. This unique path is called the *spine* of the caterpillar. The *2-claw* is the smallest tree that is not a caterpillar. It consists of a vertex called the *root* that has three neighbors, and each neighbor is additionally adjacent to a leaf. See Figure 2.1 for illustrations of a caterpillar and a 2-claw.

A *path* in graph G is a subgraph of G defined by a sequence of vertices in which each edge is incident on a pair of consecutive vertices and each pair of consecutive vertices is adjacent. A *simple path* is a path defined by a sequence of non-repeating vertices. We often denote a path by listing the vertices in their sequential order. A graph G is then said to be *connected* whenever there is a (simple) path in G between each pair of vertices. A *cycle* in a graph G is a subgraph of G defined by a cyclic sequence of the vertices in which each edge is incident on a pair of consecutive vertices and each pair of consecutive vertices is adjacent. A *simple cycle* is then a cycle defined by sequence of non-repeating vertices. A graph is said to be *acyclic* if it contains no simple cycles. A *tree* is an acyclic and connected graph. A *forest* is a graph composed of one or more trees.

A vertex v is a *cut-vertex* in a graph G if every path between two vertices of G other than v contain v. If a graph contains no cut-vertices, then we say that it is *biconnected*. An edge e is a *bridge* in G if every path between two vertices of G contains e.

Let $P = v_1 \dots v_p$ be a simple path of length at least two in a graph G. If $\deg'_G(v_1) \geq 3$,

$\deg'_G(v_p) = 1$, and the remaining vertices v_i have $\deg'_G(v_i) = 2$, then the subgraph induced by the vertices of P and the neighbors of vertices $v_2, \ldots, v_p$ is called a *pendant caterpillar* of G. This pendant caterpillar is said to be *connected* to the graph at v_1, its *connection point*. If, instead, we have $\deg'_G(v_p) \geq 3$, then the subgraph induced by the vertices of P and the neighbors of vertices $v_2, \ldots, v_{p-1}$ is called an *internal caterpillar* of G. This internal caterpillar is said to be *connected* to G at vertices v_1 and v_p, its *connection points*. If an internal caterpillar is a path, then it is also called an *internal path.*

Any graph that can be transformed into a simple cycle C by removing all of its leaves is called a *wreath*. The edges of C are called the *cycle edges* and C is called the *wreath cycle* of the wreath. A wreath subgraph is called a *pendant wreath* if exactly one of the vertices v in the wreath has a neighbor outside of the wreath, and v is on the wreath cycle. The wreath subgraph is said to be *connected* to the rest of the graph at v, its *connection point*.

In this thesis, we define a *drawing* of a graph to be a drawing in the plane in which each vertex is assigned a unique point in the plane and each edge is drawn as a curve connecting its end-vertices. We often use the symbol Γ_G (or just Γ when the context is clear) to refer to a drawing of a graph G. A *planar drawing* of a graph is a drawing in which no two edges cross. Two edges *cross* whenever they intersect at a point other than at a shared end-vertex. Figure 1.1(a) gives an example of a planar drawing and Figure 1.1(b) gives an example of a non-planar drawing. A graph is *planar*, then, if it has a planar drawing. It is well-known that a planar drawing of a graph induces an cyclic ordering on the set of edges incident on each single vertex. The ordering is obtained by sorting the edges according to the angle at which they first leave the vertex in the drawing. Thus, a *planar embedding* of a graph consists of an ordering of incident edges for each vertex as defined by a planar drawing of the graph. In this thesis, we use Φ_G (or just Φ when the context is clear) to denote the planar embedding of a given planar graph G. In a planar graph drawing, the edges cut the plane into regions. Each region is called a *face* and corresponds to a cycle in the graph composed of the vertices and edges that lie on its border. We say that the cycle *bounds* the face and observe that, in the planar embedding corresponding to the drawing, each pair of consecutive edges in the cycle is also consecutive in the cyclic ordering of their shared end-vertex. In the drawing, exactly one face extends to infinity. We call this face the *external* face, and we call the other faces *internal* faces. A planar graph is *maximal planar* if no edge can be added without violating its planarity. We observe that if G is maximal planar, then the set of faces in each planar embedding of G remains the same except for the choice of the external face. Also, each face in each embedding is bounded by a 3-cycle.

A graph is *Hamiltonian* if it has a simple cycle that traverses all its vertices; such a cycle is called a *Hamiltonian cycle*. A planar graph G is *sub-Hamiltonian* if either G is

Hamiltonian or G can be augmented with edges in such a way that it becomes Hamiltonian and remains planar. Observe that a maximal planar graph is sub-Hamiltonian if and only if it is Hamiltonian.

2.3 Layered Drawings

An *k-layer drawing* of a graph is a drawing in which each vertex is placed on one of k parallel lines. In all chapters except Chapter 5, edges are represented in the drawings that we consider by straight-line segments between their end-vertices. In Chapter 5, edges are drawn as polylines consisting of at most two straight-line segments.

We assume, without loss of generality, that the layers are horizontal and parallel to the x-coordinate axis. We say that the *left* direction on a layer is toward toward $x = -\infty$, and the *right* direction on a layer is toward $x = \infty$. When we are not concerned about the exact number of layers, we simply call an k-layer drawing a *layered drawing*.

In a layered drawing, we say that an edge is *proper* if its endpoints lie on adjacent layers, *short* if they lie on the same layer, and *long* otherwise. Thus, a *proper layered drawing* contains only proper edges, a *short layered drawing* contains no long edges, an *upright layered drawing* contains no short edges, and an *(unconstrained) layered drawing* may contain any of these types of edges.

When referring to planar drawings on k layers, we say that the drawings are *proper / short / upright / unconstrained k-layer planar* drawings. Thus, a graph with such a drawing is said to be *proper / short / upright / unconstrained k-layer planar.* In many cases, we will drop the term 'unconstrained'.

The following result is fundamental to nearly all results presented in this thesis. It characterizes the set of graphs that are proper 2-layer planar.

Lemma 2.1 ([31,45,93]) *Let G be a graph. The following are equivalent:*

1. *G is proper 2-layer planar;*
2. *G is a forest of caterpillars;*
3. *G is acyclic and contains no 2-claw; and*
4. *The graph obtained from G by deleting all leaves is a forest and contains no vertex with degree three or greater.*

A graph is *biplanar* if it is a proper 2-layer planar graph. A set of edges whose removal from a graph G makes it biplanar is called a *biplanarizing set* for G. The *biplanarizing number* of G, denoted by $\mathsf{bpn}(G)$, is the size of the minimum biplanarizing set for G.

It is convenient to extend these definitions as they relate to the 1-LAYER PLANARIZATION problem whose input is a bipartite graph $G = (A, B; E)$ and a linear ordering π of A. A bipartite graph $G = (A, B; E)$ and a linear ordering π of A is *biplanar* if G has a proper 2-layer planar drawing in which the vertices of A lie on a single layer in order given by π. Thus, a *biplanarizing set* for G and π is a set of edges whose removal from G makes the resulting graph and π biplanar. Finally, the *biplanarizing number* of G and π, denoted by $\mathsf{bpn}(G, \pi)$, is the size of the minimum biplanarizing set for G and π.

For the 2-SIDED CROSSING MINIMIZATION problem, given a bipartite graph $G = (A, B; E)$, the *bipartite crossing number* of G, denoted $\mathsf{bcr}(G)$, is the minimum number of edge crossings in any proper 2-layer drawing of G. For the 1-SIDED CROSSING MINIMIZATION problem, we also have as input a linear ordering π of the vertices of A. In this case, the *bipartite crossing number* of G and π, denoted $\mathsf{bcr}(G, \pi)$, is the minimum number of edge crossings in any proper 2-layer drawing of G in which the vertices of A lie on a single layer in the order given by π.

In Chapters 6 and 7, we use linear and partial orderings extensively. To avoid unnecessary detail, we give only simple, intuitive definitions here. A *linear ordering* of a set of elements S simply defines a sequential ordering of the elements in S. A *partial ordering* of S defines an ordering of various pairs of elements in S such that there exists a linear ordering of S that defines the same ordering for these pairs. Given a linear or partial ordering π, we use $x <_\pi y$ to say that element x appears before element y in π, and we use $x \leq_\pi y$ to say that element x appears before element y in π or is equal to y. One implication of these definitions is that linear and partial orderings are *transitive*, i.e. if π is a linear or partial ordering such that $a <_\pi b$ and $b <_\pi c$, then we must also have $a <_\pi c$.

2.4 Points in the Plane

A point p in the plane has a position defined by two values, an x-coordinate denoted $\mathrm{X}(p)$ and a y-coordinate $\mathrm{Y}(p)$. In many cases, we will need to refer to the locations of vertices in planar drawings. In those cases, we will treat the vertex v as a point and use $\mathrm{X}(v)$ and $\mathrm{Y}(p)$ to refer to its 2-dimensional location. A sequence of points $p_1, \ldots, p_n$ in the 2-dimensional plane are *x-monotone* if their x-coordinates are non-decreasing as we traverse the list from one end to the other. They are *strictly x-monotone* if their x-coordinates are always increasing as we traverse the list from one end to the other.

Part I

Planar Layered Drawings

Overview of Part I

In the next three chapters, we consider planar layered drawings, addressing the problem of efficiently determining whether or not a given graph has a planar layered drawing on a given number of layers. As we noted in Chapter 1, this problem tends to be $\mathcal{NP}$-hard [51] for most drawing models. To further illustrate the difficulty of obtaining planar layered drawings, we show, in Chapter 5, that the problem is $\mathcal{NP}$-hard when each edge may be drawn with a single bend. Thus, in Chapters 3 and 4, we consider restricted versions of the problem. In Chapter 3, we consider drawings on only three layers and, in Chapter 5, we consider drawing trees rather than general graphs.

In spite of the fact that we consider different problems in each chapter, we solve each inductively on the number of layers, say k, in the drawing. The induction is made possible by the existence of a "special" path in the input graph whose removal creates a graph that can be drawn on $k-1$ layers. We then use this inductive approach to describe corresponding recognition and drawing algorithms and to prove their correctness. The proofs of correctness have two parts: a proof of necessity, such as that a recognition algorithm produces no false negatives, and a proof of sufficiency, such as that a recognition algorithm produces no false positives. Proofs of necessity begin by assuming the existence of a drawing of the input graph on k layers, and then show that the drawing contains a "special" path that "cuts" the drawing into smaller drawings on $k-1$ layers. Our proofs of sufficiency, on the other hand, begin by assuming that the given graph is a yes-instance to the problem, and we then show how to compute a drawing of the graph on k layers. Drawings are computed by first drawing the "special" path which must exist because the graph is a yes-instance, and then by recursively drawing the remainder of the graph, which, we recall, can be drawn on $k-1$ layers.

Chapter 3

Proper Three Layer Drawings

To Sue Whitesides—*there is always something good, and you see it.*

In this chapter, we characterize the graphs that are proper 3-layer planar and derive a corresponding linear-time recognition algorithm. In other words, we show that the PROPER 3-LAYER PLANAR decision problem can be solved in linear time:

Given: A graph G.
Question: Is G a proper 3-layer planar graph?

In addition, we show how to obtain a linear-time drawing algorithm from the recognition algorithm.

Fößmeier and Kaufmann [40] claim to have already obtained a linear-time algorithm for solving this decision problem. They refer to proper 3-layer planar drawings as *BAB-drawings*. This is because the vertices drawn on the top and bottom layers, denoted by B, are not adjacent to one another, and the vertices on the middle layers, denoted by A, are likewise not adjacent to one another. We believe, however, that the algorithm they present is neither correct nor sufficiently specified. We describe these difficulties in Appendix A in detail. Though some of the errors may be easy to fix, a few major problems remain. The linear-time algorithm we present in this chapter is partially inspired by the algorithm of Fößmeier and Kaufmann. Differences are due to the fact our algorithm handles additional difficult cases.

The purpose of this chapter is not only to correct an error in the literature, but also to determine whether or not our approach and that of Fößmeier and Kaufmann is generalizable for drawings with more than three layers. Unfortunately, as we will show, our current approach is complicated, especially considering the fact that we are only considering drawings on three layers. Consequently, we will conclude this chapter by suggesting an alternative approach for future investigations.

We recall from Chapter 1 that Cornelsen, Schank and Wagner [17] also describe a linear-time algorithm for recognizing and drawing graphs on three layers; however, in their drawings, the edges may be non-proper, i.e. edges may cross the middle layer or may connect

vertices on the same layer. In addition, they add the restriction that vertices with degree one drawn on the top or bottom layer must be adjacent to a vertex on the middle layer. Hence, though they consider planar drawings on three layers, the type of the drawings are quite different.

3.1 Three-Layer Planarity Characterization Theorem

We obtain our algorithm from a characterization of proper 3-layer planar graphs. The simplest constraint in our characterization states that proper 3-layer planar graphs are bipartite. As mentioned earlier, the vertices on the top and bottom layers of a proper 3-layer planar drawing can form one bipartition class because no pair of them is adjacent, and the vertices on the middle layer can form the other class because no pair of them is adjacent.

The remaining constraints in the characterization apply to components of the graph. For the restricted case where the input graph G is a tree, the characterization roughly says that G is 3-layer planar if and only if, for each vertex v of G, at most two connected components of $G - v$ are not proper 2-layer planar. The reason is that, if there are three components that are not 2-layer planar, then each component in the drawing uses all three layers so one component is drawn between the other two components. The problem, however, arises from the fact that v is adjacent to all three components so one of this incident edges must cross an edge of the middle component. In the more general case where G may not be a tree, we note that even-length cycles are proper 3-layer planar but not proper 2-layer planar. Figure 3.1(a) gives a proper 3-layer planar drawing of a cycle and Figure 3.1(b) shows why it cannot be drawn on 2 layers. In Figure 3.1(b), vertex c is arbitrarily chosen to be leftmost in the drawing and the illustration shows that every resulting drawing of the cycle contains a crossing.

Consequently, to characterize all 3-layer planar graphs, we must deal with biconnected components. As we will show, we can handle biconnected components by generalizing the way we handle vertices in trees. For example, it is not difficult to see that if C is a biconnected component in a proper 3-layer planar graph G, then $G - C$ contains at most two connected components that are not proper 2-layer planar. Unfortunately, this is not sufficient because not all biconnected components are proper 3-layer planar; consequently, our characterization must provide additional constraints for biconnected components. For example, the biconnected component in Figure 3.2 composed of a cycle and a vertex that is adjacent to three of the cycle vertices is not proper 3-layer planar.

In the following, we describe these constraints more formally and completely. A vertex or biconnected component that satisfies these constraints will be called *safe*. We recall that

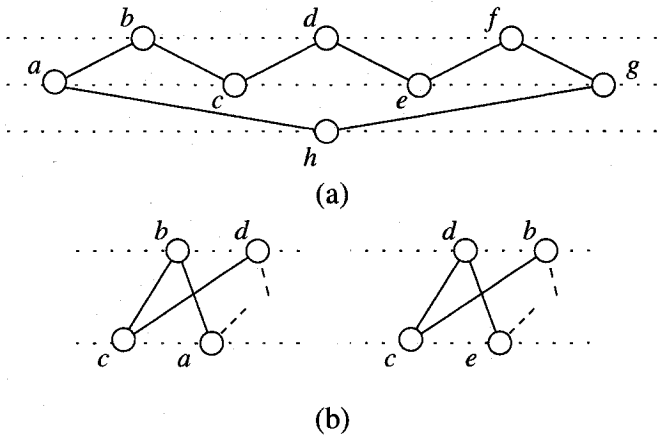


Figure 3.1: A cycle $C = abcdefgh$ has a (a) proper 3-layer planar drawing but not a (b) proper 2-layer planar drawing.

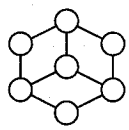


Figure 3.2: A biconnected graph that is not 3-layer planar.

proper 3-layer planar graphs are bipartite so our definitions will apply to bipartite graphs and be given with respect to a given bipartition class of the graph.

Definition 3.1 *Let V_0 be a bipartition class of a bipartite graph G. A vertex v in G is* safe *with respect to V_0 if:*

1. *$v \in V_0$ and $G - v$ contains at most two components that are not caterpillars (e.g see Figure 3.3(b)); or*
2. *$v \notin V_0$ and v has at most four non-leaf neighbors, and:*
   (a) *$G - v$ contains at most two components H_1 and H_2 such that $G(V(H_1) + v)$ and $G(V(H_2) + v)$ are not caterpillars.*
   (b) *if v has four non-leaf neighbors or v belongs to a cycle, then $G - v$ contains at most two components H_1 and H_2 such that $G(V(H_1) + v)$ plus a leaf attached to v and $G(V(H_2) + v)$ plus a leaf attached to v are not caterpillars (e.g. see Figure 3.3(c)).*

A vertex v may be safe but only just "barely safe" because the connected components H_1 and H_2 of $G - v$ mentioned above are not caterpillars. As a result, in every proper 3-

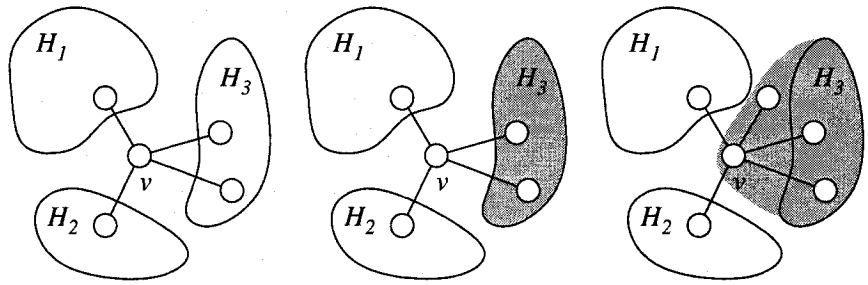


Figure 3.3: (a) Suppose that, for some vertex v in $G = (V_0, V_1; E)$, $G - v$ has three connected components H_1, H_2 and H_3. (b) If H_3 is a caterpillar and $v \in V_0$, then v is safe with respect to V_0. (c) If $v \notin V_0$ and $G(V(H_3) + v)$ plus a leaf attached to v is a caterpillar, then v is safe with respect to V_0.

layer planar drawing of G, one of these components must be drawn to the left of v and the other component must be drawn to the right. Thus, we call v a *connecting vertex* because it "connects" the left part of the drawing with the right part of the drawing.

Definition 3.2 *A vertex v is a* connecting vertex *with respect to V_0 if:*

1. *$v \notin V_0$ and v has four non-leaf neighbors in G; or*
2. *v does not belong to a cycle and $G - v$ contains two components H_1 and H_2 such that $G(V(H_1) + v)$ and $G(V(H_2) + v)$ are not caterpillars; or*
3. *v belongs to a cycle and $G - v$ contains two components H_1 and H_2 such that the graph containing $G(V(H_1) + v)$ plus a leaf attached to v and the graph containing $G(V(H_2) + v)$ plus a leaf attached to v are not caterpillars.*

To describe the properties of a *safe* biconnected component, it is necessary to know how the biconnected component is connected to the remainder of the graph:

Definition 3.3 *Let C be a biconnected component of a bipartite graph $G = (V_0, V_1; E)$. The* extension of *C with respect to vertices v_1 and v_2 in C and V_0 is a graph obtained from C by attaching leaves and pendant 2-paths to certain vertices in C. More specifically, if a vertex of C is adjacent to a leaf in G then, in the extension of C, this vertex is adjacent to a leaf. If a vertex v in C has $d \geq 1$ non-leaf neighbors outside C in G then:*

1. *If $v \neq v_1, v_2$ or $v \notin V_0$, then we attach d pendant 2-paths to v in the extension of C.*
2. *If $v = v_1 = v_2 \in V_0$ and $d \geq 2$, then we attached 2 pendant 2-paths to v in the extension of C.*

3. *Otherwise, we attach exactly one pendant 2-path to v in the extension of C.*

This definition is illustrated in Figure 3.4.

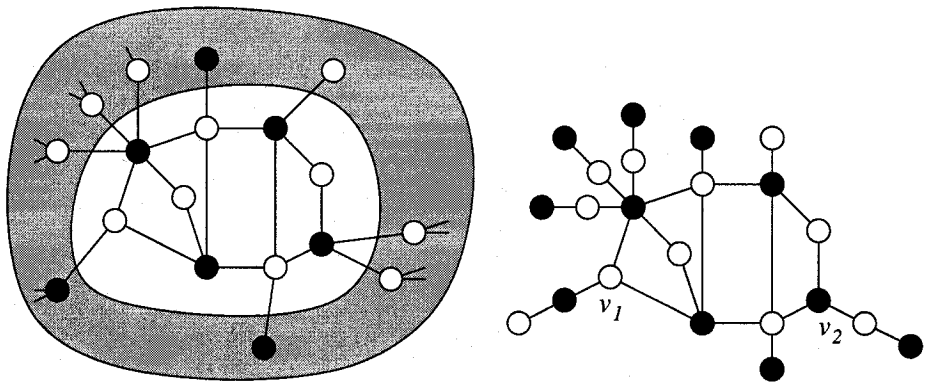


Figure 3.4: (a) A biconnected component C in a bipartite graph $G = (V_0, V_1; E)$ where the darkened vertices belong to V_0, and (b) the extension of C with respect to v_1, v_2 and V_0.

Definition 3.4 *Let C be a biconnected component of a bipartite graph $G = (V_0, V_1; E)$. Biconnected component C is* safe *with respect to V_0 if there exists a* safety certificate *for C with respect to V_0, namely $\langle v_1, v_2, P_1, P_2, \Psi_C \rangle$ consisting of two vertices v_1 and v_2 in C, two simple paths P_1 and P_2 in C_e, the extension of C with respect to v_1, v_2 and V_0, and a planar embedding Ψ_C of C, such that:*

1. *The vertices of P_1 and P_2 in C lie on the external face of Ψ_C;*
2. *P_1 and P_2 each contain a vertex of V_0 on each face of Ψ_C;*
3. *If a vertex of C belongs to V_0, then it belongs to P_1 or P_2 but not both;*
4. *If a vertex v in C has a neighbor outside C, then v belongs to P_1 or P_2;*
5. *If a vertex in C is a connecting vertex, then it is equal to v_1 or v_2;*
6. *Both v_1 and v_2 are end-vertices of the subpaths of P_1 or P_2 in C such that each path from v_1 to v_2 on the external face cycle of C in Ψ_C contains the subpath of P_1 in C or the subpath of P_2 in C;*
7. *If, for some vertex v in C, $G - v$ contains two components H_1 and H_2 vertex-disjoint with C that are not caterpillars, then $v = v_1 = v_2$; and*

8. Each pendant 2-path in C_e belongs to P_1 or P_2.

A safety certificate $\langle v_1, v_2, P_1, P_2, \Psi_C \rangle$ is said to be *tied* if $v_1 = v_2$. We note that the extension of Figure 3.4(b) does not lead to a safety certificate because a vertex has three pendant 2-paths, making it impossible to find paths P_1 and P_2 that contain all pendant 2-paths and each contain a vertex of V_0 on each face. Figure 3.5, however, shows that the biconnected component of Figure 3.4(a) is safe. The safety certificate consists of the vertices labelled v_1 and v_2, the two highlighted paths P_1 and P_2, and the embedding of the component given in the drawing.

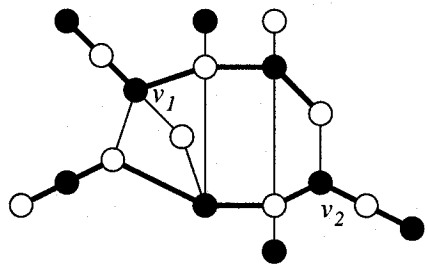


Figure 3.5: The biconnected component of Figure 3.4(a) has a safety certificate $\langle v_1, v_2, P_1, P_2, \Psi \rangle$ where P_1 and P_2 are the highlighted paths.

Based on these definitions, we state our characterization theorem:

Theorem 3.1 *A graph G is proper 3-layer planar if and only if G is bipartite and each vertex and each biconnected component of G is safe with respect to some bipartition class of G.*

In the remainder of this section, we prove this theorem. To begin the proof, we require a few more definitions and preliminary results.

Given a subgraph H of G and a proper 3-layer planar drawing Γ of G, a vertex v is *H-minimum* if it has the smallest x-coordinate of any vertex in H on its layer. A vertex v is *H-maximum* if it has the largest x-coordinate of any vertex in H on its layer. Given another subgraph H' of G, vertex-disjoint from H, we say that H' is *before* (*after*) H in the drawing if for each vertex v' in H' and each vertex v in H on the same layer, the x-coordinate of v' is less (greater) than the x-coordinate of v.

The following result is fundamental to proving our theorem.

Lemma 3.2 *Let Γ be a proper layered drawing of a graph G, and let H be a connected subgraph of G. Suppose that one of the following holds:*

1. *H contains vertices on the top and bottom layers and $G \setminus H$ contains G-minimum and G-maximum vertices.*

2. *H contains G-minimum and G-maximum vertices and $G \setminus H$ contains vertices on the top and bottom layers.*

Then, $V(H)$ is a cut-set for G.

Proof: Let c be a circle containing all of Γ.

Suppose that H contains a vertex u on the top layer and a vertex u' on the bottom layer, and that $G \setminus H$ contains a G-minimum vertex v and a G-maximum vertex v'. Let p_u be the point where the vertical ray from u to positive infinity crosses c, and let $p_{u'}$ be the point where the vertical ray from u' to negative infinity crosses c. Since H is connected, there is simple path P from u to u'. If c_H is the simple, closed curve composed of P, segments $\overline{up_u}$ and $\overline{u'p_{u'}}$, and an arc of c from p_u to $p_{u'}$, then either v is inside and v' is outside or else v is outside and v' is inside the boundary of c_H. In either case, each path P' from v to v' in G crosses c_H. By definition, P' crosses c_H only at points on the drawing of P, so, since Γ is planar, P' contains a vertex of P. Thus, $V(H)$ is a cut-set for G.

Now suppose that H contains a G-minimum vertex u and a G-maximum vertex u', and $G \setminus H$ contains a vertex v on the top layer and a vertex v' on the bottom layer. Let p_u be the point where the horizontal ray from u to negative infinity crosses c, let $p_{u'}$ be the point where the horizontal ray from u' to positive infinity crosses c,

If H is connected, then there is a simple path P from u to u'. If c_G is the simple, closed curve composed of P, segments $\overline{up_u}$ and $\overline{u'p_{u'}}$, and an arc of c from p_u to $p_{u'}$, then either v is inside and v' is outside or else v is outside and v' is inside the boundary of c_G. In either case, each path P' from v to v' in G crosses c_G. By definition, P' crosses c_G only at points in the drawing of P, so, since Γ is planar, P' contains a vertex of P so $V(H)$ is a cut-set for G. □

Recall that a vertex is safe, roughly if $G - v$ contains at most two components that occupy all three layers in any proper 3-layer drawing of G. Lemma 3.2 implies that each vertex in a proper 3-layer planar graph is safe.

Corollary 3.3 *Let Γ be a proper planar layered drawing of a graph G, and let v be a vertex in G. Then, at most two connected components in $G - v$ contain a vertex on the top and bottom layers.*

The next few results describes how paths and cycles are drawn within proper 3-layer planar drawings. A path P in a proper layered drawing of a graph G is *x-monotone* if the

x-coordinates of vertices on the same layer are monotonically increasing as we traverse P from one end to the other.

Lemma 3.4 *Let Γ be a proper 3-layer planar drawing of a graph G. If P is a simple path in G from a P-minimum vertex to a P-maximum vertex, then P is x-monotone in Γ.*

Proof: Suppose, by way of contradiction, u precedes v in P, u and v lie on the same layer in Γ, but v has a smaller x-coordinate than u. We additionally assume that u is the last vertex in P with such a successor in P, and that v is the first such successor of u in P.

Let v' be the predecessor of v in P. Since v is not P-maximum, v also has a successor v'' in P. Let u'' be the successor of u in P.

If v lies on the top layer, then the v', v'' and u'' of u all lie on the middle layer. By assumption, the x-coordinate of u'' is less than the x-coordinate of v'' implying that edges (u, u'') and (v, v'') cross.

Thus, v lies on the middle layer. If v' and v'' lie on the same layer, then, by assumption, v' has a smaller x-coordinate than v''. Since $v' \neq u$, v' has a predecessor in P on the middle layer, whose x-coordinate, by assumption, is greater than or equal to that u. However, this is impossible since such an edge would cross edge (v, v'').

Thus, v' or v'' lies on the top layer and the other on the bottom layer. The first vertex u' of P is P-minimum so it is drawn before the 2-path v', v, v''. Since u is drawn after this 2-path, then, by Lemma 3.2, the subpath of P from u' to u contains v', v or v'', a contradiction. □

The following result states that any proper 3-layer planar drawing of a cycle partitions the cycle into two x-monotone paths, one drawn on the top two layers and the other on the bottom two layers (see, e.g. Figure 3.1).

Lemma 3.5 *Let Γ be a proper 3-layer planar drawing of a graph G. If C is a simple cycle in G, then C contains two distinct vertices u and v such that u is C-minimum on the middle layer, v is C-maximum on the middle layer, one path in C from u to v is x-monotone on the top two layers, and the other path in C from u to v is x-monotone on the bottom two layers.*

Proof: Since C is cycle, then by Lemma 2.1, C occupies at least three layers, and has at least two vertices on the middle layer. Let u be the C-minimum vertex and v the C-maximum vertex on the middle layer. If one path in C from u to v occupies all three layers, then it contains a 2-path occupying all three layers. By Lemma 3.2, this implies that the 2-path is a cut-set for C. However, this is impossible because C is a simple cycle so each path from u to v occupies exactly two layers. By Lemma 3.4, both paths are x-monotone. □

From Lemma 3.5, we obtain several properties about how components attached to a cycle in a graph are drawn in a proper 3-layer planar drawing of the graph.

Corollary 3.6 *Let Γ be a proper 3-layer planar drawing of a graph G. If C is a simple cycle in G, then:*

(a) Any vertex that does not belong to C and is adjacent to a C-minimum (C-maximum) vertex of C on the middle layer is drawn before (after) C.

(b) Every vertex that belongs to C and is adjacent to a non-leaf vertex outside C that is drawn before (after) C is C-minimum (maximum).

(c) Any non-leaf vertex that does not belong to C and is adjacent to a C-minimum (C-maximum) vertex is drawn before (after) C.

(d) Any edge whose end-vertices that do not belong to C is drawn either before C or after C.

(e) At most two vertices of C are adjacent to vertices that do not belong to C and are drawn before (after) C.

(f) If a C-minimum (C-maximum) vertex v has two non-leaf neighbors that do not belong to C and are drawn before (after) C, then there exists a 2-path containing v that is vertex-disjoint with $C - v$ and has a vertex on each layer.

As we will soon see, an important fact about safe biconnected components is that the weak duals of their planar embeddings are paths. A graph is a *weak dual* of a planar embedding if there is a 1-1 mapping between the vertices of the dual and the internal faces of the embedding, and two vertices of the dual are adjacent whenever their corresponding faces in the embedding share an edge. Figure 3.6 shows that weak dual of the planar embedding of the biconnected component shown in Figure 3.4. Since this biconnected component is safe, the weak dual is a path.

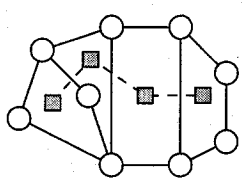


Figure 3.6: A biconnected component and its weak dual.

Lemma 3.7 *Let C be a biconnected component of a bipartite graph G with a safety certificate $\langle v_1, v_2, P_1, P_2, \Psi_C \rangle$ with respect to bipartition class V_0 of G. Then, if a vertex $v \notin V_0$ belongs to C but is not an internal vertex of the subpath of P_i in C for $i = 1$ or 2, then v has at most one neighbor in P_i. In addition, there exists a linear ordering on the internal faces of Ψ_C, such that two faces share an edge if and only if they are consecutive in the ordering.*

Proof: Consider a vertex $v \notin V_0$ in C that is not an internal vertex of some P_i in C. If v has two neighbors in P_i, then there exists an internal face in Ψ_C enclosed by a cycle composed of v and the path between the neighbors of v in P_i. However, this implies that P_i contains all the vertices of V_0 on this face, contradicting the fact that the other path P_{2-i} also contains a vertex of V_0 on this face.

Now, we prove that there is a linear ordering of the internal faces of Ψ_C such that two faces share an edge if and only if they are consecutive in the ordering. Consider a vertex v not on the external face of Ψ_C. By definition, v does not belong to P_1 and P_2 so $v \notin V_0$. Consequently, each neighbor of v belongs to V_0 and belongs to P_1 or P_2. We have just proven that v has at most one neighbor in P_1 and at most one neighbor in P_2.

Thus, each vertex not on the external face of Ψ_C has at most two neighbors, both on the external face of Ψ_C and one in P_1 and the other in P_2. Thus, each internal face of Ψ_C is enclosed by a cycle consisting of a subpath of P_1, a subpath of P_2 and at two paths between P_1 and P_2. Consequently, there exists an ordering on the internal faces of Ψ_C, such that two faces share an edge if and only if they are consecutive in the ordering. This ordering is linear because paths P_1 and P_2 are simple paths. □

Corollary 3.8 *Let C be a biconnected component of a bipartite graph $G = (V_0, V_1; E)$ with a safety certificate $\langle v_1, v_2, P_1, P_2, \Psi_C \rangle$ with respect to V_0. If a vertex $v \in V_0$ or has degree at least three, then v belongs to P_1 or P_2.*

Finally, we are ready to prove our characterization theorem.

3.1.1 Proof of Necessity

Let Γ be a proper 3-layer planar drawing of a graph G. Earlier we showed that, by definition, G is a bipartite graph. Let V_0 be the bipartition class corresponding to the vertices on the top and bottom layers.

3.1.1.1 Safety of Vertices in V_0

Consider a vertex $v \in V_0$. If v is not safe with respect to V_0, then $G - v$ contains three connected components that are each not caterpillars. By Lemma 2.1, each contains a vertex

on the top and bottom layers; however, this contradicts Corollary 3.3. Thus, each vertex in V_0 is safe with respect to V_0.

3.1.1.2 Safety of Vertices Outside V_0

Consider a vertex $v \notin V_0$, and suppose, by way of contradiction, that v is not safe with respect to V_0. If v has five non-leaf neighbors, then v has three non-leaf neighbors either on the top layer or on the bottom layer. Each of these has a neighbor other than v on the middel layer; however, this implies that we have a drawing of a 2-claw on two layers which contradicts Lemma 2.1. Thus, v has at most four non-leaf neighbors and $G - v$ contains at most four connected components of more than one vertex each.

If $G - v$ has at most two connected components of more than one vertex each, then, by Definition 3.1, v is safe, so $G - v$ contains at least three such components.

Suppose that $G - v$ contains exactly three connected components H_1, H_2 and H_3 of more than one vertex each. This implies that v has at most three non-leaf neighbors, and, by Definition 3.1, each $G(V(H_i) + v)$ is not a caterpillar. Since v lies on the middle layer, each H_i has a vertex on the top and bottom layers by Lemma 2.1. Each H_i is connected, so, in fact, each H_i has vertices on each layer. However, this contradicts Corollary 3.3.

Thus, $G - v$ contains exactly four connected components of more than one vertex each. This implies that v has exactly four non-leaf neighbors. Since v is not safe, then by Definition 3.1, three connected components H_1, H_2 and H_3 of $G - v$ are such that, for each $1 \leq i \leq 3$, graph consisting of $G(V(H_i) + v)$ plus a a leaf attached to v is not a caterpillar. Let H_4 be the fourth connected component of $G - v$. By Corollary 3.3, we can assume, without loss of generality, that H_1 has vertices on at most two layers in Γ, say the bottom two layers. By Lemma 2.1, then, each neighbor of v outside H_1 lies on the top layer. However, this implies that the 2-claw consisting of v and three edges in H_2, H_3 and H_4 is drawn on the top two layers. This contradicts Lemma 2.1 so each vertex $v \notin V_0$ is safe with respect to V_0.

3.1.1.3 Safety of Biconnected Components

Finally, we show that each biconnected component C is safe with respect to V_0. Let P be a simple, minimal-length path in G from a G-minimum to a G-maximum vertex. By Lemmas 3.2 and 3.5, P contains at least one vertex in C, so let v_1 be the first vertex of C in P and v_2 the last vertex of C in P. By Lemma 3.4, P is a x-monotone, so, by Corollary 3.6, v_1 is C-minimum and v_2 is C-maximum (see, e.g. Figure 3.7(a)).

From Γ, we derive a proper 3-layer planar drawing Γ_e of C_e, the extension of C with

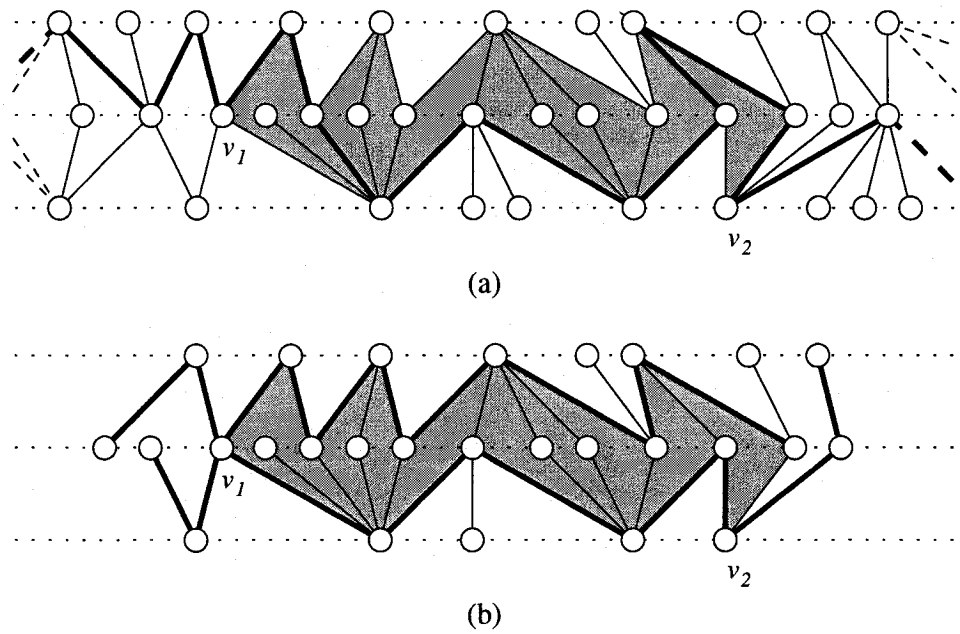


Figure 3.7: (a) A biconnected component in a proper 3-layer planar drawing and (b) the resulting drawing of its extension.

respect to v_1, v_2 and V_0. Let H be the subgraph of G induced by the vertices of C and the vertices connected to C by a single edge or a 2-path. Let Γ_H be the drawing of H inside Γ. Each vertex w in H at distance at least two from each vertex in C corresponds to a leaf end-vertex of an attached 2-path in C_e. If w is not a leaf in H, then it has two or more neighbors at distance one from the same vertex in C. In this case, we modify Γ_H by splitting w into one new vertex for each of its neighbors. To ensure that the resulting drawing is planar, we place the new vertices at the location of w a small distance apart and ordered so that their incident edges do not cross one another. After removing any 2-paths and leaves in the resulting drawing that do not appear in C_e, we obtain a proper 3-layer planar drawing Γ_e of C_e. (see, e.g. Figure 3.7(b)).

By Lemma 3.5, two paths $Q_1 = u_1, u_2, \ldots, u_p$ and $Q_2 = w_1, w_2, \ldots, w_q$ together compose the external face cycle of C in Γ_e such that $u_1 = w_1$ is C-minimum on the middle layer, $u_p = w_q$ is C-maximum on the middle layer, Q_1 lies on the top two layers, and Q_2 lies on the bottom two layers. Let Q_1' be the result of removing each end-vertex of Q_1 that has no non-leaf neighbor outside C on the top layer. Similarly, let Q_2' be the result of removing each end-vertex of Q_2 that has no non-leaf neighbor outside C on the bottom layer. Then, let P_1 be an extension of Q_1' in C_e such that P_1 contains no additional vertices in C and at least one 2-path attached to each end-vertex of Q_1', if they exist. Similarly, P_2 be an extension of Q_2' in C_e such that P_2 contains no additional vertices in C and at least

one 2-path attached to each end-vertex of Q_2', if they exist and do not already belong to P_1. Finally, let Ψ_C be the planar embedding of C induced by the drawing of C in Γ (see, e.g. Figure 3.7(b) where paths P_1 and P_2 are highlighted).

We show that $\langle v_1, v_2, P_1, P_2, \Psi_C \rangle$ is a safety certificate with respect to V_0:

1. *The vertices of P_1 and P_2 in C lie on the external face of Ψ_C.* This statement follows by the construction of P_1 and P_2.

2. *P_1 and P_2 each contain a vertex of V_0 on each face of Ψ_C.* Suppose, by way of contradiction, that P_1 does not contain a vertex of V_0 on face cycle F of Ψ_C. Since P_1 contains a C-minimum and a C-maximum vertex, then, by Lemmas 3.2 and 3.5, P_1 contains an F-minimum vertex w outside V_0 and therefore on the middle layer. By Corollary 3.6, each neighbor of this F-minimum vertex that is outside F is drawn before F. Consequently, the immediate successor of w in P_1 is drawn before F. Thus, by Lemmas 3.2 and 3.5, some successor of w in P_1 contains an F-minimum vertex on the top or bottom layer and therefore a vertex of V_0 in F.

3. *If a vertex of C belongs to V_0, then it belongs to P_1 or P_2 but not both.*

   By construction, P_1 contains the C-minimum and C-maximum vertices on the top layer while P_2 contains the C-minimum and C-maximum vertices on the bottom layer. By Corollary 3.6, then each vertex of C on the top layer belongs to P_1 but not to P_2, and each vertex of C on the bottom layer belongs to P_2 but not to P_1.

4. *If a vertex v in C has a neighbor outside C, then v belongs to P_1 or P_2.*

   Suppose, by way of contradiction, that a vertex v in C has a neighbor outside C but v does not belong to P_1 and P_2. In the previous item, we prove that v does not belong to V_0, so v lies on the middle layer. If v is C-minimum, then by Corollary 3.6, v has a neighbor drawn before C so v belongs to P_1 or P_2. Similarly, if v is C-maximum, then v has a neighbor drawn after C so v belongs to P_1 or P_2. Thus, v is neither C-minimum nor C-maximum. By Corollary 3.6, then, v belongs to the external face of C, which implies that v belongs to P_1 or P_2, a contradiction.

5. *If a vertex in C is a connecting vertex, then it is equal to v_1 or v_2.*

   If $v \notin V_0$ and has four non-leaf neighbors, then, by Corollary 3.6, v is either C-minimum and two non-leaf neighbors are drawn before C or else v is C-maximum and two non-leaf neighbors are drawn after C. Let x and y be these two non-leaf neighbors. By Lemma 3.5, v has another non-leaf neighbor in C drawn on the top layer. If x and y both lie on the top layer, then we have a drawing of a 2-claw drawn

on the top two layers. This is impossible by Lemma 2.1, so x or y lies on the bottom layer. Similarly, v has a non-leaf neighbor in C drawn on the bottom layer so x or y lies on the top layer. Since P contains a G-minimum vertex, then, by Lemma 3.2, P contains one of v, x or y before any other vertex in C. Every path from x or y to a vertex in C contains v so $v = v_1$ or $v = v_2$.

If $v \in V_0$ or v does not have four non-leaf neighbors, then, since v belongs to a cycle in C, $G - v$ contains a connected component H that is vertex-disjoint with C such that the graph H' containing $G(V(H)+v)$ and a leaf attached to v is not a caterpillar.

Suppose that v belongs to V_0. By Corollary 3.6, v has a neighbor in C on the middle layer, so, by Lemma 2.1 and the fact that H' is not a caterpillar, H has a vertex on the top and bottom layers. By Lemma 3.2, then, P contains a vertex from C and a vertex from H. Every path from C to H contains v so $v = v_1$ or $v = v_2$.

Suppose that v does not belong to V_0. By Lemma 3.5, v has a neighbor in C on the top layer so, by Lemma 2.1 and the fact that H' is not a caterpillar, H has a vertex on the bottom layer. Vertex v also has a neighbor in C on the bottom layer so H has a vertex on the top layer. By Lemma 3.2, then, P contains a vertex from C and a vertex from H. Every path from C to H contain v so $v = v_1$ or $v = v_2$.

6. *Both v_1 and v_2 are end-vertices of the subpaths of P_1 or P_2 in C such that each path from v_1 to v_2 on the external face cycle of C in Ψ_C contains the subpath of P_1 in C or the subpath of P_2 in C.*

   By Lemma 3.5, v_1 is C-minimum. Let w be the C-minimum vertex on the middle layer.

   If v_1 does not belong to P_1 or P_2, then v_1 lies on the middle layer and P_1 contains the C-minimum vertex u on the top layer and P_2 contains the C-minimum vertex v on the bottom layer. Furthermore, v_1 has no neighbors drawn before C. Since v_1 is the first vertex of P in C, v_1 is G-minimum, so, by Corollary 3.6, u and v are also G-minimum. By Corollary 3.6, u and v are the only neighbors of v_1 in G so P contains u or v. However, this contradicts the fact that P has minimal length. Thus, v_1 belongs to P_1 or P_2.

   If v_1 is the first vertex in C of neither P_1 nor P_2, then, since v_1 is C-minimum, either v_1 lies on the top layer and w is the first vertex of P_1 in C, or else v_1 lies on the bottom layer and w is the first vertex of P_2 in C. Both cases are symmetric so we consider only the first. By construction, then, w has a neighbor w' drawn before C on the top layer. As a result, v_1 is not G-minimum so the subpath P' of P, from its

G-minimum end-vertex up to but not including v_1, contains at least one vertex and, by Corollary 3.6, is drawn before C. However, this implies that v_1 has a neighbor v_1' drawn before C on the middle layer, which creates an edge-crossing between edges (w, w') and (v_1, v_1').

7. *If, for some vertex v in C, $G - v$ contains two components H_1 and H_2 vertex-disjoint with C that are not caterpillars, then $v = v_1 = v_2$.*

   By Lemma 2.1, H_1 has a vertex on the top and bottom layers and H_2 has a vertex on the top and bottom layers. By Lemma 3.2, then, P contains a vertex in H_1 and H_2. Since P is a simple path, P contains only v in C.

8. *Each pendant 2-path in C_e belongs to P_1 or P_2.*

   By Corollary 3.6, each pendant 2-path in C_e is attached to a C-minimum vertex and drawn before C in Γ_e or else it is attached to a C-maximum vertex and drawn after C in Γ_e.

   We first consider the case where a vertex $v \in V_0$ has two or more attached 2-paths drawn before C. Thus, v lies on the top or bottom layer; both cases are symmetric so we assume that v lies on the top layer. By Corollary 3.6(f), at least one of these 2-paths along with v has a vertex on each layer. Since P contains a G-minimum vertex and a G-maximum vertex, then, by Lemma 3.2, P contains a vertex in this 2-path and a vertex in C. Thus, in G, v is the first vertex of P in C; that is, we have $v = v_1$. Since v belongs to V_0 and has two or more attached 2-paths in C_e, then, by definition, $v = v_1 = v_2$ and exactly two 2-paths are attached to v. Consequently, P is vertex-disjoint with the connected component H of $G - v$ containing vertices of C. By Lemma 3.2, then, H is drawn on at most two layers in Γ, and the corresponding component H_e in C_e is drawn on at most two layers in Γ_e. As a result, H and H_e are caterpillars so at most two other 2-paths are attached to C in C_e. By Corollary 3.6(f), one of these 2-paths is drawn before C and the other is drawn after C in Γ_e. Neither of these is attached to a vertex on the top layer so they both belong to P_2. Likewise, the 2-paths attached to v belong to P_1 because v lies on the top layer.

   It remains then, to consider the case where each vertex of V_0 in C has at most one pendant 2-path drawn before C and at most one pendant 2-path drawn after C in Γ_e. By Corollary 3.6, at most two attached 2-paths are drawn before C and at most two attached 2-paths are drawn after C. If the C-minimum middle layer vertex has a neighbor outside C on the top layer, then by Corollary 3.6, the C-minimum vertex on the top layer has no neighbors drawn before C. Similarly, if the C-minimum vertex

on the middle layer has a neighbor outside C drawn on the bottom layer, then the C-minimum vertex on the bottom layer has no neighbors drawn before C. Symmetric arguments apply to C-maximum vertices, so at most two 2-paths are drawn before C and at most two 2-paths are drawn after C. Therefore, each attached 2-path belongs to either P_1 or P_2.

Thus, C is safe with respect to V_0.

3.1.2 Proof of Sufficiency

Suppose that G is bipartite and that each vertex and each biconnected component of G is safe with respect to some bipartition class V_0 of G. We show that G is proper 3-layer planar by obtaining a proper 3-layer planar drawing Γ of G.

Let $C_1, C_2, \ldots, C_n$ be the biconnected components of G. Each biconnected component C_i is safe with respect to V_0 so there exists a safety certificate $\langle v_1^i, v_2^i, P_1^i, P_2^i, \Psi_i \rangle$ for C_i with respect to V_0.

3.1.2.1 The Special Path

Next, we show that there is a simple path in G that contains each connecting vertex as well as vertices v_1^i and v_2^i in each biconnected component C_i.

Lemma 3.9 *Let C_i be a biconnected component that does not have a tied safety certificate. For each vertex $v \in V_0$, the connected component of $G - v$ containing vertices of C_i is not a caterpillar.*

Proof: If vertex v does not belong to C_i, then the connected component of $G - v$ containing vertices of C_i contains all of C_i so it is not a caterpillar. We assume then that v belongs to C_i.

If v belongs to C_i but does not lie on the boundary of an internal face in Ψ_i, then the connected component of $G - v$ containing vertices of C_i contains a cycle so it is not a caterpillar. We assume then that v belongs to C_i and lies on the boundary of each internal cycle of Ψ_i.

Let Q be the path obtained from the cycle bounding the external face of Ψ_i by removing vertex v and any end-vertex of the resulting path with degree equal to 2 (by Definition 3.4, v is on the external face because $v \in V_0$). By Corollary 3.6, each end-vertex of Q has two neighbors that are not equal to v. Thus, the connected component of $G - v$ containing vertices of C_i is not a caterpillar in each of the following cases:

- a vertex in Q has three non-leaf neighbors outside C_i;
- a vertex in Q has two non-leaf neighbors outside C_i and Q contains at least two vertices;
- a vertex in Q has on non-leaf neighbor outside C_i and the vertex is not an end-vertex of Q.

If none of these cases holds, then let C_e be the extension of C_i with respect to v and V_0. We observe that, in this case, $\langle v, v, P_1, P_2, \Psi_i \rangle$ is a safety certificate for C_i where P_1 is the path in C_e consisting of v and any pendant 2-paths attached to v, and P_2 is the path in C_e containing Q and any pendant 2-paths attached to the end-vertices of Q. However, this contradicts our assumption that C_i has no tied safety certificates. □

In light of Lemma 3.9, we make the following simplifying assumption:

Assumption 3.1 *If biconnected component C_i has a tied safety certificate, then $\langle v_1^i, v_2^i, P_1^i, P_2^i, \Psi_i \rangle$ is a tied safety certificate for C_i; that is, $v_1^i = v_2^i$.*

Now we prove the existence of our path:

Lemma 3.10 *If each vertex and each biconnected component of a bipartite graph G with a planar embedding is safe with respect to a bipartition class V_0 of G, then there exists a simple path containing each connecting vertex in G and each vertex v_j^i, $j = 1, 2$ and $1 \leq i \leq n$.*

Proof: We obtain the result by proving two claims:
Claim 1. For each connecting vertex v, $G - v$ contains at most two connected components that each contain a connecting vertex or a vertex v_j^i.
Claim 2. For each biconnected component C_k, $G - C_k$ contains at most two connected components that each contain a connected vertex or a vertex v_j^i, and, furthermore, one component is adjacent to v_1^k and the other to v_2^k.

We prove Claim 1 by way of contradiction, so we assume that $G - v$ contains three connected components that each contain either a connecting vertex or a vertex v_j^i. Suppose that $v \in V_0$. If a connected component H of $G - v$ contains a vertex v_j^i, then either H contains all of C_i or else v belongs to C_i. If H contains all of C_i, then H is not a caterpillar. If, on the other hand, v belongs to C_i, then, by Lemma 3.9, and the fact that $v \neq v_j^i$, H is not a caterpillar. If H does not contain a vertex v_j^i, then H contains a connecting vertex w that does not belong to a biconnected component. In this case, either w has four non-leaf neighbors so w is the root of a 2-claw subgraph of H or else H contains a connected

component of $G - w$ that together with w is not a caterpillar. Thus, $G - v$ contains three connected components that are each not caterpillars; however, this contradicts the fact that v is safe with respect to V_0.

Now consider the case where $v \notin V_0$. Each of the cases is the same as above, except where v belongs to C_i so H contains $C_i - v$. In that case, $G(V(H) + v)$ contains all of C_i so it is not a caterpillar. Thus, $G - v$ contains three connected components such that, each together with v, is not a caterpillar; however, this contradicts the fact that v is safe with respect to V_0.

We now prove Claim 2, also by way of contradiction. First consider the second part of Claim 2 which states that each component of $G - C_k$ that is not a caterpillar is adjacent to v_1^k or v_2^k. If a connected component H of $G - C_k$ is adjacent to a vertex $v \neq v_1^k, v_2^k$ in C_k and contains a vertex v_j^i or a connecting vertex, $G(V(H) + v)$ is not a caterpillar. However, this implies that v is a connecting vertex, a contradiction of Definition 3.4 because $v \neq v_1^k, v_2^k$.

Therefore, the second part of Claim 2 holds so we have that $G - v_1^k - v_2^k$ contains three connected components disjoint with C_k that each contain a connecting vertex or a vertex v_j^i. We assume, without loss of generality, that two of these components are adjacent to v_1^k. If one of these connected components H contains a vertex v_j^i, then H contains all of C_i except possibly vertex v_1^k. If v_1^k is not in C_i, then H contains all of C_i so it is not a caterpillar. If v_1^k belongs to C_i, then v_1^k has at least five non-leaf neighbors in G so $v_1^k \in V_0$. By Lemma 3.9, then, $C_i - v_1^k$ is not a caterpillar, so neither is H. If, on the other hand, H contains no such vertex v_j^i, then H contains a connecting vertex w that does not belong to a cycle. As a result, H contains a subgraph that is not a caterpillar. Thus, $G - v_1^k$ contains two connected components disjoint with C_k that are not caterpillars. Since C_k is safe, this implies that $v_1^k = v_2^k$. Consequently, v_1^k has at least five non-leaf neighbors so $v_1^k \in V_0$. By Lemma 3.9, the connected component of $G - v_1^k$ containing vertices of C_k is not a caterpillar, so $G - v_1^k$ contains three connected components that are not caterpillars. However, this contradicts the fact that v_1^k is safe. Thus, we have established Claim 2.

Taken together, Claims 1 and 2 imply that there is a path that contains every connecting vertex and every vertex v_j^i in G. □

The previous lemma doesn't help us much if G contains no connecting vertices or bi-connected components. However, in that case G is proper 3-layer planar tree as the next lemma will show. In fact, the proof the lemma serves as a preview for proving sufficiency in the general case.

Lemma 3.11 *If a tree T contains no connecting vertices with respect to some bipartition class V_0 of T, then T is proper 3-layer planar.*

Proof: We prove that T is proper 3-layer planar by producing a drawing, first by drawing a special path and then by inserting drawings of the remaining subtrees.

We obtain our path P inductively so that, after step $i \geq 1$, P is a path in T on i vertices $w_1, w_2, \ldots, w_i$ satisfying the following property: for each $1 \leq j < i$, the subtree T_j of $T - w_j$ containing w_{j+1} does not contain w_1 and $T(V(T_j) + w_j)$ is not a caterpillar.

For $i = 1$, P consists a single non-leaf vertex w_1. Thereafter, we repeatedly extend P until we cannot extend P without violating the property given above. Such a procedure terminates after $1 \leq p \leq |V(T)|$ steps because P is a simple path in T.

Now, consider a vertex w_i in P but outside V_0. Since w_i is not a connecting vertex, w_i has at most three non-leaf neighbors. If only one of these non-leaf neighbors belongs to P, then $i = 1$ or p. This follows from the fact that w_1 is not a leaf by construction, and w_p is also not a leaf because $T(V(T_{p-1}) + v_{p-1})$ is not a caterpillar (recall that T_{p-1} is the connected component of $T - v_{p-1}$ containing v_p). If $w_i = w_1$ or $w_i = w_p$ and two non-leaf neighbors of w_i are outside P, then we simply extend P by adding one of these non-leaf neighbors to P. This completes our construction of P.

Our path P has two important properties:

1. Each vertex of P outside V_0 has at most one non-leaf neighbor outside P.

2. For each subtree T' of $T \setminus P$ adjacent to a vertex v in P, $T(V(T') + v)$ is a caterpillar.

The first property follows by the construction of P so we prove the second property. Let T' be a subtree of $T \setminus P$, adjacent to vertex w_i in P. We show that $H = T(V(T') + w_i)$ is a caterpillar. For $1 \leq i < p$, if H is not a caterpillar, then, since T_i and T' are vertex-disjoint, w_i is a connecting vertex, a contradiction. For $i = p$, if H is not a caterpillar, then, since T' does not contain w_1, the procedure above could have continued past step p by appending the neighbor of w_p in T', a contradiction.

We obtain a proper 3-layer planar drawing of T by first drawing P so that it is x-monotone on the top two layers and the vertices of V_0 in P lie on the top layer. Next, we draw each subtree of T' of $T \setminus P$. If T' is attached to a vertex $v \in V_0$ in P, then, since T' is a caterpillar, there exists a proper 2-layer planar drawing of T' by Lemma 2.1. Thus, we insert such a drawing of T' onto the bottom two layers between the vertices before and after v in P. If T' is attached to a vertex $v \notin V_0$ in P, then we consider two cases. If T' is a leaf, then we draw the leaf vertex on the top layer between the vertices before and after v in P. If T' is not a leaf, then, by the first property of P above, T' is the only such subtree of $T \setminus P$ adjacent to v. By the second property of P, $H = T(V(T') + v)$ is a caterpillar, so we draw H on the bottom two layers at the location of v, which we have already drawn. □

For the remainder of this section (proof of sufficiency), then, we assume that G contains at least one connecting vertex or at least one biconnected component.

As in the proof of Lemma 3.11, we obtain a special path before drawing the graph. In view of our assumption above, Lemma 3.10 implies the existence of a minimal-length path P_0 containing each connecting vertex and each vertex v_i^j in a biconnected component C_i. We may also need to extend that path. If an end-vertex v of P_0 does not belong to V_0, and v is a connecting vertex due to two connected components H_1 and H_2 of $G - v$, then we append to P_0 a neighbor of v in H_1 if it is not in P_0 and, otherwise, we append to P_0 a neighbor of v in H_2. Let P_1 be the resulting path. If an end-vertex v of P_1 does not belong to V_0, and v has two or more non-leaf neighbors outside P_1, then we append one of these non-leaf neighbors to P_1. Let P be the path resulting from these possible additions, if any.

3.1.2.2 Biconnected Components

We begin drawing G by obtaining the drawing Γ_i of each C_i.

Lemma 3.12 *For each $1 \leq i \leq n$, there exists a proper 3-layer planar drawing of C_i in which:*

1. *A vertex in C_i lies on the middle layer if and only if it does not belong to V_0.*
2. *The subpaths of P_1^i and P_2^i in C_i each have one end-vertex that is C_i-minimum and another end-vertex that is C_i-maximum.*
3. *The subpath of P_1^i in C_i lies on the top two layers, and the subpath of P_2^i in C_i lies on the bottom two layers.*

Proof: Let F be an internal face cycle in Ψ_i. By Lemma 3.7, each edge not on the boundary of the external face of Ψ_i either has one end-vertex in P_1^i and the other in P_2^i or else it belongs to an induced 2-path with one end-vertex in V_0 and P_1^i and the other in V_0 and P_2^i. Thus, F is composed of a subpath of P_1^i containing a vertex in V_0, a subpath of P_2^i containing a vertex in V_0, and two paths of length at most two from P_1^i to P_2^i. Thus, there is a proper 3-layer planar drawing of F in which one path from P_1^i to P_2^i consists of F-minimum vertices and the other consists of F-maximum vertices. In addition, the vertices of V_0 and P_1^i in F lie on the top layer and the vertices of V_0 and P_2^i in F lie on the bottom layer. Finally, each path between P_1^i and P_2^i lies on at most two internal faces of Ψ_i; thus, we obtain a proper 3-layer planar drawing of C_i by merging such drawings of the internal faces of Ψ_i. □

We observe that, as we traverse P from one end to the other, the vertices of each biconnected component are consecutive, and the vertices of two biconnected components overlap at a single shared vertex. Without loss of generality, then, we assume that in P we encounter the vertices of component C_1 first, then C_2, and so on until we reach the vertices of C_n last. We also assume that we visit each v_1^i before v_2^i. We then insert each of these drawings in this left-to-right order into Γ. For an example, see Figure 3.8(a).

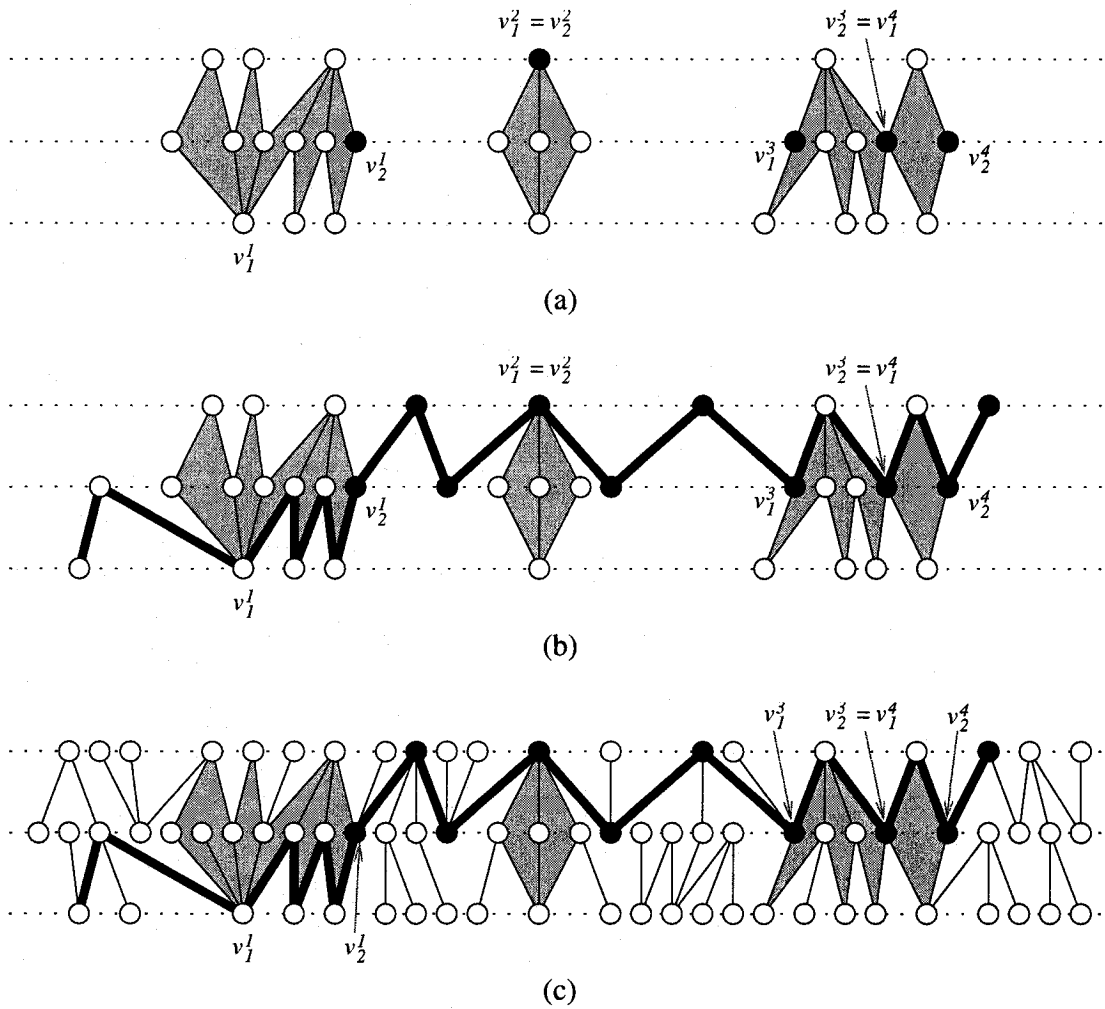


Figure 3.8: Three main steps for obtaining a proper 3-layer planar drawing of a graph: (a) draw the biconnected components separately, (b) draw the remainder of the special path, and (c) draw the pendant subtrees.

Next, we draw the remainder of P. For convenience, we draw the subpath of P between each pair of vertices v_2^i and v_1^{i+1}, for $1 \leq i < n$, so that it is strongly x-monotone and entirely on the top two layers or entirely on the bottom two layers. This is possible if the subpath of P_j^i that contains v_2^i occupies the same two layers as does the subpath of P_k^{i+1}

that contains v_1^{i+1}. We satisfy this property by reflecting some of the inserted biconnected component drawings about the horizontal.

We have now drawn all of P except for the subpath before C_1 and the subpath following C_n. If the subpath of P before C_1 consists of a single vertex adjacent to a vertex v in C_1 on the top or bottom layer, then we draw the leaf inside C_1 between the vertices next to v in the external face cycle of C_1. Otherwise, we draw it before C_1 on the two layers occupied by the subpath of P_j^1 in C_1 that contains v_1^1. Likewise, if the subpath of P after C_n consists of a single vertex adjacent to a vertex v in C_n on the top or bottom layer, then we draw the leaf inside C_n between the vertices next to v in the external face cycle of C_n. Otherwise, we draw it after C_n on the two layers occupied by the subpath of P_j^n in C_n that contains v_2^n. For an example, see Figure 3.8(b).

It remains for us to draw pendant trees attached to vertices in biconnected components and vertices in P. This step is illustrated in Figure 3.8(c).

3.1.2.3 Pendant Trees

For convenience, we assign a direction to each path P_j^i, $j = 1, 2$ and $1 \leq i \leq n$, so that if a vertex of P_j^i is C_i-minimum but not C_i-maximum, then it appears before vertices of P_j^i that are not C_i-minimum, and, if a vertex of P_j^i is C_i-maximum but not C_i-minimum , then it appears after vertices of P_j^i that are not C_i-maximum.

We first draw each pendant tree T that is attached to a vertex v in a biconnected component C_i such that T is not a leaf and $v \neq v_1^i, v_2^i$. By Lemma 3.12, v is C_i-minimum or C_i-maximum. Since v is not a connecting vertex and C_i is not a caterpillar, the tree containing $G(V(T) + v)$ and a leaf adjacent to v is a caterpillar. In other words, T has a proper 2-layer planar drawing Γ_T in which the neighbor of v in T is T-minimum. Each such tree is by definition uniquely associated with a pendant 2-path attached to v in C_e, the extension of C_i with respect to v_1^i, v_2^i and V_0. Furthermore, the 2-path belongs to $P_j^i = P_1^i$ or P_2^i. We then insert Γ_T into the drawing of G on the same two layers occupied by P_j^i. If v is the first vertex of C_i in P_j^i (recall that we assigned a direction to P_j^i), then we insert Γ_T before C_i and, otherwise, we insert Γ_T after C_i. Drawing each such tree in this way does not create any edge crossings because, by Lemma 3.12, each P_j^i contains at most two pendant 2-paths in C_i, one attached to a C_i-minimum vertex and another attached to a C_i-maximum vertex. Thus, the drawing of T does not overlap the drawing of P or any other pendant tree attached to a vertex in C_i.

Now consider each pendant tree that is a leaf vertex w attached to a vertex v in a biconnected component C_i. If $v \in V_0$, then we draw (v, w) so that w lies inside C_i between the two neighbors of v in C_i on the middle layer. These neighbors exist by Lemma 3.12.

Since the drawing of G so far is planar, drawing (v, w) in this way does not create any edge-crossings. If $v \notin V_0$ and v is neither C_i-minimum nor C_i-maximum, then we draw (v, w) so that w lies between the neighbors of v in C_i that both lie on the top layer or both lie on the bottom layer. These neighbors exist by Lemma 3.12. Once again, since the drawing of G so far is planar, drawing (v, w) in this way does not create any edge-crossings. Finally, if $v \notin V_0$ and v is C_i-minimum (C_i-maximum), then, by definition, v belongs to $P_j^i = P_1^i$ or P_2^i. We draw edge (v, w) so that v and w occupy the same two layers as P_j^i and w lies immediately before (after) C_i. Drawing (v, w) in this way does not create any edge-crossings because if any vertex attached to a vertex in C_i has been drawn on the same layer as w and on the same side as of C_i as w, then it is adjacent to v.

The only remaining vertices to draw belong to pendant trees attached to vertices in P that, if these vertices in P belongs to a biconnected component C_i, then they belong to V_0 and are equal to v_1^i or v_2^i.

Let T be a pendant tree attached to a vertex v in V_0; in other words, v lies on the top or bottom layer. None of the vertices of T belong to P, so no vertex of T is a connecting vertex. Since P contains a vertex from a biconnected component or a connecting vertex, T is a caterpillar. If v belongs to a biconnected component C, then we draw T on the two layers opposite v, immediately before C if v is C-minimum, and after C, otherwise. If v does not belong to a biconnected component, then we draw T on the two layers opposite v between the vertices before and after v in P.

Finally, consider each vertex v in P but outside V_0. Let H be the induced subgraph of G containing vertex v and the components of $G - v$ vertex-disjoint with P. Since v does not belong to a biconnected component and v is safe, then there exist at most two connected components H_1 and H_2 of $G - v$ such that $G(V(H_i) + v)$ is not a caterpillar. According to the construction of P, the vertex before v in P belongs to, say, H_1 and the vertex after v in P belongs to H_2. Thus, H is a caterpillar so we draw this caterpillar at the location of v on the middle layer and the layer unoccupied by the vertices of P next to v.

The only remaining vertices adjacent to v are leaves. We draw them between the vertices of P next to v.

This concludes our proof of Theorem 3.1.

3.2 3-Layer Planarity Testing

Our proper 3-layer planarity recognition and drawing algorithm is derived directly from Theorem 3.1. For convenience, we first derive a recognition algorithm and then show how to modify it so as to obtain a drawing algorithm.

The algorithm simply applies Theorem 3.1.

ISPROPER3LAYERPLANAR(G)

Input: An undirected graph G.

Output: TRUE if G is 3-layer planar; otherwise, FALSE.

1. **if** G is not bipartite **then return** FALSE
2. $C_1, C_2, \ldots, C_p \leftarrow$ the biconnected components of G.
3. COMPUTECATERPILLARS(G).
4. **for** each bipartition class V_0 of G **do**
5. **for** each vertex v in G **do**
6. **if** ISSAFEVERTEX(G, V_0, v) = FALSE **then goto** line 4.
7. $S \leftarrow$ CONNECTINGVERTICES(G, V_0).
8. **for** each biconnected component C_i of G **do**
9. **if** ISSAFECOMPONENT(G, V_0, C_i, S) = FALSE **then goto** line 4.
10. **return** TRUE.
11. **return** FALSE.

The definitions of safe vertices and biconnected components depend on knowing whether or not various subgraphs of the input graph G are caterpillars or not. Procedure COMPUTECATERPILLARS below computes all of the necessary facts using dynamic programming on the block-cut tree of G. A *block-cut tree* of a graph is a tree whose vertices correspond to cut-vertices and biconnected components in the graph. Edges in the block-cut tree connect biconnected components with the cut-vertices that they contain in the graph.

COMPUTECATERPILLARS(G)

Input: An undirected graph G.

Output: For each vertex v and each connected component H of $G - v$, determines whether H, $H' = G(V(H) + v)$ and $H'' = H'$ plus a leaf attached to v are caterpillars.

1. $T \leftarrow$ the block-cut tree for G. ▷*A linear-time algorithm for computing block-cut trees is given in [92].*
2. For each vertex v and connected component H of $G-v$, determine whether H, H' and H'' are caterpillars. ▷*We can compute this for the entire graph in linear time using two traversals of* T.

Using the results computed by COMPUTECATERPILLARS, procedures ISSAFEVERTEX and CONNECTINGVERTICES simply apply Definitions 3.1 and 3.2, respectively. Therefore, we omit the definitions of these procedures here.

Determining whether or not a biconnected component is safe requires a little more than simply applying the definitions.

IsSafeComponent(G, V_0, C, S)

Input: A bipartite graph G with bipartition class V_0, a biconnected component C of G, and the set of connecting vertices S of G with respect to V_0.

Output: TRUE if C_i is safe with respect to V_0; otherwise, FALSE.

1. $C_\Psi \leftarrow$ a simple cycle in C containing all vertices of V_0 or vertices with degree three.
2. **if** $\nexists C_\Psi$ **then return** FALSE.
3. $\Psi_C \leftarrow$ a planar embedding of C with C_Ψ on the external face.
4. **if** $\nexists \Psi_C$ **then return** FALSE.
   ▷*See Corollary 3.8.*
5. ... continued below ...

We note that the only vertices that do not belong to C_Ψ belong to V_0 and have degree equal to 2. Consequently, all planar embeddings of C with C_Ψ on the external face are combinatorially equivalent. The only difference between embeddings is the ordering of pairs of vertices of degree two that have the same two neighbors. Thus, we need only consider one planar embedding Ψ_C of C.

Given Ψ_C, we now determine if we can fill in the remainder of a safety certificate for C. By Definition 3.4, the two paths P_1 and P_2 in any safety certificate contain vertex-disjoint subpaths in C_Ψ that each contain a vertex of V_0 on each face of Ψ_C. This implies, that if C is safe, then there are two unique minimal-length, vertex-disjoint subpaths Q_1 and Q_2 of C_Ψ that contain a vertex of V_0 on each face of Ψ_C. The next step in the algorithm checks for the existence of these paths. Based on the constraints, this test can be applied in linear time in the size of C.

5. $Q_1, Q_2 \leftarrow$ two minimal-length, vertex-disjoint subpaths of C_Ψ that each path contains at least one vertex of V_0 on each face of Ψ_C.
6. **if** $\nexists Q_1, Q_2$ **then return** FALSE.
7. ... continued below ...

Also by Definition 3.4, paths P_1 and P_2 in a safety certificate contain all pendant 2-paths in the extension of C. By Definition 3.3, if a vertex has a non-leaf neighbor outside C, then it is adjacent to a pendant 2-path in the extension of C. Therefore, each vertex with a non-leaf neighbor outside C is an end-vertex of a subpath of P_1 or P_2 in C.

7. **if** there are more than four vertices with non-leaf neighbors outside C **then return** FALSE.
8. **if** a vertex with a non-leaf neighbor outside C is internal to Q_1 or Q_2 **then return** FALSE.
9. ... continued below ...

A safety certificate contains two vertices v_1 and v_2 in C_Ψ such that, every path from v_1 to v_2 in C_Ψ contains the subpath of P_1 or P_2 in C. By the definition of Q_1 and Q_2, $C_\Psi = Q_1Q_1'Q_2Q_2'$ for subpaths Q_1' and Q_2' of C_Ψ such that Q_1, Q_2, Q_1', Q_2' contain all of the edges in C_Ψ. Thus, if v_1 and v_2 belong to a safety certificate, then, without loss of generality, v_1 belongs to Q_1' and and v_2 belongs to Q_2'. Using these observations, we now either prove that C has a tied safety certificate or rule out such a certificate and determine whether or not C has a non-tied certificate.

If a certificate is tied, then Q_1 or Q_2 contains exactly one vertex v that belongs to V_0 and $v_1 = v_2 = v$.

9. **for** $j = 1, 2$ **do**
10. **if** $Q_j = v$ for some vertex v and $v \in V_0$ **then**
11. $v_1, v_2 \leftarrow v$.
12. $C_e \leftarrow$ the extension of C with respect to v_1, v_2 and V_0. ▷*See Definition 3.3.*
13. $P_1 \leftarrow v$ plus the at most 2 pendant 2-paths attached to v in C_e.
14. $P \leftarrow C_\Psi - v$.
15. $P' \leftarrow P$ minus any end-vertex with degree equal to 2.
16. $P_2 \leftarrow P'$ plus up to one pendant 2-path attached to each end-vertex of P'.
17. **if** $\langle v_1, v_2, P_1, P_2, \Psi_C \rangle$ is a safety certificate wrt V_0 **then return** TRUE.
18. ... continued below ...

According to Definition 3.4, there is one case in which every certificate must be tied. Since we have ruled such a certificate out, if that case still holds, then C is not safe.

18. **if** $\exists$ vertex v in C such that $G - v$ contains two non-caterpillar components H_1 and H_2 vertex-disjoint with C **then return** FALSE.
19. ... continued below ...

Now, we attempt to find a non-tied certificate for C. Above we showed that, in a certificate, v_1 belongs to Q_1' and v_2 belongs to Q_2'. We now further show that, without loss of generality, we can assume that each v_i is equal to a vertex in C with a non-leaf neighbor outside C or to an end-vertex of Q_1' or Q_2'.

Lemma 3.13 *If C is safe with respect to V_0, then there exists a safety certificate $\langle v_1, v_2, P_1, P_2, \Psi_C\rangle$ for C with respect to V_0 in which each v_i has a non-leaf neighbor outside C or is an end-vertex of Q'_1 or Q'_2.*

Proof: Let $\langle v'_1, v'_2, P'_1, P'_2\Psi_C\rangle$ be a safety certificate for C with respect to V_0. If v'_1 does not have the desired properties, then v'_1 has degree 2. If v'_1 has degree equal to three, then v'_1 has an incident edge that does not belong to C_Ψ. However, this implies that the edges incident on v'_1 in C_Ψ lie on the boundary of different internal faces of Ψ_C. However, since v'_1 belongs to neither Q_1 nor Q_2, Q_1 and Q_2 do not contain vertices of V_0 on each of these faces, a contradiction. Therefore, v'_1 has exactly two neighbors. If Q'_1 contains no vertices with non-leaf neighbors outside C, then we obtain a new certificate by setting v'_1 to be the end-vertex common to Q_1 and Q'_1, shorten P'_1 so that v'_1 is its new end-vertex and length P'_2 so that it contain all of $Q'_1 - v'_1$. On the other hand, if Q'_1 contains a vertex adjacent to a non-leaf vertex outside C, then we set v'_1 to be that vertex. In this case, the only change to P'_1 or P'_2 may be to remove any pendant 2-path that is removed from the extension of C by changing v'_1. □

Since there are at most four vertices with non-leaf neighbors outside C, and at most four end-vertices of Q'_1 and Q'_2, the possible cases for v_1 and v_2 that we need to consider is bounded by a small constant. Therefore, we consider each case and determine if any leads to a safety certificate. In each case, we compute the corresponding extension of C and then determine if we can find the necessary paths P_1 and P_2 to obtain a safety certificate. This part can be done efficiently because Definition 3.4 severely restricts acceptable pairs of paths P_1 and P_2 once v_1, v_2 and the planar embedding are given.

19. **for** each pair of vertices v_1 and v_2 in C such that v_1 is either an end-vertex of Q'_1 or a vertex in Q'_1 with a non-leaf neighbor outside C, v_2 is either an end-vertex of Q'_2 or a vertex in Q'_2 with a non-leaf neighbor outside C, and each connecting vertex in C is equal to v_1 or v_2 **do**
20. $C_e \leftarrow$ the extension of C with respect to v_1, v_2 and V_0.
21. **if** $\exists$ a pair of paths P_1 and P_2 in C_e such that the subpath of P_1 in C contains Q_1 and is a subpath of a path in C_Ψ from v_1 to v_2, the subpath of P_2 in C contains Q_2 and is a subpath of a path in C_Ψ from v_1 to v_2, each vertex of V_0 or with degree three in C belongs to P_1 or P_2, and each pendant 2-path in C_e belongs to P_1 or P_2 **then return** TRUE.
22. **return** FALSE.

Hence, we obtain the following result:

Theorem 3.14 *Algorithm* ISPROPER3LAYERPLANAR *determines whether or not a graph is proper 3-layer planar in linear time.*

We can transform this recognition algorithm into a drawing algorithm by returning a proper 3-layer planar drawing anytime the recognition algorithm returns TRUE. The drawing is constructed as described in the sufficiency proof of Theorem 3.1 which is illustrated in Figure 3.8. To do this, we require the set of connecting vertices and a safety certificate for each biconnected component. Fortunately, as described above, the recognition algorithm already computes all of these things. We recall, however, that the sufficiency proof requires that all safety certificates be tied whenever possible (see Assumption 3.1). Once again, our recognition algorithm first determines whether or not the biconnected component has a tied certificate before considering other certificates.

We describe the remainder of the drawing algorithm as follows:

DRAWPROPER3LAYERGRAPH(G, V_0, S, R)

Input: A proper 3-layer planar bipartite graph $G = (V_0, V_1, E)$ with connecting vertices S with respect to V_0 and a safety certificate $\langle v_1^i, v_2^i, C_i, \Psi_i \rangle \in R$ with respect to V_0 for each biconnected component C_i in G.

Output: A proper 3-layer planar drawing of G with the vertices of V_0 on the top and bottom layers and the vertices of V_1 on the middle layer.

1. **if** G contains no connecting vertices or biconnected components **then**
2. Draw G as described in the proof of Lemma 3.11.
   ▷*The drawing can be computed in linear time using the results computed by* COMPUTECATERPILLARS.
3. **else** ▷G *contains a connecting vertex or a biconnected component.*
4. $P_0 \leftarrow$ a path containing all connecting vertices and all vertices v_j^i from the safety certificates.
   ▷*This path exists by Lemma 3.10 and can be found efficiently using Claims 1 and 2 in the proof and the results computed by* COMPUTECATERPILLARS.
5. $P_1 \leftarrow P_0$ with at most one additional vertex attached to the end-vertices of P_0, as described on page 47.
6. $P \leftarrow$ the path computed from P_1 as described on page 47.
7. ▷*Can be done efficiently using the results computed by* COMPUTECATERPILLARS.
8. **for** each biconnected component C_i **do**
9. $\Gamma_i \leftarrow$ the drawing of C_i described by Lemma 3.12.

10. Insert each Γ_i into the drawing so that P can be drawn x-monotone.
11. Draw the remaining subpaths of P (those that do not belong to biconnected components), as described on page 49.
12. Draw the remaining pendant trees and insert their drawings into the main drawing as described in Section 3.1.2.3.

Since this drawing algorithm simply follows the sufficiency proof of our characterization theorem, Theorem 3.1, it creates a proper 3-layer planar drawing of the input graph, whenever one exists. We call the resulting algorithm PROPER3LAYERDRAWER. Throughout our description of the algorithm, we show that the running-time of each non-trivial step is linear, so we obtain the following result:

Theorem 3.15 *Algorithm* PROPER3LAYERDRAWER *determines whether or not a graph is proper* 3*-layer planar, and, if so, obtains a proper* 3*-layer planar drawing of the graph, all in linear time.*

3.3 Conclusions and Open Problems

We have shown that the PROPER 3-LAYER PLANAR decision problem can be solved and proper 3-layer planar drawings obtained all in linear time. Unfortunately, as can be seen by the length of this chapter, the difficulty of obtaining these algorithms, though they are linear, is much greater than for obtaining recognition and drawing algorithms for two layers. Because of this, it seems unlikely that our present approach will generalize easily to solving the same problems for four layers. Since our approach is partially based on the algorithm of Fößmeier and Kaufmann [40], it seems equally unlikely that their approach will easily generalize to four layers.

Another approach that may generalize uses graph operations that reduce the graph to an empty graph if and only if it has a planar drawing on a given number of layers. Arnborg and Proskurowski [3] use this approach to recognize graphs of treewidth three, and Matousek and Thomas [69] modify their set of reductions to obtain an efficient quadratic-time recognition algorithm. Generalizing these results, Sanders [85] shows that this approach can be use to efficiently recognize graphs with treewidth four. Our hope is to similarly find a set reductions for proper 3-layer planar graphs and likewise use them to obtain reductions for proper 4-layer planar graphs. We believe that this approach might be successful because Dujmović *et al.* [24] use pathwidth, a restricted version of treewidth, to obtain algorithms for recognizing k-layer planar graphs (inefficient algorithms though they may be). In addition to this, in Chapter 4 of this thesis, we show how to use pathwidth to obtain

planar drawings of trees on a minimum number of layers. These results show that pathwidth and hence treewidth are closely related to the number of layers required to obtain planar drawings of graphs.

Chapter 4

Tree Drawings

To David Wood—*you have many good ideas, and you express them elegantly.*

In this chapter we study the problem of obtaining planar drawings of trees on a minimum number of layers with respect to the pathwidth of the tree. We consider proper, short, upright and unconstrained drawings (see Chapter 2 for definitions) and obtain optimal bounds on the number of layers for each. We also give linear-time algorithms for obtaining layered drawings that match these bounds.

To our knowledge, these algorithms are the first practical algorithms for minimizing layers in planar drawings of undirected graphs. We hope that the ideas behind our algorithms will lead to efficient algorithms for drawing other more general classes of graphs. The only other such algorithm for minimizing layers is due to Dujmović *et al.* [24]. However, as we mention in Chapter 1, though their algorithm is general and polynomial, its running time is too large for the algorithm to be useful in practice.

The results in this chapter correct a claim by Felsner *et al.* [37] that a tree with pathwidth k is (unconstrained) k-layer planar. Though their result and proof of it seem plausible, it turns out to be oversimplified. Indeed, we will present examples of trees whose drawings occupy at least $\lceil 3h/2 \rceil$ layers.

4.1 Preliminaries

Because we are considering only planar drawings in this part of the thesis, we often omit the term 'planar' for convenience of exposition.

We number the layers in our drawings consecutively from 1 to h, with layer 1 as the top layer and layer h as the bottom layer. The following simple lemma states one of the key observations that we use to establish lower bounds on the number of layers in a drawing.

Lemma 4.1 *In any unconstrained h-layer planar drawing of a tree T with a vertex v, the drawings of at most two components of $T \setminus v$ occupy h layers.*

Proof: Assume the contradiction, that $T \backslash v$ contains at least three components T_1, T_2 and T_3 whose drawings each occupy h layers. Each T_i occupies all h layers so each has a vertex v_i on the top layer. Assume without loss of generality that $\mathrm{X}(v_1) \leq \mathrm{X}(v_2) \leq \mathrm{X}(v_3)$. However, T_2 has another vertex v_2' on the bottom layer so the drawing is not planar: an edge in the path from v_1 to v_3 in $T \setminus T_2$ crosses an edge in the path from v_2 to v_2' in T_2. □

From Lemma 4.1, we obtain the following result about layered drawings of complete ternary trees, which was already proven by Felsner *et al.* [37]:

Corollary 4.2 *Every unconstrained planar layered drawing of a complete ternary tree of depth $d \geq 0$ occupies at least $d + 1$ layers.*

Proof: The proof is by induction on the depth d of the tree beginning at depth $d = 0$ where the tree consists of a single vertex. □

We also obtain the following bounds for upright layered drawings of nearly complete ternary trees. A *nearly complete ternary tree* of depth d is obtained from a complete ternary tree of depth d by removing exactly two children from each vertex at depth $d - 1$.

Corollary 4.3 *Every upright planar layered drawing of a nearly complete ternary tree of depth $d \geq 0$ occupies at least $d + 1$ layers.*

Proof: The proof is by induction on the depth d of the tree. The result differs from Corollary 4.2 because in upright drawings the endpoints of edges occupy different layers. □

Nearly all of the remaining results depend on the pathwidth of the given tree. A *path decomposition* B of a graph G is a sequence $B_1, B_2, \ldots, B_p$ of subsets of $V(G)$ that satisfies the following three properties:

1. $\bigcup_{1 \leq i \leq p} B_i = V(G)$;
2. for every edge $(u, v) \in E(G)$, there is a subset B_i such that both $u, v \in B_i$; and
3. for all $1 \leq i < j < k \leq p$, $B_i \cap B_k \subseteq B_j$.

The *width* of B is $\max\{|B_i| \mid 1 \leq i \leq p\} - 1$. The *pathwidth* of a graph G, denoted $\mathrm{pw}(G)$, is the minimum width of a path decomposition of G.

Both Scheffler [87] and Ellis *et al.* [34] give linear-time algorithms for computing the pathwidth of trees. Both algorithms depend on the following fundamental result about trees and pathwidth:

Lemma 4.4 ([87,34]) *A tree T has pathwidth at most h if and only if for all vertices v in T at most two components of $T \setminus v$ have pathwidth h and the remainder have pathwidth at most $h-1$.*

As defined by Ellis *et al.* [34], we say that a vertex v is *h-critical* in a rooted tree T if exactly two subtrees rooted at children of v have pathwidth h and the remainder have pathwidth at most $h-1$. They prove the following two results about critical vertices:

Lemma 4.5 ([34]) *Let T be a tree rooted at r. If at most two subtrees rooted at children of r have pathwidth h, neither has an h-critical vertex, and every other subtree rooted at a child of r has pathwidth at most $h-1$, then $\mathrm{pw}(T) \leq h$.*

Lemma 4.6 ([34]) *A tree T has at most one $\mathrm{pw}(T)$-critical vertex.*

In the next section we prove optimal upper and lower bounds on the number of layers required by short layered drawings of trees. We follow that with bounds for proper, upright and unconstrained layered drawings in Sections 4.3 and 4.4, and finally, in Section 4.5, we give linear-time algorithms for obtaining short, proper, upright and unconstrained drawings matching the upper bounds.

We prove our upper bounds by constructing drawings of trees. Similar to Felsner *et al.* [37], the drawings are constructed in two steps: we draw one or more paths in the tree and then recursively draw the remaining components next to the previously drawn paths. The most important of these paths is the *main path*. A *main path* P of a tree T is a path such that the pathwidth of $T \setminus P$ is at most $\mathrm{pw}(T) - 1$.

Lemma 4.7 *Every tree has at least one main path.*

Proof: Consider a path decomposition $B = B_1, B_2, \ldots, B_p$ of T of minimum width. Let v_1 be a vertex in B_1, v_p a vertex in B_p, and P the path between v_1 and v_p. By definition, each B_i contains at least one vertex in P so, if we remove the vertices of P from each B_i, then the result is a path decomposition of $T \setminus P$ with width at most $\mathrm{pw}(T) - 1$. □

The remaining components must be drawn so that we can insert the edges connecting the components to the previously drawn paths without creating crossings. As a result, if Γ is a drawing of a component and vertex v in the component is adjacent to a previously drawn path vertex, then v lies on the top or bottom layer of Γ. We say that v is *exposed* in Γ. In general, a vertex $v \in T$ is *exposed* in a layered drawing of a tree T if v lies on the top or bottom layer of the drawing.

The next result illustrates how to obtain proper 2-layer planar drawings of certain trees given one of their main paths.

Lemma 4.8 *For every tree T with $\mathrm{pw}(T) \leq 1$, there exists a proper 2-layer planar drawing.*

Proof: By Lemma 4.7, T has a main path P, and $T \setminus P$ consists of vertices with degree zero. We draw T by first drawing P on both layers and then inserting each vertex v in $T \setminus P$ adjacent to a vertex $w \in P$ on the layer opposite w. □

4.2 Short Layered Drawings

Using a similar though slightly more complicated drawing algorithm than in Lemma 4.8, we obtain an upper bound for short layered drawings, first proved by Dujmović *et al.* [27]:

Lemma 4.9 *Every tree T with $\mathrm{pw}(T) \geq 2$ has a short $(2\mathrm{pw}(T)-1)$-layer planar drawing.*

Proof: We obtain a short $(2\mathrm{pw}(T) - 1)$-layer planar drawing by first drawing a main path P of T on the top layer. Each component T' in $T \setminus P$ has pathwidth at most $\mathrm{pw}(T) - 1$. Let v be the vertex in T' adjacent to a vertex w in P. We insert a drawing of each T' onto the $(2\mathrm{pw}(T) - 2)$ layers below w and then draw the missing edge (v, w) as a straight line between v and w. To avoid edge-crossings, we draw T' so that v lies on the layer immediately below w; that is, we draw T' so that v is exposed. It remains to prove, then, that such drawings of T' exist. In other words, we must prove that for any tree T with $\mathrm{pw}(T) \geq 1$ and vertex $v \in T$, there exists a short $2\mathrm{pw}(T)$-layer planar drawing of T in which v is exposed.

The proof is by induction on the pathwidth of T. When $\mathrm{pw}(T) = 1$, there is a short 2-layer planar drawing of T by Lemma 4.8. Since there are only two layers, every vertex including v is exposed. Suppose that $\mathrm{pw}(T) \geq 2$ and let $P = v_1 v_2 \ldots v_n$ be a main path in T and R the path between v and a vertex v_i in P. We begin drawing T on $2\mathrm{pw}(T)$ layers by first drawing the path $Rv_iv_{i-1}\ldots v_1$ on the top layer and then the path $v_{i+1}v_{i+2}\ldots v_n$ on the second layer below edge (v_i, v_{i-1}). Each connected component T' of $(T \setminus P) \setminus R$ has pathwidth at most $\mathrm{pw}(T) - 1$ so, by induction, there exists a short $(2\mathrm{pw}(T) - 2)$-layer planar drawing of T' in which vertex $v' \in T'$ adjacent to vertex w in $P \cup R$ is exposed. We recursively construct and then insert this drawing of T' onto the layers below w. The final drawing is illustrated in Figure 4.1. □

This upper bound is optimal for a set of rooted trees that we define below. We construct the trees in such a way that we know the location of designated vertices in any minimum layered drawing. The following lemma describes the key construction technique. It describes how

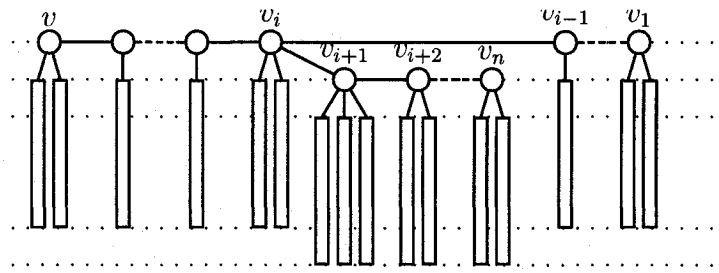


Figure 4.1: A short $(2\text{pw}(T)-1)$-layer planar drawing T in which vertex v in T is exposed.

we can force the roots of certain trees to be "squeezed" outward onto an increasingly smaller number of top or bottom layers as we decrease the number of layers that the drawing of the tree occupies.

Lemma 4.10 *Let $h \geq 1$ and T be a tree rooted at a vertex r with $n \geq 0$ children. Suppose that each subtree rooted at a child c of r has the property that every short layered drawing occupies at least $h-1$ layers and at least h layers if c is exposed. If $n \geq (i+3)(i+2)+1$ for some $0 \leq i < h$ then, in any short $(h+i)$-layer planar drawing of T, r is on one of the top or bottom i layers.*

Proof: Assume by way of contradiction that r does not lie on the top or bottom i layers; that is, r lies on layer j, $i+1 \leq j \leq h$. If the drawing of a subtree rooted at a child c of r occupies exactly $l = h-1$ layers then c is not exposed in that drawing so r lies on one of those l layers. If instead the subtree occupies $l \geq h$ layers then these l layers include layers $i+1$, $i+2$, ..., h, one which is occupied by r. By Lemma 4.1, the drawings of at most two subtrees occupy the same set of layers. There are $h+i-l+1$ ways to choose $l \geq h-1$ consecutive layers from $h+i$ total layers so r can have at most $2\Sigma_{l=h-1}^{h+i} h+i-l+1 = 2[(i+2)+(i+1)+\ldots+1] = (i+2)(i+3)$ children. □

Using Lemma 4.10, we describe the set of trees recursively. For $k = 1$, $S^k = S^1$ is the complete ternary tree of height one. For $k \geq 2$, S^k consists of a root v with one child x that in turn has two children u and w. In addition, we make use of Lemma 4.10 when $i = 1$ by giving u, w and x each $n = (i+3)(i+2)+1 = 13$ children. Each child is a root of a subtree isomorphic to S^{k-1}. See Figure 4.2.

Lemma 4.11 *For $k \geq 1$, every short planar layered drawing of S^k occupies at least $2k-1$ layers and at least $2k$ layers if its root v is exposed.*

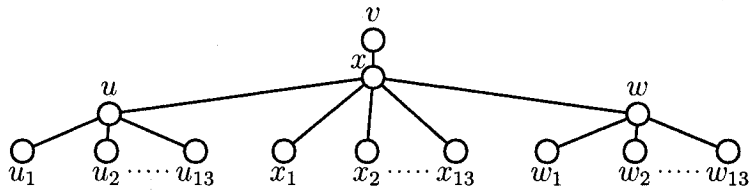


Figure 4.2: Tree S^k for $k \geq 2$. Each u_i, w_i and x_i is the root of a subtree isomorphic to S^{k-1}.

Proof: The proof is by induction on k. For $k = 1$, $S^k = S^1$ is a complete ternary tree of depth 1 so by Corollary 4.2 every drawing occupies at least $2k = 2$ layers.

Assume that $k \geq 2$. By induction, every short layered drawing of S^{k-1} occupies at least $2k - 3$ layers and $2k - 2$ layers if its root is exposed. Thus, Lemma 4.10 applies to rooted subtrees S^k_u, S^k_w and $(S^k_x \setminus S^k_u) \setminus S^k_w$ In Lemma 4.10, we let root $r = u, w$, or x and $i = 1$, so $h+i = h+1 = 2k-1$; thus, any drawing of these subtrees occupies at least $h+1 = 2k-1$ layers, and if exactly $2k - 1$ layers, then the root $r = u, w$, or x is exposed.

Now consider a drawing of S^k. Since the subtree rooted at u occupies at least $2k - 1$ layers, then every drawing of S^k also occupies at least $2k - 1$ layers. If v is exposed in a $(2k - 1)$-layer planar drawing of S^k then either u, w or x is not exposed in the drawing of its corresponding subtree. However, in that case, the corresponding subtree occupies $2k$ layers; therefore, the drawing of S^k occupies at least $2k$ layers. □

We obtain an upper bound on the pathwidth of each S^k.

Lemma 4.12 *For $k \geq 1$, $\mathrm{pw}(S^k) \leq k$.*

Proof: The proof is by induction on k. For $k = 1$, the pathwidth of $S^k = S^1$ is 1. For $k \geq 2$, there are, by induction, path decompositions for each $S^k_{u_i}$, $S^k_{w_i}$ and $S^k_{x_i}$ of width $k - 1$. From these, we construct a path decomposition of width k for S^k as follows:

- The first bags are those from the decompositions of each $S^k_{u_i}$ but with u added to each.
- The next bag consists of x and u.
- The next bags are those from the decompositions of each $S^k_{x_i}$ but with x added to each.
- The next bag consists of x and v.
- The next bag consists x and w.

- The final bags are those from the decompositions of each $S^k_{w_i}$ but with w added to each.

□

Thus, by Lemmas 4.11 and 4.12, the upper bound given in Lemma 4.9 is optimal for each S^k.

Corollary 4.13 *For each $h \geq 2$, there exists a tree T with $\mathrm{pw}(T) \leq h$ for which every short planar layered drawing occupies at least $2h - 1$ layers.*

The lower bound for short layered drawings is implied by a result of Felsner *et al.* [38]. They prove that if a planar graph G has a planar drawing on an $a \times b$ grid, then $a, b \geq \mathrm{pw}(G)$. This implies that a short layered drawing of G also occupies at least $\mathrm{pw}(G)$ layers. We reproduce our own proof of this fact here because most of it can be reused later to prove a similar lower bound for proper layer drawings.

Our proof involves constructing a path decomposition of G with width h from a short h-layer planar drawing of G. The first bag in the decomposition contains the left-most vertices on each layer in the drawing, and we show that we can construct each successive bag by removing one vertex v from the current bag and adding a new vertex immediately to the right of v in the drawing.

To do this, we require a few definitions and preliminary results. Given a vertex v in a short h-layer planar drawing of G, we use $\mathrm{R}(v)$ to denote the set of vertices on the same layer but to the right of x:

$$\mathrm{R}(v) = \{u \mid u \in V(G),\ \mathrm{Y}(v) = \mathrm{Y}(u),\ \mathrm{X}(v) < \mathrm{X}(u)\}.$$

We use $\mathrm{next}(v)$ to denote the vertex in $\mathrm{R}(v)$ closest to v; that is, $\mathrm{next}(v)$ is the vertex in $\mathrm{R}(v)$ with the minimum x-coordinate. If $\mathrm{R}(v)$ is empty then $\mathrm{next}(v)$ is undefined. Given a set of vertices $S \subseteq V(G)$ with exactly one vertex on each layer, we use $\mathrm{R}(S)$ to denote the set of vertices $\bigcup_{v \in S} \mathrm{R}(v)$. Finally, we use $\mathrm{F}(S)$ to denote the set of *frontier* vertices in S; that is, the vertices $v \in S$ such that $\mathrm{R}(v) \neq \emptyset$ and v has no neighbors in $\mathrm{R}(S)$ on a different layer:

$$\mathrm{F}(S) = \{v \mid v \in S,\ \mathrm{R}(v) \neq \emptyset,\ (u, v) \in E(G) \text{ and } u \in \mathrm{R}(S) \Rightarrow u \in \mathrm{R}(v)\}.$$

When we construct our path decomposition, we use $\mathrm{F}(S)$ to determine which vertex to remove from the current bag in the decomposition and which vertex to add in order to construct the next bag in the sequence.

Lemma 4.14 *In a short h-layer planar drawing of a graph G, if $S \subseteq V(G)$ has exactly one vertex on each layer and $\mathrm{R}(S) \neq \emptyset$ then $\mathrm{F}(S) \neq \emptyset$.*

Proof: Assume that $\mathrm{R}(S) \neq \emptyset$ and let $S' \subseteq S$ be the set of vertices $v \in S$ for which $\mathrm{R}(v) \neq \emptyset$; that is, $S' = \{v \mid v \in S,\ \mathrm{R}(v) \neq \emptyset\}$. In addition, let $S'' \subseteq S'$ be the set of vertices $v \in S'$ having a neighbor in $\mathrm{R}(S)$ on layer $\mathrm{Y}(v) - 1$; that is, $S'' = \{v \mid v \in S',\ \exists (v, v') \in E(G),\ v' \in \mathrm{R}(S),\ \mathrm{Y}(v') = \mathrm{Y}(v) - 1\}$. We observe that S' is not empty because $\mathrm{R}(S)$ is not empty and consider two cases:

1. $S'' = \emptyset$. In this case, the vertex v in S' with the largest y-coordinate has no neighbors in $\mathrm{R}(S)$ on layers $\mathrm{Y}(v) - 1$ or $\mathrm{Y}(v) + 1$; therefore, v belongs to $\mathrm{F}(S)$.

2. $S'' \neq \emptyset$. Let v be the vertex in S'' with the smallest y-coordinate. Thus, v has a neighbor v' in $\mathrm{R}(w)$ for some $w \in S'$ with $\mathrm{Y}(v') = \mathrm{Y}(w) = \mathrm{Y}(v) - 1$. Vertex w does not belong to S'' because v has the smallest y-coordinate of any vertex in S''; consequently, w has no neighbors in $\mathrm{R}(S)$ on layer $\mathrm{Y}(w) - 1$. Vertex w also has no neighbor in $\mathrm{R}(S)$ on layer $\mathrm{Y}(w) + 1 = \mathrm{Y}(v)$ because such an edge would cross edge (v, v'). Therefore, w belongs to $\mathrm{F}(S)$.

□

Finally, we obtain our lower bound:

Lemma 4.15 *If a graph G has a short planar layered drawing then that drawing occupies at least $\mathrm{pw}(G)$ layers.*

Proof: We show that, given a short h-layer planar drawing of G, we can construct a path decomposition of G with width h. When $|V| \leq h$, a path decomposition consisting of a single bag containing all vertices in G is sufficient. We assume, then, that $|V| > h$.

We construct the path decomposition $B_1,\ B_2,\ \ldots,\ B_{|V|-h}$ from a sequence of sets $S_0,\ S_1,\ \ldots,\ S_{|V|-h}$. More specifically, we let each $B_i = S_i \cup S_{i-1}$. We define each S_i inductively as follows:

- S_0 is the set of h vertices with minimum x-coordinates on each layer.

- We define S_i in terms of S_{i-1}. By Lemma 4.14, there exists at least one vertex in $\mathrm{F}(S_{i-1})$ for $1 \leq i \leq |V| - h - 1$ so we let v be the vertex with the smallest y-coordinate; thus, $S_i = (S_{i-1} \setminus \{v\}) \cup \{\mathrm{next}(v)\}$.

If we let each $B_i = S_i \cup S_{i-1}$ then we claim that $B_1,\ B_2,\ \ldots,\ B_{|V|-h}$ is a path decomposition of G with width h. The first bag B_1 contains $h + 1$ vertices, and each successive bag contains exactly one vertex not found in the bags before it so $B_1 \cup B_2 \cup \ldots \cup B_{|V|-h} = V(G)$.

Now consider an edge $(u, v) \in E(G)$ such that B_i is the first bag containing u and B_k is the first bag containing v, for $i \leq k$. Thus, we have $u \in S_i$, $v \in S_k$, and $v \in \mathrm{R}(S_j)$ for each S_j, $i \leq j \leq k-1$. Consequently, u belongs to each S_j and most importantly to S_{k-1}; thus, u and v both belong to B_k.

Finally consider a vertex $v \in B_i \cap B_k$, for $i < k$. In other words, v is in S_i and S_{k-1} so in fact $v \in S_j$ for each $i \leq j \leq k-1$; thus, v also belongs to each bag B_j. □

We cannot improve on this lower bound because the nearly complete ternary tree of depth $d \geq 1$ has pathwidth d by Lemma 4.4 and a short d-layer planar drawing. To obtain such a drawing, we simply place vertices at depth i on layer i except for each leaf which we place next to its parent and on the same layer.

Lemma 4.16 *For each $h \geq 1$, there exists a graph G with $\mathrm{pw}(G) \geq h$ and a short h-layer planar drawing.*

By Lemma 4.9 and Corollary 4.13, and Lemmas 4.15 and 4.16, then, our bounds on the number of layers in short layered drawings of trees are optimal:

Theorem 4.17 *For each $h \geq 2$, the lower bound h and the upper bound $2h-1$ are optimal bounds on the number of layers in short planar layered drawings of trees with pathwidth h.*

4.3 Proper Layered Drawings

As with short layered drawings, we obtain our upper bound by constructing drawings. This bound was first proved by Dujmović *et al.* [27]:

Lemma 4.18 *Every tree T with $\mathrm{pw}(T) \geq 2$ has a proper $(3\mathrm{pw}(T)-2)$-layer planar drawing.*

Proof: We obtain a proper $(3\mathrm{pw}(T)-2)$-layer planar drawing of any tree T with $\mathrm{pw}(T) \geq 2$ by first drawing a main path P of T on the top two layers. Each component T' in $T \setminus P$ has pathwidth at most $\mathrm{pw}(T)-1$. Let v be the vertex in T' adjacent to a vertex w in P. We insert a drawing of each T' onto the $(3\mathrm{pw}(T)-4)$ layers below $w \in P$ and then draw the missing edge (v, w) as a straight line between v and w. To avoid crossings, we draw T' so that v lies on the layer immediately below w; that is, v is exposed in the drawing of T'. It remains to prove, then, that such drawings of T' exist; that is, we must prove that for any tree T with $\mathrm{pw}(T) \geq 1$ and vertex $v \in T$, there exists a proper $(3\mathrm{pw}(T)-1)$-layer planar drawing of T in which v is exposed.

Let $P = v_1 v_2 \dots v_n$ be a main path of T. The proof is by induction on $\text{pw}(T)$. For $\text{pw}(T) = 1$, there exists a proper 2-layer planar drawing of T by Lemma 4.8. Clearly v is exposed in this drawing since there are only two layers. Now suppose that $\text{pw}(T) \geq 2$, and let R be the path between v and a vertex v_i on P. We begin drawing T on $(3\text{pw}(T) - 1)$ layers by first drawing the path $Rv_iv_{i-1} \dots v_1$ on layers one and two and then the path $v_{i+1}v_{i+2} \dots v_n$ on layers two and three below edge (v_i, v_{i-1}). Each connected component T' of $(T \setminus P) \setminus R$ has pathwidth at most $\text{pw}(T) - 1$. Let v' be the vertex in T' adjacent to a vertex $w \in P \cup R$. By induction, there exists a proper $(3\text{pw}(T) - 4)$-layer planar drawing of T' in which v' is exposed. We insert this drawing onto the layers below w. The final drawing is illustrated in Figure 4.3. □

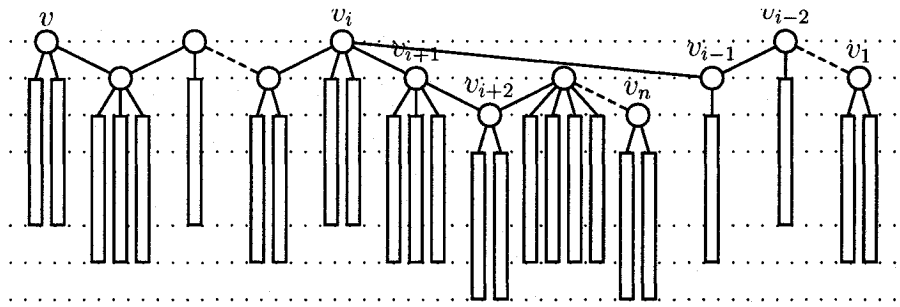


Figure 4.3: A proper $(3\text{pw}(T) - 1)$-layer planar drawing T in which vertex v in T is exposed.

As we did for short layered drawings, we show that this upper bound is optimal by using Lemma 4.10 to recursively define a set of rooted trees. We define P^1 to be the tree consisting of a single edge and, for $k \geq 2$, we define P^k to be just like S^k except that:

- we attach another child y to the root v that has two children and each of those children has exactly one child leaf;
- we make use of Lemma 4.10 when $i = 2$ by giving u, w and x each $n = (3 + i)(2 + i) + 1 = 21$ children. Each child is the root of a subtree isomorphic to P^{k-1}.

See Figure 4.4.

Lemma 4.19 *For $k \geq 1$, every proper planar layered drawing of P^k occupies at least $3k - 2$ layers and at least $3k - 1$ layers if v is exposed.*

Proof: The proof is by induction on k. For $k = 1$, tree $P^k = P^1$ consists of a single edge so every proper layered drawing of P^k occupies at least $3k - 1 = 2$ layers.

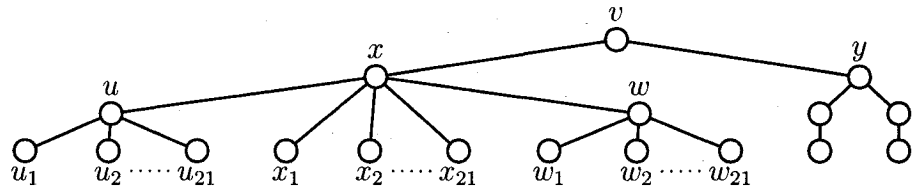


Figure 4.4: Tree P^k for $k \geq 2$. Each u_i, w_i and x_i is the root of a subtree isomorphic to P^{k-1}.

Assume that $k \geq 2$. By induction, every proper layered drawing of P^{k-1} occupies at least $3(k-1)-2 = 3k-5$ layers and $3k-4$ layers if the root is exposed. By Lemma 4.10, then, any proper layered drawing of subtree P_u^k, P_w^k or $(P_x^k \setminus P_u^k) \setminus P_w^k$ occupies at least $3k-3$ layers and if exactly $3k-3$ layers then the subtree root lies on the top or bottom layer (simply let $r = u, w$, or x and $i = 1$ so $h + i = h + 1 = 3k - 3$ in Lemma 4.10). In addition, if the drawing uses exactly $3k-2$ layers then the root lies on one of the 2 topmost or bottommost layers (simply let $r = u, w$, or x and $i = 2$ so $h + i = h + 2 = 3k - 2$ in Lemma 4.10).

Now consider a proper layered drawing of P^k. The subtree rooted at u occupies at least $3k-3$ layers so P^k also occupies at least $3k-3$ layers. Furthermore, because uxw is a path in the tree, all three cannot occupy the same layer. Consequently, if P^k occupies exactly $3k-3$ layers, then u, w or x does not lie on the top or bottom layer so it is not exposed in the drawing of its corresponding subtree. However, in that case, the corresponding subtree occupies at least $3k-2$ layers so every drawing of P^k occupies at least $3k-2$ layers.

If v is exposed on the top layer in a proper layered drawing of P^k then either u or w is on layer three. Assume without loss of generality that w is on layer three. See Figure 4.5.

We consider two cases: $k = 2$ and $k \geq 3$. If $k = 2$, then $P_w^k = P_w^2$ contains no vertices on the top layer. This is because subtrees P_u^k and P_y^k have vertices on layers 1, 2 and 3; thus, P_w^k would need a path of length 4 starting at w to reach the top layer. The longest such path has length 2. If P_w^k occupies exactly $3k-3 = 3$ layers then it occupies layers 3, 4 and 5; otherwise, it occupies at least $3k-2 = 4$ layers, layers 2 to 5. Therefore, the drawing of P^k occupies at least $3k-1 = 5$ layers.

If $k \geq 3$, then every drawing of P_w^k occupies at least $3k-3 \geq 6$ layers. If it occupies exactly $3k-3$ layers then it occupies layers 3 to $3k-1$. If it occupies exactly $3k-2$ layers then it occupies layers 2 to $3k-1$. Thus, every drawing of P^k occupies at least $3k-1$ layers. □

The pathwidth of each P^k is identical to that of S^k.

Lemma 4.20 *For $k \geq 1$, $\mathrm{pw}(P^k) \leq k$.*

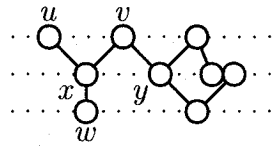


Figure 4.5: Vertex v is exposed on the top layer in a proper layered drawing of P^k.

Thus, by Lemmas 4.19 and 4.20, the upper bound given in Lemma 4.18 is optimal for each P^k.

Corollary 4.21 *For each $h \geq 2$, there exists a tree T with $\mathrm{pw}(T) \leq h$ for which every proper planar layered drawing occupies at least $3h - 2$ layers.*

For proper layered drawings, our lower bound is one layer larger than for short layered drawings because we do not permit short edges. Our proof proceeds as in Lemma 4.15 except that we let each bag $B_i = S_i$. This is possible because we do not permit short edges. Thus, we have the following lower bound for proper drawings of graphs.

Lemma 4.22 *If a graph G has a proper planar layered drawing then that drawing occupies at least $\mathrm{pw}(G) + 1$ layers.*

We cannot improve this lower bound because the complete ternary tree of depth d has pathwidth d by Lemma 4.4 and a proper $d+1$-layer planar drawing. We simply place each vertex at depth i on layer $i + 1$.

Lemma 4.23 *For each $h \geq 0$, there exists a graph G with $\mathrm{pw}(G) \geq h$ and a proper $(h+1)$-layer planar drawing.*

By Lemma 4.18 and Corollary 4.21, and Lemmas 4.22 and 4.23, then, our bounds on the number of layers in proper layered drawings of trees are optimal:

Theorem 4.24 *For each $h \geq 2$, the lower bound h and the upper bound $3h - 2$ are optimal bounds on the number of layers used in proper planar layered drawings of trees with pathwidth h.*

4.4 Upright and Unconstrained Layered Drawings

In this section we first state and prove an upper bound for upright layered drawings and then prove that this bound is the optimal upper bound for unconstrained layered drawings. Because every upright layered drawing is by definition an unconstrained layered drawing,

we will have thus proven that our upper bound is optimal for both upright and unconstrained layered drawings.

First we prove the upper bound:

Lemma 4.25 *Every tree T with $\mathrm{pw}(T) \geq 1$ has an upright $\lceil 3\mathrm{pw}(T)/2 \rceil$-layer planar drawing.*

Proof: We obtain an upright $\lceil 3\mathrm{pw}(T)/2 \rceil$-layer planar drawing of any tree T with $\mathrm{pw}(T) \geq 1$ by first drawing a main path P of T so that its vertices alternate between the top and bottom layers. We then insert drawings of each component T' in $T \setminus P$ next to the drawing of P. More specifically, let v be the vertex in T' adjacent to a vertex w in P. We insert a drawing of T' with v exposed into the $\lceil 3\mathrm{pw}(T)/2 \rceil - 1$ layers below w (or above if w is on the bottom layer). It remains to prove, then, that such a drawing of T' with $\mathrm{pw}(T') = \mathrm{pw}(T) - 1$ exists. In other words, we must show that, for any tree T with $\mathrm{pw}(T) \geq 0$ and vertex $v \in T$, there exists an upright $(\lceil 3(\mathrm{pw}(T) + 1)/2 \rceil - 1)$-layer planar drawing in which v is exposed. We note that $\lceil 3(\mathrm{pw}(T) + 1)/2 \rceil - 1$ simplifies to $\lceil (3\mathrm{pw}(T) + 1)/2 \rceil$.

The proof is by induction on the pathwidth of T. If $\mathrm{pw}(T) = 0$, then T contains no edges and can be drawn on a single layer. If $\mathrm{pw}(T) = 1$, then by Lemma 4.8, T has an upright 2-layer planar drawing.

Assume that $\mathrm{pw}(T) \geq 2$. We begin by drawing $P = v_1 v_2 \ldots v_m$, a main path of T, on $\lceil (3\mathrm{pw}(T)+1)/2 \rceil$ layers so that its vertices alternate between the top and bottom layers. If v lies on P then v will be exposed in the final drawing. Otherwise, v belongs to a component T' of $T \setminus P$. Let P' be a main path of T', v' the vertex in T' adjacent to a vertex v_i in P and let $Q = w_1 w_2 \ldots w_n$ be a path from v' to an end-vertex of P' such that Q contains as much of the path from v to v' as is possible. For example, see Figure 4.6.

We continue by drawing Q on $\lceil (3\mathrm{pw}(T)+1)/2 \rceil - 1$ layers next to v_i so that its vertices alternate between the top and bottom of these layers. If v belongs to Q then we draw Q so that v is exposed in the final drawing. Otherwise, v belongs to a component T'' of $T' \setminus P'$. Let v'' be the vertex in T'' adjacent to a vertex w_j of Q and R the path from v'' to v. In this case we draw Q so that w_j is not exposed so that we can expose v. We then draw path R on $\lceil (3\mathrm{pw}(T) + 1)/2 \rceil - 2$ layers next to w_j so that its vertices alternate between the top and bottom of these layers and v is exposed. If Q does not contain all of P' and vertex w_k in Q is adjacent to a vertex in $P' \setminus Q$ then we draw $P' \setminus Q$ on the $\lceil (3\mathrm{pw}(T) + 1)/2 \rceil - 2$ layers next to w_k so that its vertices alternate between the top and bottom of these layers. Figure 4.6 illustrates the relationships between paths P, P', Q and R in T.

Each component C of $(T \setminus T') \setminus P$ has pathwidth at most $\mathrm{pw}(T) - 1$ so, by induction, each has a $(\lceil (3\mathrm{pw}(T) + 1)/2 \rceil - 1)$-layer planar drawing in which the vertex in C adjacent

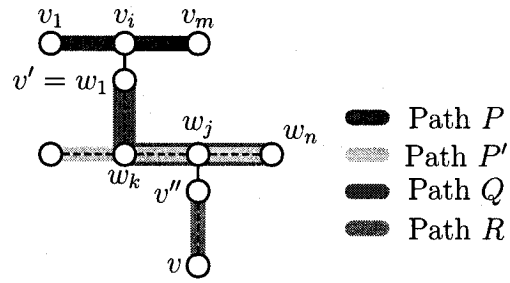


Figure 4.6: Paths P, P', Q and R in a tree T.

to a vertex in P is exposed. We recursively obtain such drawings and insert them into the main drawing next to the appropriate vertices in P. Similarly, each component of $T' \setminus P'$ and therefore each component C of $T' \setminus P' \setminus Q \setminus R$ has pathwidth at most $\text{pw}(T) - 2$, so, by induction, each has a $(\lceil (3\text{pw}(T)+1)/2 \rceil - 3)$-layer planar drawing in which the vertex in C adjacent to $P' \cup Q \cup R$ is exposed. We recursively obtain such drawings and insert them into the main drawing next to the appropriate vertices in $P' \cup Q \cup R$. The final result is illustrated in Figure 4.7. The rectangles represent drawings of components of $T \setminus P \setminus P' \setminus Q \setminus R$. □

We now prove that the upper bound just proven is optimal for unconstrained layered drawings. As for short and proper layered drawings, we prove optimality by describing a set of rooted trees whose unconstrained layered drawings use the number of layers given in the upper bound. These trees have the property that, when drawn on a minimum number of layers, the root is not accessible. A vertex v is *accessible* in a layered drawing Γ if we can insert a layered drawing of some path P into Γ without creating crossings so that one end vertex is adjacent to v and the other is exposed. Notice that if a vertex is exposed then it is accessible but the reverse is not always true. The next two lemmas show how we prevent the root from being accessible in a minimum layer drawing.

Lemma 4.26 *Let T be a tree rooted at v, and let u and w be children of v. Let Γ be an unconstrained h-layer planar drawing of T for $h \geq 1$ in which subtrees T_u and T_w each occupy at least $h - 1$ layers. Then, Γ contains a drawing of T_u in which u is accessible.*

Proof: There is a path in the drawing of T outside T_u that begins at v and ends at an exposed vertex in T_w; therefore, u is accessible in the drawing of T_u. □

Lemma 4.27 *Let u and w be vertices in a tree T rooted at vertex v such that v is the only common ancestor of u and w. Assume that u_1 and u_2 are children of u such that any unconstrained planar layered drawing of subtree T_{u_i} occupies at least $h \geq 0$ layers but u_i is not accessible in any h-layer planar drawing. Similarly, assume that w_1 and w_2 are*

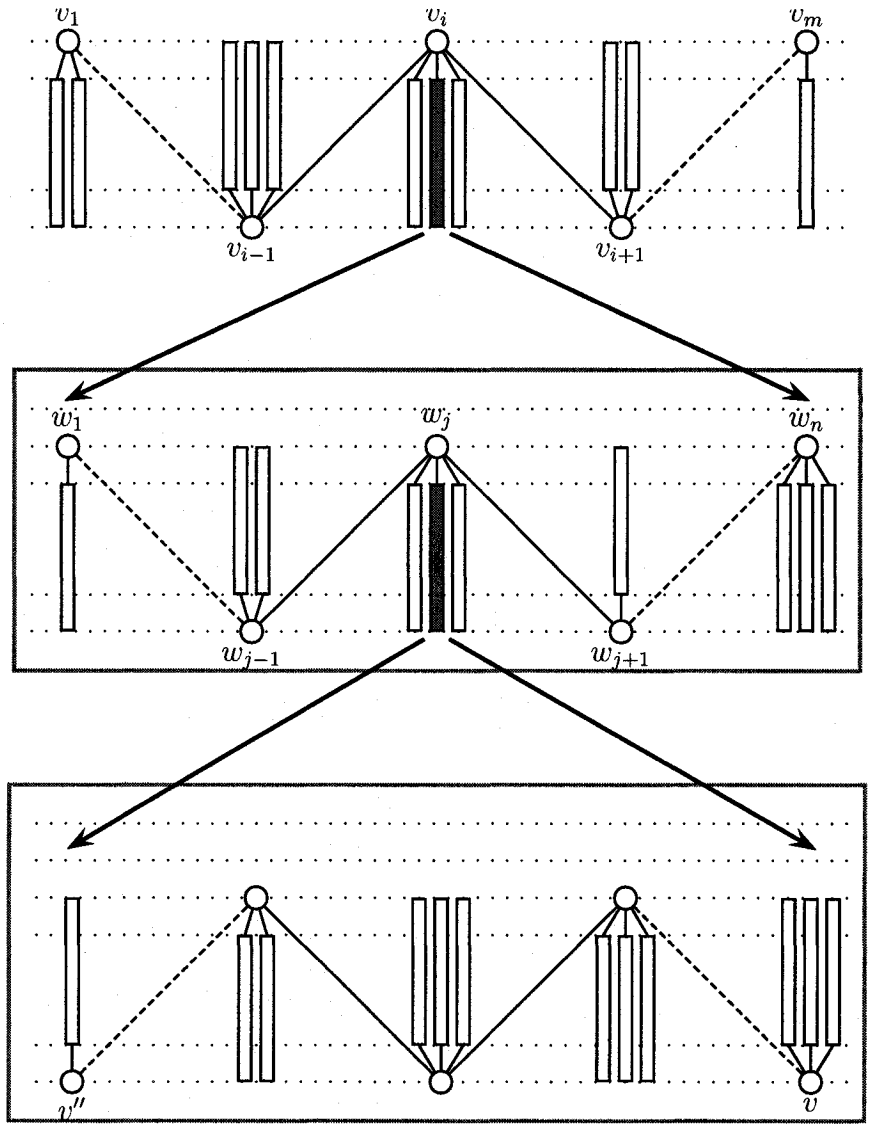


Figure 4.7: An upright $(\lceil(3\text{pw}(T)+1)/2\rceil)$-layer planar drawing of T (top), an upright $(\lceil(3\text{pw}(T)+1)/2\rceil-1)$-layer planar drawing of T' (middle) and an upright $(\lceil(3\text{pw}(T)+1)/2\rceil-2)$-layer planar drawing of T'' (bottom).

children of w such that any drawing of subtree T_{w_i} occupies at least $h+1$ layers but w_i is not accessible in any $(h+1)$-layer planar drawing. Then there are no unconstrained $(h+2)$-layer planar drawings of T in which v is accessible.

Proof: Assume by way of contradiction that we have an unconstrained $(h+2)$-layer planar drawing Γ of T in which v is accessible. We thus insert the drawing of a path $P = v_1v_2\ldots v_n$ into Γ so that v_1 is adjacent to v and v_n is on the top layer. We let $T' = T \cup P$.

We claim that, in this drawing, $T' \setminus T'_w$ occupies the top $h+1$ layers and $(T' \setminus T'_w) \setminus T'_u$ occupies the top h layers. Each subtree T'_{w_i} occupies at least $h+1$ layers so, by Lemma 4.26, w_i is accessible in the drawing of T'_{w_i}. Therefore, each T'_{w_i} occupies at least $h+2$ layers. By Lemma 4.1, then, $T' \setminus T'_w$ occupies at most $h+1$ layers. Similarly, each subtree T'_{u_i} occupies at least h of these layers so, by Lemma 4.26, u_i is accessible in the drawing of T'_{u_i}. Therefore, each T'_{u_i} occupies at least $h+1$ layers so, by Lemma 4.1, $(T' \setminus T'_w) \setminus T'_u$ occupies at most h layers. Since $v_n \in P$ lies on the top layer and $v_n \in T' \setminus T'_w$, subtree $T' \setminus T'_w$ occupies the top $h+1$ layers, that is layers $1, 2, \ldots, h+1$. Similarly, we have $v_n \in (T' \setminus T'_w) \setminus T'_u$ so $(T' \setminus T'_w) \setminus T'_u$ occupies the top h layers.

Let T'' be a subdivision of T' created by subdividing each long edge that crosses layer $h+1$ in Γ. We obtain an $(h+2)$-layer planar drawing Γ' of T'' from Γ by placing the new vertices in T'' on layer $h+1$ where the edges they subdivide intersect layer $h+1$. Let S be the non-empty set of vertices in T'' on layer $h+2$ and T''' the connected component in $T'' \setminus S$ containing u. Thus, Γ' contains an $(h+1)$-layer planar drawing Γ'' of T'''. However, this contradicts Lemma 4.1 because $T''' \setminus u$ contains three components that occupy $h+1$ layers in Γ''. The component containing u_1 occupies $h+1$ layers because we showed above that T'_{u_1} occupies the top $h+1$ layers. The same applies to the component containing u_2. The component containing v also contains $v_n \in P$ on the top layer because we showed above that P lies on the top h layers. This component also contains a vertex on layer $h+1$ adjacent to a vertex in S; therefore, the component containing v occupies all $h+1$ layers. □

Using Lemma 4.27, we recursively define a tree T^k for each $k \geq 0$. Tree T^0 is the empty tree, and tree T^1 is the single vertex. For $k \geq 2$, tree T^k consists of a root v with two children u and w. Child u has two children u_1 and u_2, each roots of subtrees isomorphic to T^{k-2}. Similarly, child w has two children w_1 and w_2, each roots of subtrees isomorphic to T^{k-1}. These trees are illustrated in Figure 4.8. As expected, we obtain the following result:

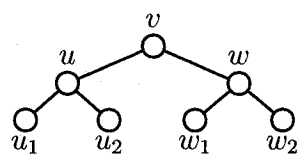


Figure 4.8: Tree T^k, for $k \geq 2$. Each u_i is the root of a subtree isomorphic to T^{k-2} and each w_i the root of a subtree isomorphic to T^{k-1}.

Lemma 4.28 *Every unconstrained planar layered drawing of T^k occupies at least $k-1$ layers and at least k layers if v is accessible.*

Proof: Every layered drawing of tree T^1 occupies at least 1 layer. Tree T^2 contains a complete ternary tree of height 1 so by Corollary 4.2 every drawing of T^2 occupies at least 2 layers. For $k \geq 3$, then, the result follows by induction and Lemmas 4.26 and 4.27. □

Next we obtain an upper bound on the pathwidth of each tree T^k.

Lemma 4.29 *For $k \geq 0$, $\text{pw}(T^k) \leq \lfloor 2k/3 \rfloor$*

Proof: The proof is by induction on k. We first prove that for $k' \geq 0$ and $k \geq 2$, if T^{k-2} contains no k'-critical vertices and we have $\text{pw}(T^{k-1}) \leq k'$ then T^k contains no $(k'+1)$-critical vertices, T^{k+1} contains no $(k'+2)$-critical vertices and $\text{pw}(T^{k+2}) \leq k'+2$.

Subtrees $T^k_{w_1}$ and $T^k_{w_2}$ are isomorphic to T^{k-1} so they each have pathwidth at most k'. Therefore, subtree T^k_w contains no $(k'+1)$-critical vertices and neither does subtree T^k_u since it is isomorphic to a subtree of T^k_w. Subtrees $T^{k+1}_{u_1}$ and $T^{k+1}_{u_2}$ are isomorphic to T^{k-2} so, by Lemma 4.5, subtree T^k_u has pathwidth at most k' and therefore v is not $(k'+1)$-critical. Thus, T^k contains no $(k'+1)$-critical vertices.

Subtree T^{k+1}_u is isomorphic to T^k_w so it contains no $(k'+1)$-critical vertices. By Lemma 4.5, T^{k+1}_u has pathwidth at most $k'+1$ so v is not $(k'+2)$-critical. Subtrees $T^{k+1}_{w_1}$ and $T^{k+1}_{w_2}$ are isomorphic to T^k so they contain no $(k'+1)$-critical vertices. By Lemma 4.5, then, subtree T^{k+1}_w has pathwidth at most $k'+1$ and therefore contains no $(k'+2)$-critical vertices. Thus, T^{k+1} contains no $(k'+2)$-critical vertices.

Subtree T^{k+2}_u is isomorphic to T^{k+1}_w so it has pathwidth at most $k'+1$. Subtrees $T^{k+2}_{w_1}$ and $T^{k+2}_{w_2}$ contain no $(k'+2)$-critical vertices because they are isomorphic to T^{k+1}. By Lemma 4.5, then, T^{k+2} has pathwidth at most $k'+2$.

The lemma then follows by induction on k because for $k = 0$ trees $T^k = T^0$ and $T^{k+1} = T^1$ both have pathwidth 0. □

Finally, we show that the upper bound of Lemma 4.25 is optimal:

Lemma 4.30 *For each $h \geq 0$, there exists a tree T with $\text{pw}(T) \leq h$ for which every unconstrained planar layered drawing occupies at least $\lceil 3h/2 \rceil$ layers.*

Proof: Consider the tree T rooted at r having three children each the roots of subtrees isomorphic to $T^{\lceil 3h/2 \rceil - 1}$. By Lemma 4.29, tree $T^{\lceil 3h/2 \rceil - 1}$ has pathwidth at most $h-1$. Therefore, by Lemma 4.5, T has pathwidth at most h.

By Lemma 4.28, in any drawing of T, the three subtrees occupy at least $\lceil 3h/2 \rceil - 2$ layers. By Lemma 4.1, then, T occupies at least $\lceil 3h/2 \rceil - 1$ layers. However, if T occupies

exactly $\lceil 3h/2 \rceil - 1$ layers then each subtree has a vertex on either the top or bottom layer implying that the root of each subtree is accessible. Therefore, by Lemma 4.28, each subtree occupies $\lceil 3h/2 \rceil - 1$ layers. Consequently, by Lemma 4.1, every layered drawing of T occupies at least $\lceil 3h/2 \rceil$ layers. □

The optimal lower bounds for upright and unconstrained layered drawings differ. We recall that Felsner *et al.* [38] prove that if a planar graph G has a planar drawing on an $a \times b$ grid, then $a, b \geq \text{pw}(G)$. This implies that an unconstrained layered drawing of G also occupies at least $\text{pw}(G)$ layers:

Lemma 4.31 ([38]) *For every graph G, any unconstrained planar layered drawing of G occupies at least $\text{pw}(G)$ layers.*

Short layered drawings are unconstrained layered drawings so, by Lemma 4.16, this lower bound is optimal.

Corollary 4.32 *For each $h \geq 1$, there exists a graph G with $\text{pw}(G) \geq h$ and an unconstrained h-layer planar drawing.*

Thus, by Lemmas 4.25, 4.30 and 4.31, and Corollary 4.32, then, our bounds on the number of layers in unconstrained layered drawings of trees are optimal:

Theorem 4.33 *For each $h \geq 1$, the lower bound h and the upper bound $\lceil 3h/2 \rceil$ are optimal bounds on the number of layers in unconstrained planar layered drawings of trees with pathwidth h.*

For upright layered drawings, we observe that in the proof of Lemma 4.31 we can simply let each $B_i = S_i$ because short edges are not permitted in the drawing.

Lemma 4.34 *If a graph G has an upright planar layered drawing then that drawing occupies at least $\text{pw}(G) + 1$ layers.*

The optimality of this bound follows from Lemma 4.23 because every proper layered drawing is by definition an upright layered drawing.

Corollary 4.35 *For each $h \geq 0$, there exists a graph G with $\text{pw}(G) \geq h$ and an upright $(h+1)$-layer planar drawing.*

Thus, by Lemmas 4.25, 4.30 and 4.34, and Corollary 4.35, then, our bounds on the number of layers in upright layered drawings of trees are optimal:

Theorem 4.36 *For each $h \geq 1$, the lower bound $h + 1$ and the upper bound $\lceil 3h/2 \rceil$ are optimal bounds on the number of layers in upright planar layered drawings of trees with pathwidth h.*

4.5 Linear-Time Drawing Algorithms

We can obtain the layered drawings described in the proofs of Lemmas 4.9, 4.18 and 4.25 in linear time. Each drawing depends on there being an algorithm that can efficiently decompose a tree into one or more paths and subtrees. More specifically, given a vertex v in a tree T, the drawings depend on three different decompositions:

1. a main path P of T and the components of $T \setminus P$;
2. a main path P of T, the path R from v to P, and the components of $(T \setminus P) \setminus R$; and,
3. a main path P of T, a main path P' of the subtree in $T \setminus P$ containing v, the path Q from P to an end vertex of P' such that Q intersects the path from v to P, the path R from v to Q, and the components of $(((T \setminus P) \setminus P') \setminus Q) \setminus R$.

Recall that the first decomposition is applied initially to the whole tree and then the last two decompositions are recursively applied to the subtrees until the entire tree is decomposed into paths. We describe an algorithm that can accomplish this in linear time.

As a preprocessing step, we root the tree at an arbitrary vertex and then apply the linear-time algorithm of Ellis *et al.* [34] for finding the pathwidth of a tree. More specifically, given a tree T the algorithm computes a label for each a vertex v in the tree. The label consists of a sequence of non-negative integers $(a_1, a_2, \ldots a_p)$ in descending order and a corresponding sequence of vertices $(v_1, v_2, \ldots v_p)$ in the subtree T_v rooted at v such that:

1. $\mathrm{pw}(T_v) = a_1$;
2. for $1 \leq i \leq p-1$, $\mathrm{pw}(T_v \setminus T_{v_1} \setminus T_{v_2} \setminus \ldots \setminus T_{v_{i-1}}) = a_{i+1}$; and
3. for $1 \leq i \leq p-1$, v_i is an a_i-critical vertex in $T_v \setminus T_{v_1} \setminus T_{v_2} \setminus \ldots \setminus T_{v_{i-1}}$.

For our algorithm we do not need to save the whole label for each vertex v. Instead, we need only retain the values a_1 and a_2 and corresponding vertices v_1 and v_2. We refer to v_1 with $\mathrm{cr}_1(v)$, v_2 with $\mathrm{cr}_2(v)$, a_1 with $\mathrm{pw}_1(v)$ and a_2 with $\mathrm{pw}_2(v)$. In the case that a_2 and v_2 do not exist, we simply say that $\mathrm{cr}_2(v)$ and $\mathrm{pw}_2(v)$ are undefined.

The first step of each decomposition is to find a main path in the tree. The next two results show how we find a main path.

Lemma 4.37 *Let v be a vertex in a rooted tree T. Then, there exist at most two vertices u and w that are descendants of v with $\mathrm{pw}_1(u) = \mathrm{pw}_1(w) = \mathrm{pw}_1(v)$, and each child c of u or w has $\mathrm{pw}_1(c) < \mathrm{pw}_1(v)$. Furthermore, the path between u and w contains $\mathrm{cr}_1(v)$ and is a main path for T_v.*

Proof: Suppose that there are two such descendants, u and w. Since $\mathrm{pw}_1(u) = \mathrm{pw}_1(w) = \mathrm{pw}_1(v)$, vertex u is not a descendant of w, and w is not a descendant of u. In addition, vertex $\mathrm{cr}_1(v)$ is the lowest common ancestor of u and w because, by Lemma 4.6, $\mathrm{cr}_1(v)$ is the unique $\mathrm{pw}_1(v)$-critical vertex in T_v.

If there is a third such descendant x of v then $\mathrm{cr}_1(v)$ is the lowest common ancestor of u, w and x, so $T_v \setminus \mathrm{cr}_1(v)$ contains three components with $\mathrm{pw}_1(v)$. By Lemma 4.4, however, this means that T_v has pathwidth greater than $\mathrm{pw}_1(v)$, a contradiction. Thus, there are at most two such descendants u and w.

Let P be the path from u to w, and consider a component T' of $T \setminus P$. Let x be the vertex on P that is adjacent to a vertex in T'. If $\mathrm{pw}(T') \geq \mathrm{pw}_1(v)$ then x is not a descendant of u or w. Consequently, there are at least three components in $T_v \setminus x$ with pathwidth at least $\mathrm{pw}_1(v)$: one containing T_u, another containing T_w, and the third containing T'. By Lemma 4.4, however, this means that T_v has pathwidth greater than $\mathrm{pw}_1(v)$, a contradiction. Thus, P is a main path. □

Corollary 4.38 *To find a main path P in a tree rooted at v, we initialize P to be the single-vertex path consisting of $\mathrm{cr}_1(v)$. We then walk down the tree following edges to vertices u with $\mathrm{pw}_1(u) = \mathrm{pw}_1(v)$. As we walk, we add each such u to the appropriate end of P.*

It is convenient if the main path removed in the first decomposition contains the root. This is because each remaining component is a rooted subtree of the original tree; consequently, we can reuse the labels calculated in the preprocessing step to recursively decompose each remaining component. If the main path does not contain r then we reroot the tree at $\mathrm{cr}_1(r)$ and relabel the tree. Then, the new root r is equal to $\mathrm{cr}_1(r)$ so the main path found using the algorithm described in Corollary 4.38 contains $r = \mathrm{cr}_1(r)$. This decomposition is illustrated in Figure 4.9(a).

We then apply either decomposition two or three to the remaining subtrees. The root of each subtree is the vertex v in the subtree that is adjacent to the main path just removed.

We apply decomposition two to subtree T_v by again using the algorithm described in Corollary 4.38 to find a main path P of T_v. The path R from the subtree root v to P is simply the path from v to $\mathrm{cr}_1(v) \in P$. We traverse this path by walking up the tree from $\mathrm{cr}_1(v)$ to v. This decomposition is illustrated in Figure 4.9(b).

We apply decomposition three to subtree T_v by again using the algorithm described in Corollary 4.38 to find a main path P of T_v. If $v = \mathrm{cr}_1(v)$, then the decomposition is complete since paths P', Q and R have zero length. On the other hand, if $v \neq \mathrm{cr}_1(v)$ then v belongs to subtree $T' = T_v \setminus T_{\mathrm{cr}_1(v)}$. We find a main path P' of T' by again applying the algorithm described in Corollary 4.38 except that, for each ancestor w of $\mathrm{cr}_1(v)$, we refer to

$\mathrm{cr}_2(w)$ and $\mathrm{pw}_2(w)$ instead of to $\mathrm{cr}_1(w)$ and $\mathrm{pw}_1(w)$, respectively. Let X be the path from $\mathrm{cr}_2(v)$ to v. Path Q is then composed of the path from $\mathrm{cr}_1(v)$ to the nearest vertex w in X and the path from an end vertex of P' to w. We traverse Q by walking up the tree from $\mathrm{cr}_1(v)$ to w and then from an end vertex of P' again up to w. Path R is the path from w to v. We traverse R by walking up the tree from w to v. Two cases for this decomposition are illustrated in Figure 4.9(c-d).

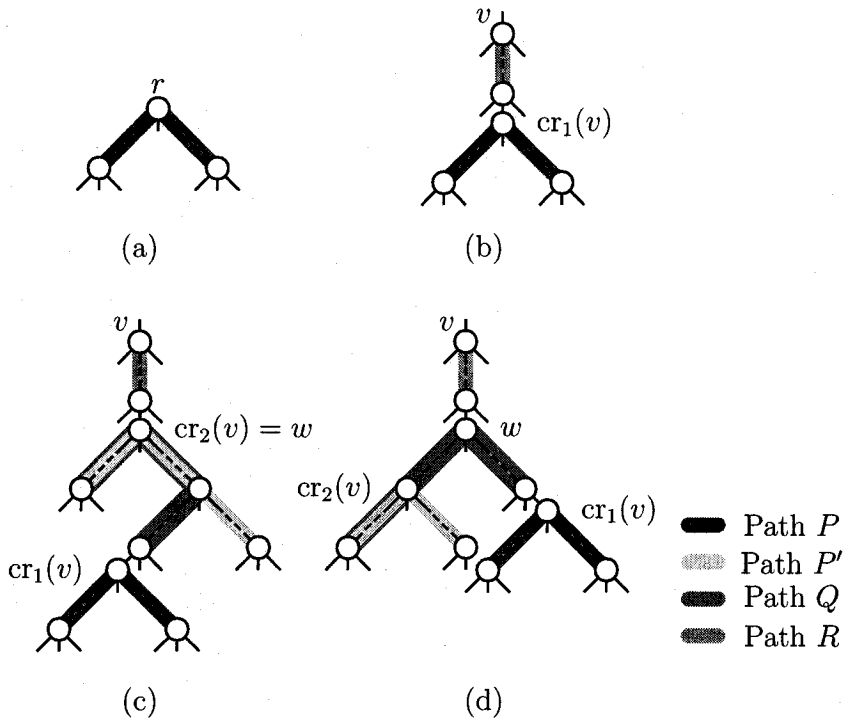


Figure 4.9: Decomposing a tree.

After applying decompositions two or three to T_v, we recursively apply either of these decompositions to any remaining components of T_v. Because the set of paths removed from T_v are connected and contain the root v, each of the remaining components is actually a rooted subtree of the original tree. In addition, the component vertex adjacent to the removed paths is precisely the root of the component. Consequently, we can reuse the vertex labels calculated earlier to recursively apply either of these decompositions exactly as we just described them for T_v.

We claim that the recursion finishes in linear time. The preprocessing step applies a linear-time labelling algorithm to the tree. The first decomposition may require rerooting and relabelling the tree, but both of these are accomplished in linear time. The remainder of the algorithm involves traversing various paths and collecting adjacent subtrees for further

decomposition. These traversals involve visiting each vertex in the tree a small constant number of times. Thus, we have linear-time algorithms to construct the drawings described in the proofs of Lemmas 4.9, 4.18 and 4.25:

Theorem 4.39 *The following drawings of a tree T with $\mathrm{pw}(T) = h$ can be obtained in linear time:*

- *an unconstrained $(\lceil 3h/2 \rceil)$-layer planar drawing (if $h \geq 1$);*
- *an upright $(\lceil 3h/2 \rceil)$-layer planar drawing (if $h \geq 1$);*
- *a short $(2h-1)$-layer planar drawing (if $h \geq 2$); and*
- *a proper $(3h-3)$-layer planar drawing (if $h \geq 2$).*

4.6 Conclusions and Open Problems

In this chapter, we have proven optimal upper and lower bounds on the number of layers required by layered drawings of trees with respect to their pathwidth. We have also given linear-time algorithms for obtaining drawings that match the upper bounds. It remains, however, an open problem to efficiently compute the minimum number of layers required to draw any specific tree. Though the algorithms presented in this chapter are optimal with respect to pathwidth, it is clear that they could be improved for individual trees.

Solving this problem for other classes of graphs similarly remains open. Outerplanar graphs are a slightly more general class than trees so our results for trees may generalize to outerplanar graphs. Biedl [8] shows that outerplanar graphs have visibility drawings that occupy $\mathcal{O}(n \log n)$ area. Though these are not layered drawings, visibility drawings do resemble layered drawings in some respects so combining our approach with that of Biedl may lead to interesting results.

Chapter 5

One-Bend Drawings

To Beppe, Emilio and Walter—*happy as we worked, where did the time go?*

A common aesthetic requirement for many graph drawings is that it be easy to locate the end-vertices of each edge. The most common way to achieve this is to draw each edge as a straight-line segment. In some applications, such a restriction is too drastic, resulting in drawings that violate some other equally important aesthetic requirement, such as area requirements or minimum distance between vertices. In this chapter, we relax this restriction slightly by allowing each edge to bend. More specifically, we consider *k-layer, 1-bend planar graph drawings*, k-layer planar drawings in which each edge is drawn as a polyline composed of at most two straight-line segments. A graph with such a drawing is said to be *k-layer, 1-bend planar*. Thus, we have the k-LAYER, 1-BEND PLANAR problem:

Given: A graph G and an integer $k \geq 1$.
Question: Is G a k-layer, 1-bend planar graph?

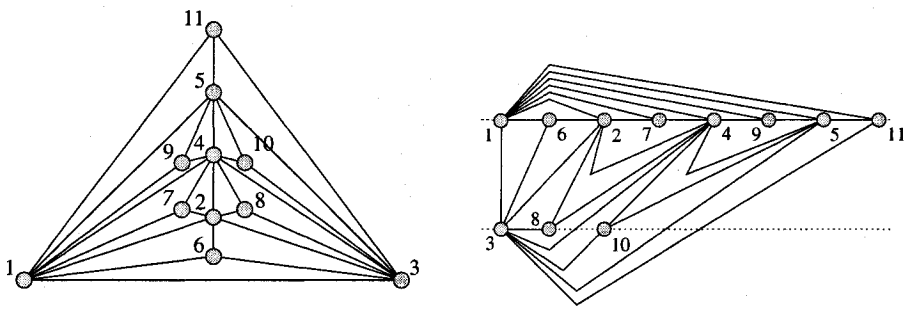
When $k = 1$, this problem turns out to be equivalent to the classic problem of testing whether or not a graph has a 2-page book embedding or 2-stack layout. Bernhart and Kainen [7] show that a planar graph has a 2-page book embedding if and only if it is sub-Hamiltonian. A planar graph is *sub-Hamiltonian* if it can be transformed into a Hamiltonian planar graph by adding edges. Wigderson [97] shows that testing maximal planar graphs for Hamiltonicity is $\mathcal{NP}$-complete, so the more general problem of testing planar graphs for sub-Hamiltonicity is at least $\mathcal{NP}$-hard.

In the process of examining a related problem, the point-set embeddability problem, Kaufmann and Wiese [61], characterize the set of 1-layer, 1-bend planar graphs as the set of sub-Hamiltonian planar graphs. Consequently, due to the results mentioned above, they show that the 1-LAYER, 1-BEND PLANAR problem is equivalent to 2-page book embeddability testing and is therefore $\mathcal{NP}$-complete.

On the other hand, Di Giacomo *et al.* [44] and Kaufmann and Wiese [61] show that, by slightly relaxing the constraints of the k-LAYER, 1-BEND PLANAR problem for $k = 1$,

we sometimes obtain polynomial problems. Di Giacomo *et al.* [44] show that all planar graphs are 1-layer, 1-bend planar when we allow the layer to be a convex curve rather than a straight line. Kaufmann and Wiese [61] prove a result that implies that all planar graphs are 1-layer, 2-bend planar. In both cases, the resulting drawings can be obtained in polynomial time.

These results lead us to ask whether or not all planar graphs are k-layer, 1-bend planar for some large enough k. In other words, we would like to know if using additional layers in 1-bend drawings will give us the power of a single curved layer or the power of using 2-bends on a single straight-line layer. Figure 5.1 suggests a positive answer, since Figure 5.1(a) shows a graph that is not 1-layer, 1-bend planar [61] but, as Figure 5.1(b) shows, the graph does have a 2-layer, 1-bend planar drawing.



(a) (b)

Figure 5.1: (a) A graph G that is not 1-layer, 1-bend planar. (b) A 2-layer, 1-bend planar drawing of G.

In spite of this positive evidence, we show in this chapter the following negative results:

- We prove that, for any given $k \geq 1$, there exists a graph that is not k-layer, 1-bend planar. In fact, we show that there is a planar 3-tree with maximum degree 12 that is not k-layer, 1-bend planar for each $k \geq 1$.
- In fact, we prove that, for each $k \geq 2$, the k-LAYER, 1-BEND PLANAR problem is $\mathcal{NP}$-hard.

Because of these negative results, we restrict ourselves to only 2 layers and obtain the following positive results:

- We generalize the equivalence between 2-page book embeddable graphs (i.e. 1-layer, 1-bend planar graphs) and sub-Hamiltonicity by introducing the notion of *sub-Hamiltonian-with-handles* graphs, and showing that it is equivalent to 2-layer, 1-bend planarity.

- We exploit the above characterization for the 2-LAYER, 1-BEND PLANAR problem to show that 2-outerplanar graphs are 2-layer, 1-bend planar and that a 2-layer, 1-bend planar drawing can be constructed efficiently. We note that there are 2-outerplanar graphs that are not 1-layer, 1-bend planar (see, e.g. Figure 5.1(a)).

The remainder of the chapter is structured as follows. In Section 5.1, we recall a technique introduced by Kaufmann and Wiese to compute 1-layer, 2-bend planar drawings because we generalize and use it extensively in the remainder of the chapter. Section 5.2 introduces the new notion of *cutting path*, a tool that we use extensively in the chapter. In Section 5.3 we describe how, for any given $k \geq 1$, it is possible to construct graphs that are not k-layer, 1-bend planar. The $\mathcal{NP}$-hardness of the k-LAYER, 1-BEND PLANAR problem is proven in Section 5.4, and in Section 5.5 we provide a characterization of the class of graphs that are 2-layer, 1-bend planar. In Section 5.6, a subclass of planar graphs that always admit a 2-layer, 1-bend planar drawing is described, and an efficient drawing algorithm is presented. Conclusions and open problems are given in Section 5.7.

5.1 Technique of Kaufmann and Wiese

In [61], Kaufmann and Wiese study the point-set embeddability problem and prove that it is closely connected with Hamiltonicity. As mentioned earlier, we generalize and use their drawing technique in the chapter, so we recall their technique and its proof of correctness below:

Lemma 5.1 (Kaufmann and Wiese, [61]) *Let S be a set of points in the plane, and let G be a planar graph with $|S|$ vertices. If G is Hamiltonian, then there exists a planar drawing of G in which each vertex is mapped to a unique point in S and each edge is drawn as a polyline with at most one bend.*

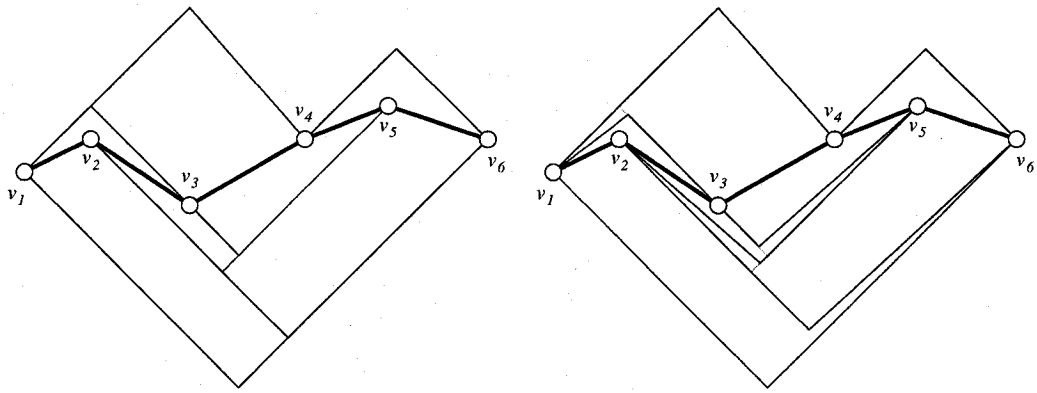
Proof: We assume that each vertical line contains at most one point in S. If this is not the case, then we rotate the points until it is the case. Let $p_1, p_2, \ldots, p_n$ be the sequence of points in S ordered by increasing x-coordinate. Let $C = v_1, v_2, \ldots, v_n$ be a Hamiltonian cycle in G, and let Ψ be a planar embedding of G such that edge (v_1, v_n) lies on the external face (notice that Ψ always exists). We describe how to compute a planar drawing of G that maps the vertices of G to the points of S and that preserves Ψ.

Assign each vertex v_i to point p_i in P and draw the edges of path $P = C \setminus \{(v_1, v_n)\}$ as straight-line segments between their end-vertices. Draw each remaining edge e using two segments, one with slope $\sigma > 0$ and the other with slope $-\sigma$. We prevent e from crossing the previously drawn edges in P by choosing our slope σ to be greater than the absolute

value of the slope of each edge in P. With segments of slope $\pm\sigma$, it is possible to draw e above or below P. In order for the drawing to preserve the planar embedding Ψ, draw e above P if e is inside C in G, and below P, otherwise.

The resulting drawing is planar except that edges outside P that are incident on the same vertex may contain overlapping segments. To eliminate overlapping, perturb overlapping edges by decreasing the absolute value of their segment slopes by slightly different amounts. The slope changes are chosen to be small enough to avoid creating edge crossings while preserving the same planar embedding. See Figure 5.2.

More formally, given an angle $\Theta > 0$, if segment s overlaps with one or more other segments and is the m^{th} longest of these segments, then we decrease the absolute value of its slope by an angle of $(m-1)\cdot\Theta$. We observe that this ensures the correct ordering of these segments around their adjacent vertex with respect to the embedding of the graph. We prevent edge crossings by choosing a small enough Θ. Namely, to prevent crossings with another edge outside P, we let Θ be small enough that our rotated s does not intersect any other previously parallel segment. An upper bound of $\frac{\epsilon}{L\cdot\Delta}$ on Θ is sufficient, where ϵ is the smallest distance between parallel segments, L is the length of the longest segment drawn so far, and Δ is the maximum degree of any vertex in G. To prevent crossings with edges in P, we further bound our choice of Θ from above so that, if we rotate a segment with slope σ by an angle of $\Theta\cdot\Delta$, then the resulting slope is still larger than the absolute value of the slope of each edge in P. □



(a) (b)

Figure 5.2: Drawing of a graph on a point-set (a) before, and (b) after perturbing overlapping segments.

5.2 Cutting Paths

Our approach to solving the k-LAYER, 1-BEND PLANAR problem is inductive on the number k of layers in the drawing. To do this, we investigate ways of partitioning k-layer, 1-bend planar graphs into $(k-1)$-layer, 1-bend planar subgraphs by removing a "special path."

We find this path by considering a k-layer, 1-bend planar drawing Γ of the graph and showing that there is a path from a vertex that is the leftmost on its layer to a vertex that is the rightmost on its layer, and that removing this path from Γ leaves behind connected components that are drawn on $k-1$ layers. More specifically, we find the path by starting at a vertex and then moving to the right of the vertex along the layer until we encounter either another vertex or an edge crossing the layer. If we encounter a vertex, then we simply continue moving to the right along the layer. On the other hand, if we encounter an edge, then we "jump" to a new layer by following the drawing of the edge to one of its end-vertices. Since the edge has at most one bend, then it is possible to follow the edge to an end-vertex without encountering the bend in the edge, if it has one. Upon arriving at the end-vertex, we continue moving to the right of the vertex along its layer as before. Below, we will show that this procedure terminates when we reach a vertex v, such that, we can move to the right of v as far as we like without encountering another vertex or edge.

By symmetry, then, we can perform the same procedure starting at v but moving to the left until we reach a vertex w, such that, like v, we can move to the left from w as far as we like without encountering another vertex of edge. Our traversal of the drawing from v to w traces a polyline through the drawing that splits the drawing into a part above the polyline and a part below the polyline. No edge connects these two parts so each part occupies at most $k-1$ layers. Finally, we show that the polyline either corresponds to a path in Γ or else it is possible to complete the path by adding edges without violating planarity. The resulting path is thus the "special path" that splits our original graph into components that are $(k-1)$-layer, 1-bend planar.

We now formalize this procedure and prove its correctness. During our traversal of the drawing, we use straight-line segments of edges to "jump" from one layer to a vertex v on another layer. We call such a segment a *jumping segment* to a vertex v. More formally, a jumping segment is a straight-line segment $\overline{pv}$ contained in an edge incident on v such that point p and vertex v lie on different layers. Also during our traversal, we move from a vertex v to a point p on an edge on the same layer. This is a horizontal line segment which we call a *landing segment* from vertex v. More formally, a landing segment from vertex v is a horizontal line segment $\overline{vp}$ from vertex v to a point p such that:

- v is to the left of point p;
- p does not coincide with a vertex but does belong to an edge;
- if an edge intersects $\overline{vp}$ at a point $q \neq p$, then q coincides with a vertex.

Thus, our traversal can be described as a alternating sequence of jumping and landing segments which we call a *jumping sequence*. More formally, a *jumping sequence* is an alternating sequence of jumping segments and landing segments $S_1, S_2, \ldots, S_p$ such that, for each pair of consecutive segments S_i and S_{i+1}, if S_i is a jumping segment $\overline{pv}$, then S_{i+1} is a landing segment $\overline{vp'}$ for some point p' right of v, and otherwise, if S_i is a landing segment $\overline{vp}$, then S_{i+1} is a jumping segment $\overline{pw}$.

We will use jumping sequences to prove the existence of our "special path" which we call a cutting path. To do this, we first show that the points shared by a layer and a jumping sequence are strictly x-monotone as we traverse the layer from left-to-right.

Lemma 5.2 *Let Γ be a k-layer, 1-bend planar drawing of a graph G, and let $S_1, S_2, \ldots, S_p$ be the subsequence of jumping segments in a jumping sequence of Γ. Then, if a segment S_i contains a point p and another segment S_j, for $i < j$, contains a point q on the same layer, then p is left of q on the layer.*

Proof: Suppose, by way of contradiction, that S_j is the first segment that violates x-monotonicity. Thus, there is a segment S_i, $1 \leq i < j$, such that S_i contains a point p on a layer L_p that is not to the left of a point q in S_j on the same layer. Assume that S_i is the last such segment before S_j for which this is true. In other words, segments $S_1, S_2, \ldots, S_{j-1}$ are x-monotone, and segments $S_{i+1}, S_{i+2}, \ldots, S_j$ are also x-monotone.

We first show that $i + 1 \neq j$. By definition, S_i and S_{i+1} do not intersect because they belong to different edge segments in a planar drawing, and they do not contain the same vertices. Also by definition, S_i contains a vertex v strictly to the left of an end-point of S_{i+1}; therefore, in each case where S_i and S_{i+1} cross the same layer, S_i crosses strictly to the left of S_{i+1}.

Next, we show that S_i and S_j do not intersect. So, if S_i and S_j contain the same vertex v, then, by definition, S_{i+1} contains a point strictly to the right of v on the layer of v. However, this contradicts our assumption that segments $S_{i+1}, S_{i+2}, \ldots, S_j$ are x-monotone. Thus, S_i and S_j do not contain the same vertex so, if they intersect at a point r, then they belong to the same edge because S_i and S_j are different edge segments in a planar drawing. By definition, S_{j-1} contains a vertex v strictly to the left of r on the layer of r. However, this contradicts the fact that segments $S_1, S_2, \ldots, S_{j-1}$ are x-monotone. Therefore, S_i and S_j do not intersect.

In other words, if L is any layer that S_i and S_j both intersect, then S_i intersects the layer strictly to the right of S_j. Furthermore, this implies that segments $S_{i+1}, S_{i+2}, \ldots, S_{j-1}$ do not intersect layer L. None can intersect at or to the left of S_i because segments $S_1, S_2, \ldots, S_{j-1}$ are x-monotone. Neither can they intersect at or to the right of S_j because segments $S_{i+1}, S_{i+2}, \ldots, S_j$ are x-monotone.

Because each pair of consecutive segments in $S_{i+1}, S_{i+2}, \ldots, S_j$ intersect a common layer, segments $S_{i+1}, S_{i+2}, \ldots, S_j$ all lie above the layers intersected by S_i and S_j or all lie below these layers. Without loss of generality, we assume that they lie below. Let L_b the lowest layer intersected by S_i and S_j. Layer L_b is not the lowest layer intersected by S_i because S_{i+1} and S_i intersect a common layer. However, this implies that L_b is similarly not the lowest layer intersected by S_j because S_{j-1} and S_j intersect a common layer. We have a contradiction, so p must lie to the left of q on layer L_p. □

Next, we show the conditions under which a jumping sequence can be inductively extended.

Lemma 5.3 *Let Γ be a k-layer, 1-bend planar drawing of a graph, and let $S_1, S_2, \ldots, S_p$ be a jumping sequence of Γ. Let p be the end-point of S_p that is also the end-point of the jumping sequence.*

1. *If S_p is a landing segment, then there exists a jumping segment S_{p+1} such that S_1, S_2, ..., S_p, S_{p+1} is a jumping sequence.*

2. *Otherwise, S_p is a jumping segment and, either each edge intersection on the layer of p to the right of p coincides with a vertex, or else there exists a landing segment S_{p+1} such that $S_1, S_2, \ldots, S_p, S_{p+1}$ is a jumping sequence.*

Proof: Suppose that S_p is a landing segment. By definition, p does not coincide with a vertex but does belong to an edge $e = (u, v)$. We show that e contains either $\overline{pu}$ or $\overline{pv}$. If not, then e contains at least two different line-segments between u and p, and at least two different line-segments between p and v. However, in this case, e is drawn with at least three different line-segments (i.e. with at least two bends), a contradiction. Therefore, either $\overline{pu}$ or $\overline{pv}$ belongs to e so one of these is a jumping segment.

Now suppose that S_p is a jumping segment. If an edge intersects the layer of p to the right of p, and the intersection does not coincide with a vertex, then let q be the leftmost such intersection. By definition, p coincides with a vertex v so $\overline{pq} = \overline{vq}$ is a landing segment. □

Using these results, we prove the existence of a *cutting sequence*. A cutting sequence in a k-layer, 1-bend planar drawing is an infinite polyline that contains at least one end-vertex of each edge that it intersects and consists of a jumping sequence, a horizontal ray

pointing at negative infinity and a horizontal ray pointing at positive infinity. See Figure 5.3 for an example of a cutting sequence in a 2-layer, 1-bend planar drawing. Recall that non-horizontal segments are all jumping segments.

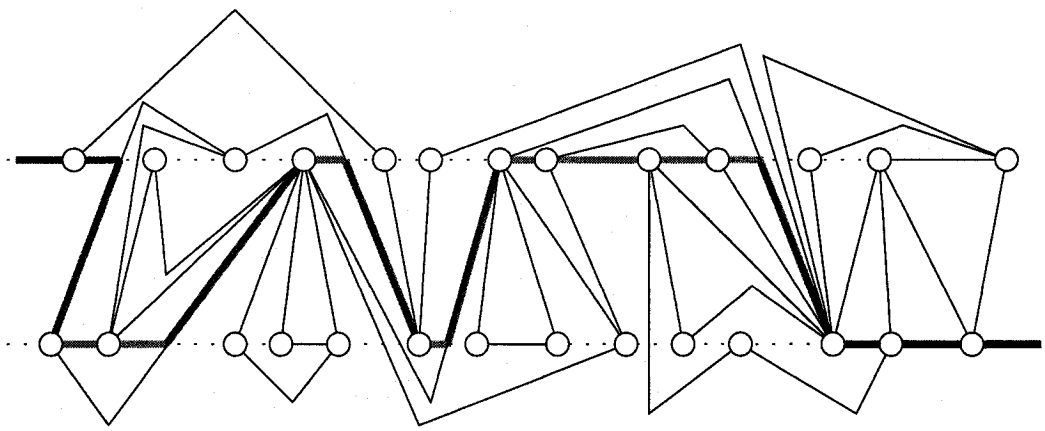


Figure 5.3: A cutting sequence in a 2-layer, 1-bend planar drawing.

Lemma 5.4 *Every k-layer, 1-bend planar drawing Γ of a graph contains a cutting sequence.*

Proof: By Lemmas 5.2 and 5.3, there exists a vertex v in Γ such that each edge intersection on the layer of v to the right of v coincides with a vertex. By symmetry, then, there exists a vertex w in Γ such that each edge intersection on the layer of w to the left of w coincides with a vertex. We observe that the horizontal ray anchored at w and pointing at negative infinity contains at least one end-vertex of each edge that it intersects.

If, in addition, each edge intersection on the layer of w to the right of w coincides with a vertex, then our cutting sequence consists of two horizontal rays anchored at w, one pointing at negative infinity and the other at positive infinity.

Otherwise, by Lemmas 5.2 and 5.3, there is a jumping sequence S starting at w and ending at a vertex w' such that each edge intersection to the right of w' coincides with a vertex. In this case, our cutting sequence consists of a horizontal ray anchored at w' and pointing at positive infinity, our jumping sequence S, and a horizontal ray anchored at w and pointing at negative infinity. □

We are nearly ready to prove that every k-layer, 1-bend planar drawing contains a *cutting path*. Formally, a cutting path is a simple path in the drawing that contains all the vertices of a cutting sequence in the same order as they appear in the cutting sequence. In some cases, the drawing does not contain all of the edges needed to complete a cutting path.

In these cases, however, it is possible to augment the drawing by adding edges so that the resulting drawing does contain a cutting path and remains k-layer, 1-bend planar. We then say that the original drawing contains an *augmenting cutting path.*

A key to this proof is the following observation about augmenting a drawing by adding edges:

Lemma 5.5 *Let Γ be a k-layer, 1-bend planar drawing of a graph G. Let v be a vertex in G, and let p be the point where an edge e intersects the layer of v. If no vertex or edge intersects the layer between v and p, then Γ can be augmented by adding at most one edge so that v is adjacent to an end-vertex of e and the resulting drawing is k-layer, 1-bend planar.*

Proof: If a vertex w coincides with p, then w is an end-vertex of e and we can augment Γ by adding segment $\overline{pv}$ without crossing any other edges.

If no vertex coincides with p, then let q be a point between v and p at an arbitrarily small distance from p. Since p is on e and e is drawn with at most two line-segments, the drawing of e contains segment $\overline{pw}$ for some end-vertex w of e. We can augment Γ by adding segments $\overline{vq}$ and $\overline{qw}$ so that v and w are adjacent. Figure 5.4 illustrates this case. The added edge is drawn as a dashed polyline.

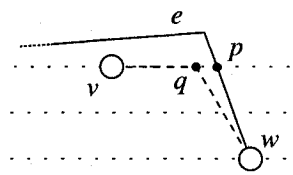


Figure 5.4: Augmenting a drawing so that v is adjacent to w.

Of course, in either of these cases, no augmentation is needed if v is already adjacent to w. □

Lemma 5.6 *Every k-layer, 1-bend planar drawing Γ of a graph contains an augmenting cutting path.*

Proof: By Lemma 5.4, Γ contains a cutting sequence. Consider two consecutive vertices u and v on the sequence that are not joined by an edge. By Lemma 5.5, we can insert a drawing of (u, v) into Γ. If u and v lie on the same layer, then either they belong to the same landing segment or else to the same horizontal ray. In either case, no edge intersects the layer between them so we can draw edge (u, v) by drawing horizontal line-segment $\overline{uv}$. If they lie on different layers, then, we assume, without loss of generality, that v coincides

with the end-vertex of a jumping segment with an end-point p on the layer of u. Since no edge intersects the layer between u and p, we can insert a drawing of edge (u, v) by Lemma 5.5.

In Figure 5.5, we illustrate the augmenting cutting path corresponding to the cutting sequence shown in Figure 5.3. The edges that we add to the drawing are indicated by dashed polylines. □

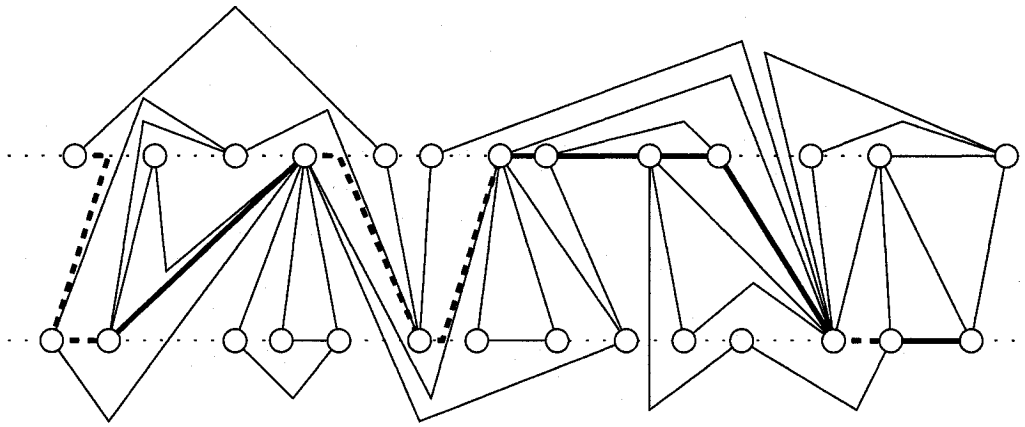


Figure 5.5: The augmenting cutting path corresponding to the cutting sequence shown in Figure 5.3.

It follows easily, that if we remove the vertices of an augmenting cutting path together with their incident edges in a k-layer, 1-bend planar drawing, then the resulting connected components are drawn on $k - 1$ layers.

Lemma 5.7 *For some $k \geq 2$, let P be an augmenting cutting path in a k-layer, 1-bend planar drawing Γ of a graph G. Then, each connected component of $G \setminus P$ in Γ is drawn on $k - 1$ layers.*

Proof: By way of contradiction, let H be a connected component of $G \setminus P$ whose drawing in Γ occupies all k layers. Thus, there exists a path P' in H from a vertex on the top layerto a vertex on the bottom layer. By definition, the cutting sequence corresponding to P lies between the top and bottom layers so the cutting sequence intersects P'. Also by definition, if an edge intersects the cutting sequence, then it contains a jumping segment, so at least one end-vertex of the edge belongs to P. However, this implies that H contains a vertex in P, a contradiction. □

5.3 Counterexamples to k-Layer, 1-Bend Planarity

In this section, we describe graphs that are not k-layer, 1-bend planar for each fixed $k \geq 1$. These graphs are maximal planar and we construct them inductively with respect to k, making extensive use of the following simple corollary of Lemma 5.7:

Corollary 5.8 *If G is a maximal planar graph that is k-layer, 1-bend planar for $k \geq 2$, then there exists a simple cycle C in G such that $G \setminus C$ is $(k-1)$-layer, 1-bend planar.*

Perhaps it seems strange that Lemma 5.7 mentions a path whereas we mention a cycle here. However, we recall that the path in Lemma 5.7 has both end-vertices on the external face. In this corollary, we are considering maximal planar graphs, so any simple path with both end-vertices on the external face can be transformed into a simple cycle by adding one external face edge to the path.

As mentioned, we construct our graphs inductively. More specifically, we construct a graph N^{k+1} for each $k \geq 1$ that is not $(k+1)$-layer, 1-bend planar from copies of a graph N^k that is not k-layer, 1-bend planar. The idea is to insert the copies of N^k in such a way that no simple cycle in N^{k+1} contains a vertex from every copy of N^k. If we can do this, then, by Corollary 5.8, N^{k+1} is not $(k+1)$-layer, 1-bend planar. The following result shows us how to do this. It is a generalization of ideas from [10] about Hamiltonicity:

Lemma 5.9 *Let G be a planar graph with subgraph H. Suppose that there exists a planar embedding of G and a corresponding embedding of H such that at least $|V(H)|+1$ faces of H contain a vertex of $G \setminus H$. If C is a simple cycle in G, then the vertices of $G \setminus H$ in one of these faces do not belong to C.*

Proof: Let $F_1, F_2, \ldots F_{|V(H)|+1}$ be faces of H that each contain at least one vertex of $G \setminus H$, and, by way of contradiction, suppose that there is a simple cycle C that contains a vertex v_i of $G \setminus H$ in each face F_i. Since G is planar, there exists a vertex of H between each pair v_i and v_j, $i \neq j$, in C. Since C is simple, this implies that C contains at at least $|V(H)|+1$ vertices of H, a contradiction. □

In other words, we need a maximal planar graph H that has at least $|V(H)|+1$ faces. We recall Euler's formula which says that, for every connected planar graph H, $|V(H)| - |E(H)| + |F(H)| = 2$, where $F(H)$ is the set of faces. Also well-known is the fact that if H is a maximal planar graph, then $|E(H)| = 3|V(H)| - 6$. Thus, Euler's formula becomes $|F(H)| = 2|V(H)| - 4$ for maximal planar graph embeddings, so $|F(H)| \geq |V(H)|+1$ for $|V(H)| \geq 5$. Let H_5, then, be the maximal planar graph on 5 vertices shown in Figure 5.6. We obtain N^1 by inserting a single vertex into each face of H_5 and then triangulating

the result. Then, we inductively construct N^{k+1} by placing a copy of N^k into each of H_5 and then triangulating the result. See Figure 5.6 for drawings of these graphs.

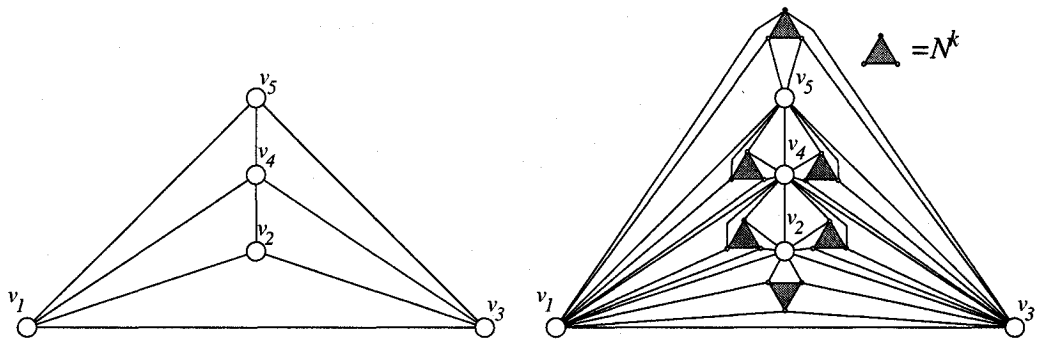


Figure 5.6: Embedded maximal planar graphs (a) H_5 and (b) N^{k+1}.

Now we show that each N^k is not k-layer, 1-bend planar. In the proof, we rely on the result of Bernhart and Kainen [7] that a maximal planar graph is 1-layer, 1-bend planar if and only if it is Hamiltonian. [1]

Lemma 5.10 *For each integer $k \geq 1$, N^k is not k-layer, 1-bend planar.*

Proof: The proof is by induction on k. By Lemma 5.9, N^1 is not Hamiltonian so N^1 is not 1-layer, 1-bend planar.

Suppose that N^k is not k-layer, 1-bend planar, and consider N^{k+1} for some $k \geq 1$. By Lemma 5.9, no cycle contains a vertex of each of the six copies of N^k in N^{k+1}. Therefore, if we assume that N^k is not k-layer, 1-bend planar, then, by Corollary 5.8, N^{k+1} is not $(k+1)$-layer, 1-bend planar. □

Thus, we have the following main result:

Theorem 5.11 *For each integer $k \geq 1$, there exists a planar graph that is not k-layer, 1-bend planar.*

We note that Theorem 5.11 has implications for drawings with 0 bends:

Corollary 5.12 *For each integer $k \geq 1$, there exists a planar graph that is not k-layer, 0-bend planar.*

[1] We note that we could have started our inductive construction at N^0 as a graph containing a single vertex, instead of with N^1. We would have then written the inductive proof of Lemma 5.10 by starting with N^0 which is trivially not 0-layer, 1-bend planar and then proceeded to the inductive step. In that case, we would not have needed the result of Bernhart and Kainen [7]. However, while this simplifies the mathematics, it is difficult to concretely describe a 0-layer, 1-bend planar drawing.

A corollary to Lemma 5.10 is that there are planar 3-trees that are not k-layer, 1-bend planar for each fixed $k \geq 1$. A 3-*tree* is any graph that is a 3-cycle or else it contains a vertex with degree equal to 3 whose removal yields a 3-tree. In other words, all planar embedded 3-trees can be constructed by starting with a planar embedded 3-cycle and then repeatedly adding a new vertex to a face of the existing graph and then triangulating. We observe that planar 3-trees are proper subclass of maximal planar graphs.

Corollary 5.13 *For each $k \geq 1$, there exists a planar 3-tree with maximum degree equal to 12 that is not k-layer, 1-bend planar.*

Proof: The proof is by induction on k, the number of layers. For each $k \geq 1$, we construct a planar embedded graph T^k just as we construct N^k, only here we give more specific instructions on how to triangulate after inserting a copy of T^{k-1} when $k \geq 2$ in order to satisfy additional properties. The additional properties include the following:

- T^k is a planar embedded 3-tree with maximum degree 12;
- T^k has at least one external face vertex with degree at most 8, another with degree at most 10 and the third with degree at most 11; and,
- T^k can be constructed by starting with a planar embedding of the 3-cycle bounding its external face, and then repeatedly adding a new vertex to an internal face and triangulating.

For $k = 1$, we let $T^1 = N^1$ since N^1 is a planar 3-tree with maximum degree equal to 8, and we can also construct T^1 as described above. Now assume the existence of T^k for some $k \geq 1$.

In constructing T^{k+1} using the method described above, our goal is to place a copy T^k into each face of H_5 and then triangulate the result. To describe the construction, we number the six faces of H_5 from 1 to 6, and their corresponding copies of T^k with the same numbers. Let face 6 be the external face of H_5. For copy of T^k number i, we let x_i, y_i, z_i denote the cycle bounding its external face. By induction, we assume that $\deg(x_i) \leq 8$, $\deg(y_i) \leq 10$ and $\deg(z_i) \leq 11$. Refer to Figure 5.7 for an illustration.

We construct T^{k+1} starting with the external face cycle v_1, v_3, x_6, where v_1 and v_3 belong to H_5 and x_6 belongs to the copy of T^k number 6. We then insert vertices v_5, v_4 and then v_2, triangulating after each insertion so that the resulting graph consists of H_5 plus vertex x_6. Next, we add the copy of T_k containing x_6. To do this, we first insert y_6 into the face bounded by v_1, x_6, v_5 and triangulate, and then we insert z_6 into the face bounded by x_6, y_6, v_5 and triangulate. Finally, we recursively insert the rest of T^k into the face x_6, y_6, z_6.

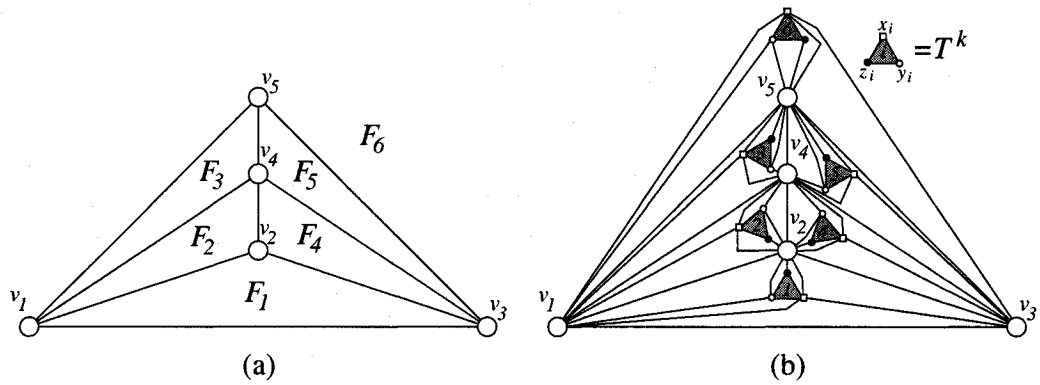


Figure 5.7: (a) The six faces of H_5 and (b) the inductive construction of T^{k+1}.

Now, consider face $i < 6$ of H_5 bounded by a cycle v_j, v_k, v_l and consider inserting the corresponding copy of T^k into that face. We first insert x_i into v_j, v_k, v_l and triangulate, then insert y_i into v_j, v_k, x_i and triangulate, and then insert z_i into v_j, x_i, y_i and triangulate. Finally, we recursively insert the rest of T^k into the face x_i, y_i, z_i. This completes the construction of T^{k+1}.

To satisfy the degree bounds in T^{k+1}, it is necessary to further refine our selection of edges used to triangulate T^{k+1}. The following table shows precisely the edges that we add to each face of H_5. These additions are illustrated in Figure 5.7.

Face	Bounding Cycle	Edges Added
F_1	v_2, v_1, v_3	$(v_2, x_1), (v_2, y_1), (v_2, z_1), (v_1, x_1), (v_1, y_1), (v_3, x_1)$
F_2	v_2, v_4, v_1	$(v_2, x_2), (v_2, y_2), (v_2, z_2), (v_4, x_2), (v_4, y_2), (v_1, x_2)$
F_3	v_5, v_4, v_1	$(v_5, x_3), (v_5, y_3), (v_5, z_3), (v_4, x_3), (v_4, y_3), (v_1, x_3)$
F_4	v_2, v_4, v_3	$(v_2, x_4), (v_2, y_4), (v_2, z_4), (v_4, x_4), (v_4, y_4), (v_3, x_4)$
F_5	v_5, v_4, v_3	$(v_5, x_5), (v_5, y_5), (v_5, z_5), (v_4, x_5), (v_4, y_5), (v_3, x_5)$
F_6	v_5, v_1, v_3	$(v_5, x_6), (v_5, y_6), (v_5, z_6), (v_1, x_6), (v_1, y_6), (v_3, x_6)$

Now, it is easy to verify that the maximum degree of each vertex in each copy of T^k is at most 12. It is also easy to verify that $\deg(v_1) = 10$, $\deg(v_2) = 12$, $\deg(v_3) = 8$, $\deg(v_4) = 12$, and $\deg(v_5) = 12$. Furthermore, of the vertices v_3, v_1 and x_6 on the external face of T^{k+1}, $\deg(v_3) = 8$, $\deg(v_1) = 10$, and $\deg(x_6) = 11$. □

5.4 Complexity of k-Layer, 1-Bend Planarity

Since we have shown that not all planar graphs are k-layer, 1-bend planar, we study the complexity of determining whether or not a planar graph is k-layer, 1-bend planar, and show

that the problem is $\mathcal{NP}$-hard. Our reduction is from the following restricted version of the HAMILTONIAN CIRCUIT problem called MAXIMAL PLANAR EXTERNAL HAMILTONIAN CIRCUIT (also MPE-HC):

> **Given:** A maximal planar graph G with a planar embedding.
> **Question:** Is G external Hamiltonian? i.e. does G contain a Hamiltonian circuit that contains an edge on the external face?

Before describing the reduction, we must first show that this problem is itself $\mathcal{NP}$-complete:

Lemma 5.14 MPE-HC *is $\mathcal{NP}$-complete.*

Proof: It is easy to verify that the problem belongs to $\mathcal{NP}$, so it remains for us to give a reduction. The reduction is from the HAMILTONIAN CIRCUIT problem when restricted to maximal planar graphs. Wigderson [97] has shown that this problem is $\mathcal{NP}$-complete.

Let G be a maximal planar graph. We obtain input $f(G)$ to the MPE-HC problem by first selecting a planar embedding of G such that a vertex v of minimum degree in G lies on the external face. Since G is maximal planar, v has degree $d = 3$, 4 or 5. Let $v_1, v_2, \ldots, v_d$ be the neighbors of v in the counter-clockwise ordering around v as defined by the embedding such that v_1 and v_d are on the external face. We obtain $f(G)$ by replacing v in G with a planar embedded copy of K_4. Let w_1, w_2 and w_3 be the vertices on the external

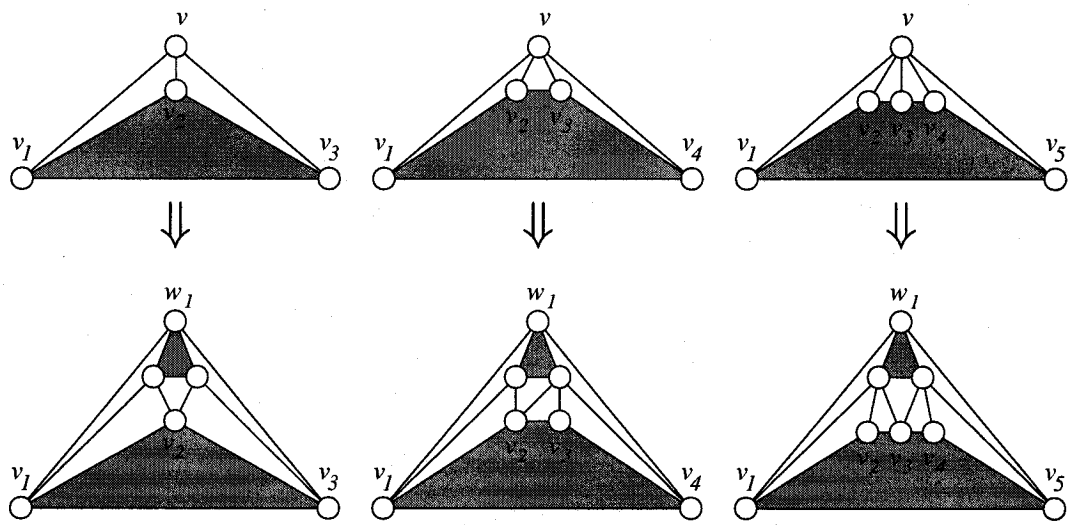


Figure 5.8: Cases for the reduction to the MPE-HC problem. The small darkened triangles in the bottom three drawings denote the graph illustrated in Figure 5.9.

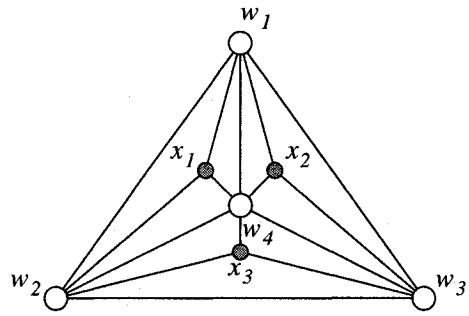


Figure 5.9: View of a planar embedded K_4 with vertices x_1, x_2 and x_3 inserted into its internal faces.

face of the inserted K_4 and let w_4 be its other vertex. We triangulate the resulting graph by inserting edges (v_1, w_1), (v_1, w_2), (v_d, w_1), (v_d, w_3), (v_2, w_2), ..., $(v_{\lceil \frac{d}{2} \rceil}, w_2)$, $(v_{\lceil \frac{d}{2} \rceil}, w_3)$, ..., (v_{d-1}, w_3) (see Figure 5.8). Finally, we insert a new vertex into each of the three internal faces of the inserted K_4 and triangulate each (see Figure 5.9). We note that the resulting graph is maximal planar and contains at least three vertices of degree 3, in particular, those vertices inserted into the faces of the inserted K_4. Thus, we set $f(G)$ equal to the resulting graph embedded so that a vertex of degree 3 lies on the external face. Since we refer to the vertices inserted into the faces of the inserted K_4, we label them x_1, x_2, and x_3 as in Figure 5.9 so that x_1 is adjacent to w_1 and w_2, x_2 is adjacent to w_1 and w_3, and x_3 is adjacent to w_2 and w_3.

It remains for us to show that G is Hamiltonian if and only if $f(G)$ is external Hamiltonian. Let C be a Hamiltonian circuit in G. In each case, we obtain a Hamiltonian circuit C' for $f(G)$ by replacing v in C with one of the following paths or its reversal: $w_1, x_1, w_4, x_2, w_3, x_3, w_2$, or $w_1, x_2, w_4, x_1, w_2, x_3, w_3$, or $w_2, x_1, w_1, x_2, w_4, x_3, w_3$. The resulting circuit is also external Hamiltonian because, in the embedding of $f(G)$, at least one external face vertex has degree 3.

Let C be an external Hamiltonian circuit in $f(G)$. We observe that vertices w_1, w_2, w_3, w_4, x_1, x_2 and x_3 are consecutive in C. This is because each vertex x_i is between some w_j and some $w_{j'}$ for each $1 \leq i \leq 3$ and some $1 \leq j \neq j' \leq 4$. In addition, the vertex before and the vertex after this subpath of C are adjacent to v in G. Thus, we obtain a Hamiltonian circuit for G by replacing this subpath in C with vertex v. □

We are now ready to describe our reduction from MPE-HC to k-LAYER, 1-BEND PLANAR. We obtain our reduction by inductively describing a maximal planar graph $H^k(G)$, for $k \geq 1$, that is k-layer, 1-bend planar if and only if the input embedded maximal planar graph G is external Hamiltonian. The construction is very similar to our construction of

N^k.

For the base of our induction, we recall the fact that a maximal planar graph is 1-layer, 1-bend planar if and only if it is Hamiltonian. We construct $H^1(G)$ by inserting G into a 3-cycle and triangulating the result.

Lemma 5.15 *Let G be an embedded maximal planar graph. Then, $H^1(G)$ is 1-layer, 1-bend planar if and only if G is external Hamiltonian.*

Proof: If $H^1(G)$ is 1-layer, 1-bend planar, then $H^1(G)$ contains a Hamiltonian circuit C. Each maximal subpath P in C consisting of vertices outside G is preceded and succeeded in C by vertices on the external face of G. Therefore, we can transform C into a Hamiltonian circuit for G by simply removing all such maximal paths and replacing them with the appropriate external face edge of G. There is at least one such maximal path P in C, so the resulting cycle contains at least one external face edge in G. Thus, G is external Hamiltonian.

Now suppose that G is external Hamiltonian, and let C be a Hamiltonian circuit in G containing an external face edge of G. We construct a Hamiltonian circuit for $H^1(G)$ by replacing the external face edge $e = (u, v)$ of G in C with the subpath u, w_1, w_2, w_3, v where w_1, w_2 and w_3 lie on the external face of $H^1(G)$, and u is adjacent to w_1 and v is adjacent to w_3. This is always possible because at least two vertices on the external face of $H^1(G)$ are each adjacent to at least two vertices on the external face of G. □

Now, given $H^k(G)$ for some $k \geq 1$, we construct $H^{k+1}(G)$ by obtaining the planar embedding for H_5 as shown in Figure 5.6(a) and planar embedding of $H^k(G)$. We then insert a copy of the embedded $H^k(G)$ into each face of H_5 and then triangulate as shown in Figure 5.10.

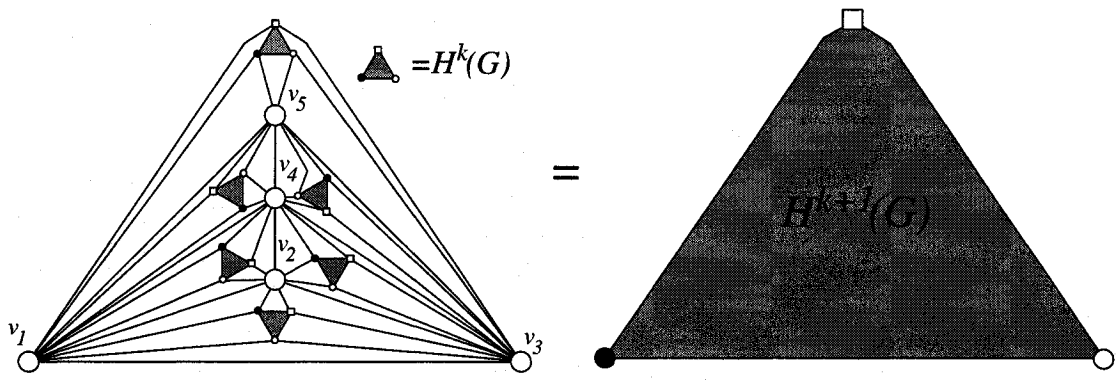


Figure 5.10: Inductive construction of $H^{k+1}(G)$.

Now we prove that if G is external Hamiltonian, then $H^k(G)$ is k-layer, 1-bend planar. In order to prove this inductively, we actually prove that $H^k(G)$ has an A-shaped k-layer, 1-bend planar drawing. In an *A-shaped k-layer, 1-bend planar drawing* of an embedded maximal planar graph, each vertex on the external face lies on the bottom layer, each edge on the external face is drawn with two straight-line segments, one with a slope $\sigma > 0$ and the other with slope $-\sigma$, and two of these edges are drawn entirely below the bottom layer. Such a drawing is illustrated in Figure 5.11.

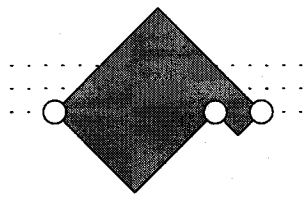


Figure 5.11: An A-shaped drawing of a graph in which the edge segments on the external face have slope ± 1.

Lemma 5.16 *Let G be a maximal planar graph that is external Hamiltonian. For each $k \geq 1$, then, $H^k(G)$ has an A-shaped k-layer, 1-bend planar drawing.*

Proof: The proof is by induction on k. For $k = 1$, Lemma 5.15 states that $H^1(G)$ has a 1-layer, 1-bend planar drawing. Since the external face contains exactly three edges and the bottom layer is the only layer in the drawing, every 1-layer, 1-bend planar drawing of $H^1(G)$ is A-shaped.

For the induction step, we assume that we have an A-shaped k-layer, 1-bend planar drawing of $H^k(G)$. We first draw $H^{k+1}(G)$ without vertex v_2 of H_5 on k layers. We duplicate the drawing of $H^k(G)$ six times and evenly space the copies so that their external face vertices lie on the bottom layer. We then draw the vertices v_1, v_3, v_4, and v_5 of H_5 on the bottom layer, v_1 left of any other vertex, v_3 between the third and fourth copies of $H^k(G)$, v_4 between the fourth and fifth copies of $H^k(G)$, and v_5 between the fifth and sixth copies of $H^k(G)$. We then draw the edges connecting the vertices of H_5 to each other and to the copies of $H^k(G)$ so that the first (leftmost) copy of $H^k(G)$ belongs to face $v_1v_2v_4$, the second to face v_1, v_2, v_3 of H_5, the third to face v_2, v_3, v_4, the fourth to face v_3, v_4, v_5, the fifth to face v_1, v_4, v_5, and the sixth to face v_1, v_3, v_5 (see Figure 5.12). Next we add the remaining edges except those incident on v_2 using the technique of Kaufmann and Wiese described in Section 5.1. No crossings are created because, according to the embedding of the graph, each edge that we draw is entirely above, entirely below or on the bottom layer.

The initial slope σ of each edge segment is any value larger than the slope of the segments on the external face of each drawing of $H^k(G)$. Thus, we have a k-layer, 1-bend planar drawing of $H^{k+1}(G) \setminus \{v_2\}$ as shown in Figure 5.13.

We obtain an A-shaped $(k+1)$-layer, 1-bend planar drawing of $H^{k+1}(G)$ by inserting a new layer above the top layer of the previous drawing and then draw v_2 on this layer immediately above the second copy of $H^k(G)$. We prove that it is possible draw the edges incident on v_2 with at most one bend per edge and without creating any edge crossings. Let t be the leftmost and u the rightmost vertex of the second copy of $H^k(G)$ on the bottom layer. Refer to Figure 5.14. We place the new layer and draw v_2 so that segment $\overline{tv_2}$ has slope σ and $\overline{uv_2}$ has slope $-\sigma$. Let q be the point where edge (v_1, t) bends, and let r be the point where edge (v_3, u) bends. We observe that v_2 is strictly above the second copy of $H^k(G)$ because the slope σ, by definition, is greater than the slopes of the segments in edge (t, u). Let s be the rightmost vertex in the first copy of $H^k(G)$ on the bottom layer.

The drawing technique of Kaufmann and Wiese guarantees that edge (s, v_4) remains above segments $\overline{tv_2}$ and $\overline{uv_2}$ even after perturbing overlapping segments because they have slopes $\pm\sigma$. Consequently, if p is the point where edge (s, v_4) bends, then segment $\overline{pv_2}$ does not intersect any edges except for (s, v_4). Thus, it is possible to draw edge (v_2, v_4).

The drawing technique also guarantees that the slope of $\overline{qt}$ is σ and the slope of $\overline{ru}$ is $-\sigma$ because no segments overlap at vertices t and u. Consequently, points q, t and v_2, and points r, u and v_2 are collinear so it is possible to draw all of the remaining edges incident on v_2 with at most one bend each and without creating any edge crossings. Figure 5.15 shows how we draw v_2 and its incident edges. □

Next, we prove the converse, that if $H^k(G)$ is k-layer, 1-bend planar, then G is external Hamiltonian.

Lemma 5.17 *Let G be an embedded maximal planar graph. For each $k \geq 1$, then, if $H^k(G)$ is k-layer, 1-bend planar, then G is external Hamiltonian.*

Proof: Our proof is by induction on k. By Lemma 5.15, we can assume that the result holds for $k \geq 1$. We prove that it holds for $k+1$.

Suppose that $H^{k+1}(G)$ is $(k+1)$-layer, 1-bend planar, but assume, by way of contradiction, that G is not external Hamiltonian. By induction, then, $H^k(G)$ is not k-layer, 1-bend planar, so, by Corollary 5.8, there exists a simple cycle C that contains at least one vertex from each copy of $H^k(G)$. However, this contradicts Lemma 5.9, so, in fact, G must be external Hamiltonian. □

By Lemmas 5.16 and 5.17, the k-LAYER, 1-BEND PLANAR problem is $\mathcal{NP}$-hard.

Theorem 5.18 *For each $k \geq 1$, the k-LAYER, 1-BEND PLANAR problem is $\mathcal{NP}$-hard.*

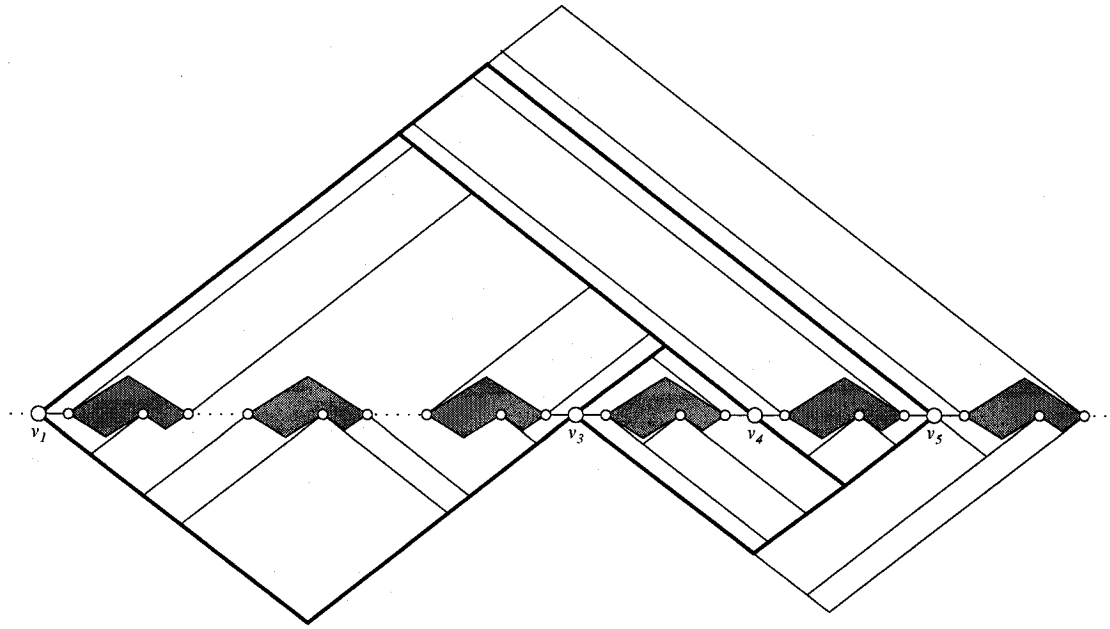


Figure 5.12: An inductive k-layer, 1-bend planar drawing of the graph $H^{k+1}(G) \setminus \{v_2\}$ with overlapping segments.

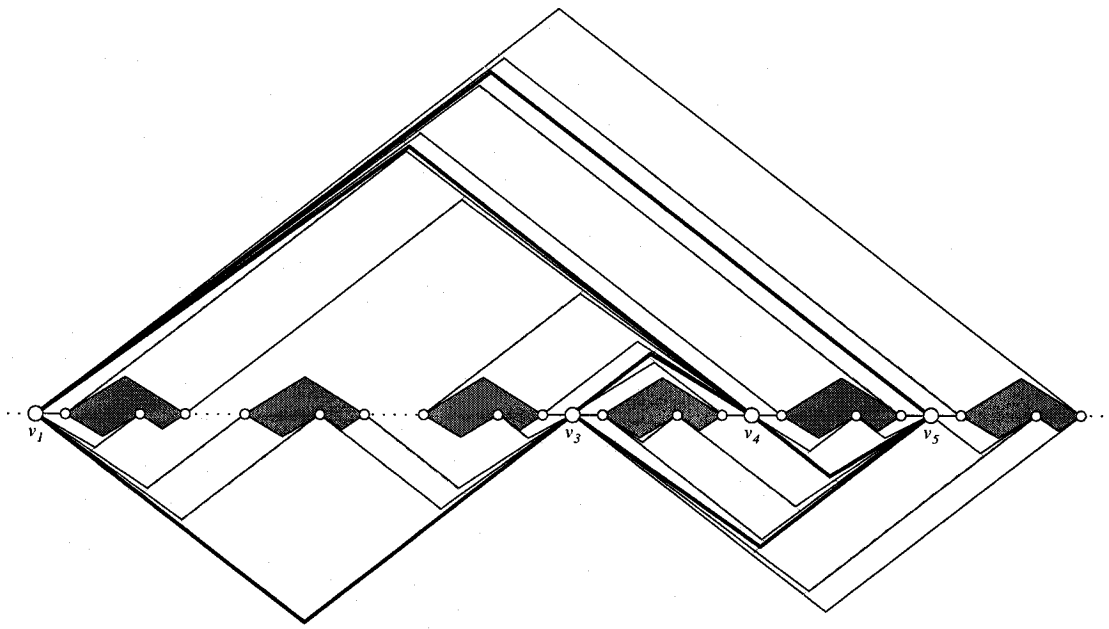


Figure 5.13: An inductive k-layer, 1-bend planar drawing of graph $H^{k+1}(G) \setminus \{v_2\}$.

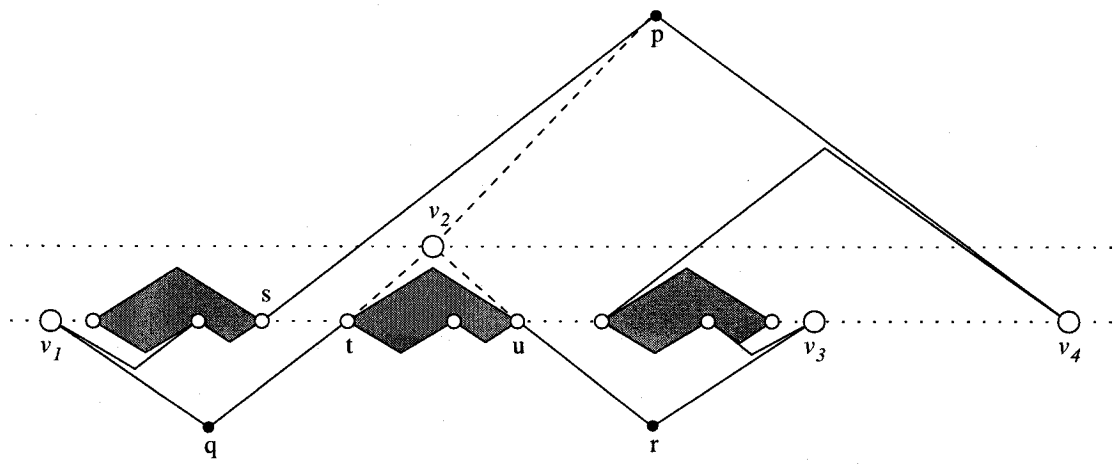


Figure 5.14: Adding v_2 to the drawing of $H^{k+1}(G) \setminus \{v_2\}$ in Figure 5.13.

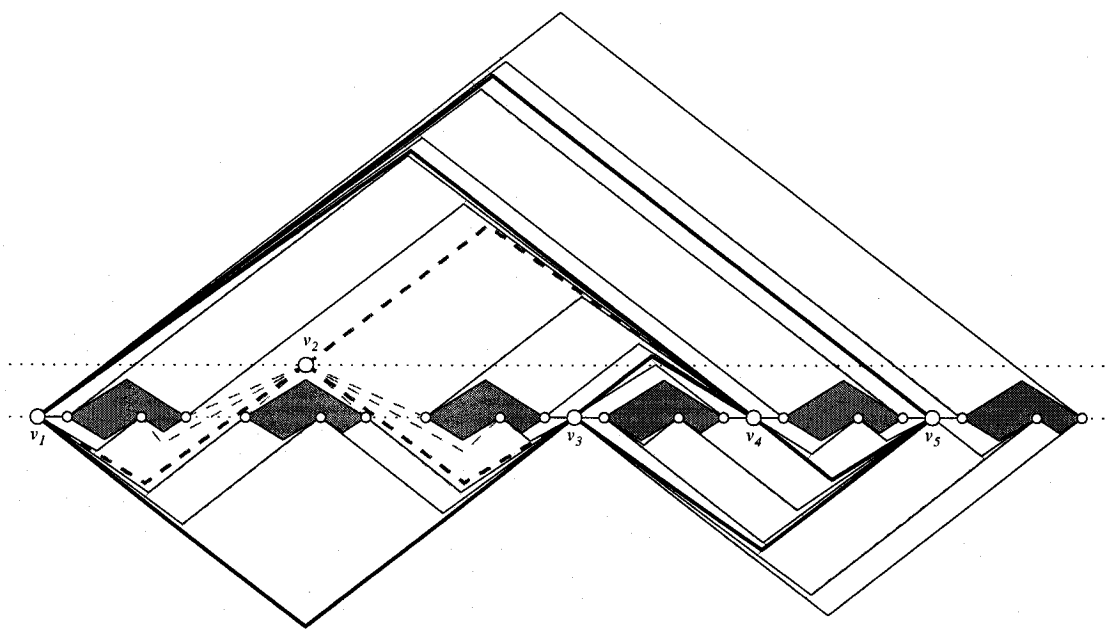


Figure 5.15: An inductive $(k+1)$-layer, 1-bend planar drawing of the graph $H^{k+1}(G)$.

5.5 Characterization of Two Layer Drawings

Given the negative results of the previous two sections, particularly the $\mathcal{NP}$-hardness of the k-LAYER, 1-BEND PLANAR problem, we consider the case where $k = 2$. Here we show that the set of 2-layer, 1-bend planar graphs is equal to the set of *sub-Hamiltonian-with-handles* graphs. Intuitively, a graph is Hamiltonian-with-handles if its vertices can be covered by a cycle and a set of vertex-disjoint paths whose end-vertices are connected to the cycle. We will define this notion in more detail shortly. We observe that our characterization is a generalization of the 1-layer, 1-bend planar characterization by Bernhart and Kainen [7] which says that the set of 1-layer, 1-bend planar graphs is equal to the set of planar sub-Hamiltonian graphs. This suggests that further generalizations may exist for drawings with three or more layers. In addition to this, our characterization is also useful for efficiently obtaining 2-layer, 1-bend planar drawings of families of graphs that are 2-layer, 1-bend planar. In Section 5.6, we give an efficient algorithm for drawing 2-outerplanar graphs.

In order to formally define the concept of Hamiltonian-with-handles graphs, we require some additional definitions. Let G be an embedded planar graph, and let Π be a simple path in G whose end-vertices lie on the external face of G. We call such a path a *base path*. A *handle* of Π consists of a simple path η (possibly consisting of a single vertex) that is vertex-disjoint with Π and, for each end-vertex of η, an edge connecting the end-vertex to Π. The vertex or vertices of the handle in Π are the *anchors* of the handle. A *dangling handle* of Π is a handle with exactly one anchor vertex. The *co-handle* of a handle is the subpath of Π between its anchors. The *handle graph* of a handle consists of the cycle composed of the handle and its co-handle, as well as any edges or vertices inside the cycle.

As we traverse Π from one end to the other, each handle is embedded on the right or left of Π. Thus, we say that two handles are embedded on the same side of Π if they are both embedded on the left or both on the right side of Π. Conversely, we say that they are embedded on opposite sides of Π if one is embedded on the left and the other on the right of Π. For example, handles η_1 and η_2 in Figure 5.16 lie on opposite sides of Π. Handle η_2 is a dangling handle. Vertices s_1 and t_1 are the anchor vertices of η_1, and vertex $s_2 = t_2$ is the anchor vertex of η_2.

We say that two handles are *overlapping* if:

1. Their handle graphs share more than one vertex; or
2. Their handle graphs share a vertex that is not an anchor for one of the handles; or
3. They are both dangling handles on opposite sides of Π that share the same anchor vertex.

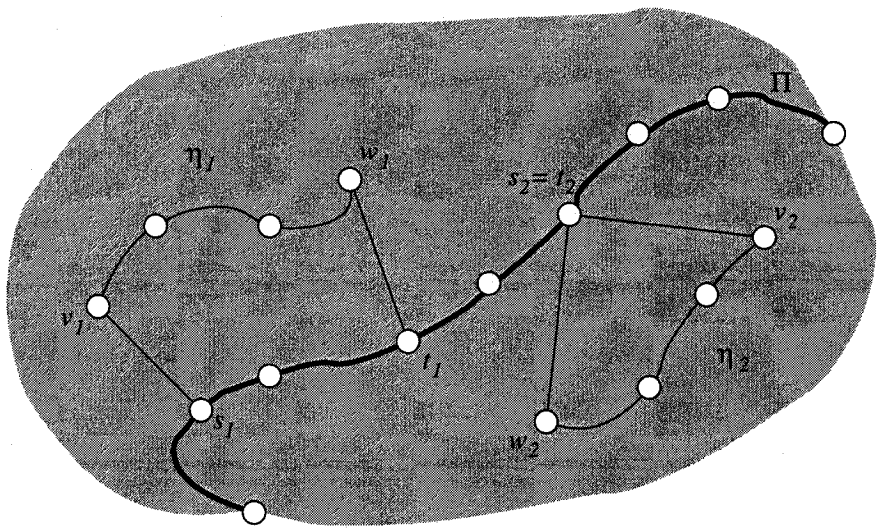


Figure 5.16: A handle and a dangling handle of Π.

We illustrate overlapping handles in Figure 5.17. Handles η_1 and η_a, η_2 and η_b, and η_3 and η_c are overlapping pairs of handles.

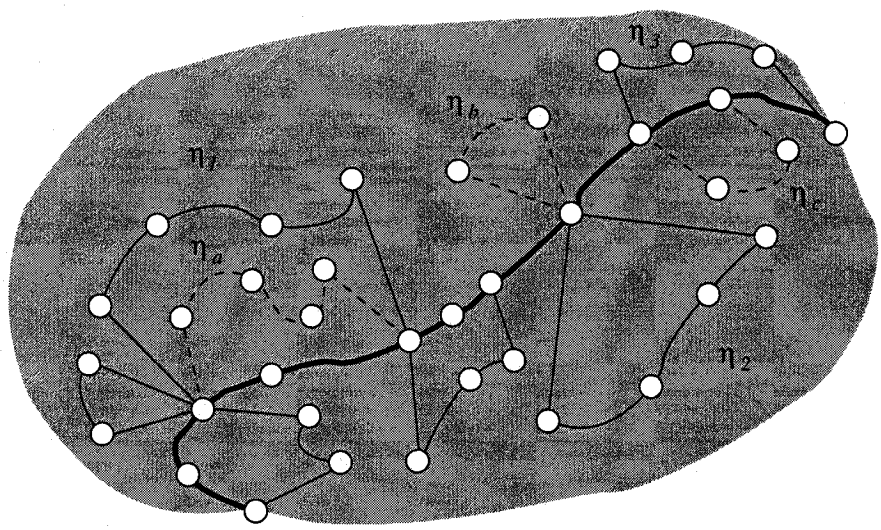


Figure 5.17: Overlapping handles.

A graph G is a *Hamiltonian-with-handles* if there exists a planar embedding of G such that vertices of G can be covered by a base path Π and a set of non-overlapping handles of Π. Thus, a graph G is *sub-Hamiltonian-with-handles* if it is possible to augment G by adding edges so that the resulting graph is still planar and Hamiltonian-with-handles.

In the remainder of this section we prove the following characterization:

Theorem 5.19 *A graph G is 2-layer, 1-bend planar iff it is sub-Hamiltonian-with-handles.*

5.5.1 Proof of Necessity

Let Γ be a 2-layer, 1-bend planar drawing of G. By Lemma 5.4, there exists a cutting sequence in Γ and, by Lemma 5.6, a corresponding augmenting cutting path Π. We will augment Γ so that we can use Π as our base path.

We select our handles in a manner very similar to the way that we obtain cutting paths. On a given layer, let $v_1, v_2, \ldots, v_q$ be a maximal sequence of consecutive vertices in the order that they appear on the layer such that each v_i is not in Π and, if an edge intersects segment $\overline{v_1 v_q}$ on the layer, then its intersection coincides with some v_i. We call $\overline{v_1 v_q}$ a *handle segment*. We observe that handle segments are very similar to landing segments defined earlier.

From each handle segment we construct a handle by finding or inserting into the drawing edges to connect the first and last vertices in the segment to Π.

First consider the case where no vertex or edge intersects the layer of v_1 to the left of v_1. In this case, v_1 lies on the external face of Γ and, by definition, Π contains a vertex v on the opposite layersuch that no vertex or edge intersects its layer to the left of v. Therefore, it is easy to draw an edge with one bend between v and v_1 entirely between the two layers without creating any edge crossings.

If this first case doesn't hold, then an edge or vertex intersects the layerto the left of v_1. By Lemma 5.5, then, it is possible to augment Γ so that v_1 is adjacent to a vertex immediately to the left of v_1 or on the opposite layer. In either case, v_1 is adjacent to a vertex w in Π. In addition, according to the proof of Lemma 5.5, the edge connecting v_1 and w lies entirely between the two layers. If w is immediately to the left of v_1, then w belongs to Π by definition, and, if w is on the opposite layer, then w belongs to Π by Lemma 5.7.

By symmetry, v_q can be connected to Π in the same manner as v_1. Thus, vertices $v_1, v_2, \ldots, v_q$ and the new edges connecting v_1 and v_q to Π form a handle of Π.

We apply this same procedure to obtain a handle for each handle segment. Thus, we are able to cover all vertices in Γ with our base path Π and its handles. Figure 5.18 illustrates how we cover the vertices with our cutting path Π and handles in the drawing of Figure 5.3. It remains to show that these handles are non-overlapping.

By definition, the vertices of the handle graph of handle η can be covered by a closed polygon. The polygon consists of one horizontal line segment on each layer and two polylines strictly between the two layers belonging to edges that connect the end-points of the

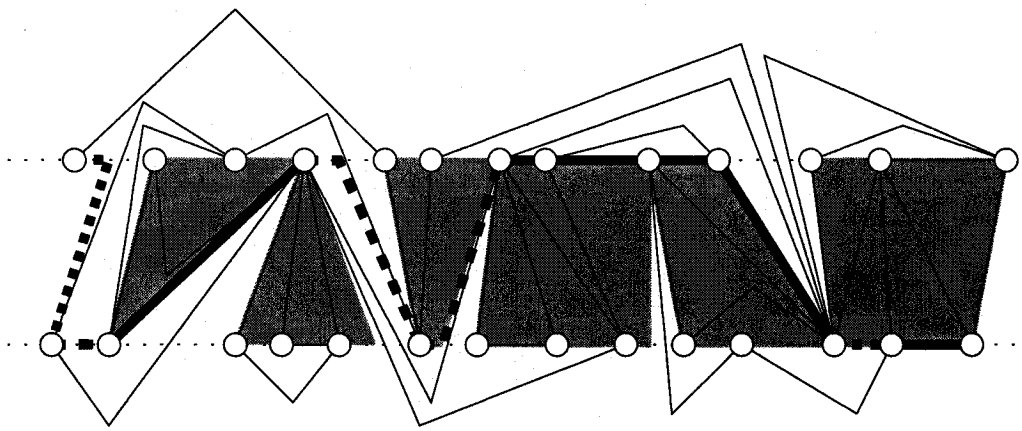


Figure 5.18: Cutting path and handles cover all of the vertices in the drawing of Figure 5.3.

horizontal line segments. One horizontal line segment L_η belongs to a landing segment in the cutting sequence, and the other horizontal line segment H_η contains the corresponding handle segment and possibly a vertex of Π at either end-point. We observe that any vertex of Π in H_η is an anchor vertex for η. We obtain the following result:

Lemma 5.20 *Let η be a handle of Γ as described above. The handle graph of η contains at least one vertex on each layer, and the leftmost vertex of the graph on one layer belongs to η while the leftmost on the other layer is an anchor vertex for η.*

Proof: Let v be the leftmost vertex in η.

We first observe that if a vertex w lies on the layer of v to the left of v, then an edge crosses the layer between them. For, otherwise, since Π is a cutting path, w cannot belong to Π because v does not belong to Π. Furthermore, w cannot belong to a handle because then v and w should belong to the same handle. By the construction of the handles, each vertex belongs to either Π or to a handle.

Thus, also by the construction of the handles, v is adjacent to an anchor vertex x for η on the opposite layer, Both v and x are leftmost in the handle graph of η. □

Using this result, we prove that the handles as constructed do not overlap, so, consider another handle η'. Let $L_{\eta'}$ and $H_{\eta'}$ be the horizontal segments containing the vertices of the handle graph of η', corresponding to segments L_η and H_η for η.

We consider two cases:

1. *H_η and $H_{\eta'}$ lie on the same layer.* Since L_η and $L_{\eta'}$ do not contain handle vertices, then, by Lemma 5.20, the leftmost vertex v of H_η belongs to η and the leftmost vertex

v' of $H_{\eta'}$ belongs to η'. Thus, H_η does not contain v' and $H_{\eta'}$ does not contain v so, in fact, H_η and $H_{\eta'}$ are disjoint segments.

Now consider L_η and $L_{\eta'}$ that both lie on the layer opposite that of H_η and $H_{\eta'}$. Since H_η and $H_{\eta'}$ are disjoint, we assume, without loss of generality, that H_η is strictly left of $H_{\eta'}$. The end-points of H_η ($H_{\eta'}$) are connected by segments belonging to edges to the end-points of L_η ($L_{\eta'}$). The drawing of Γ is planar, so, L_η and $L_{\eta'}$ share at most one vertex w, the rightmost vertex of L_η and the leftmost vertex of $L_{\eta'}$. By Lemma 5.20, w is an anchor vertex for η', so it remains for us to show that w is an anchor vertex for η. Since w belongs to both L_η and $L_{\eta'}$, one neighbor of w in Π belongs to L_η or else it lies left of H_η, and the other neighbor of w belongs to $H_{\eta'}$ or $L_{\eta'}$. In all cases, then, H_η contains no vertex of Π so w is an anchor vertex for η.

2. *H_η and $L_{\eta'}$ lie on the same layer.* Segment $L_{\eta'}$ contains only vertices of Π, and if H_η contains a vertex w of Π, then w is its rightmost vertex by Lemma 5.20. Therefore, H_η and $L_{\eta'}$ share at most one vertex in common, and if so, they share w, the rightmost vertex in H_η and the leftmost vertex in $L_{\eta'}$. We observe, that, in this case, L_η is strictly left of $H_{\eta'}$ since the leftmost vertex in $H_{\eta'}$ belongs to η' by Lemma 5.20 and the end-points of H_η ($H_{\eta'}$) are connected by segments belonging to edges to the end-points of L_η ($L_{\eta'}$). Also by Lemma 5.20, w is an anchor vertex for η' and, by the construction of η, an anchor for η as well.

Since H_η and L_η contain all vertices in the handle graph of η, and $H_{\eta'}$ and $L_{\eta'}$ contain all vertices in the handle graph of η', we have shown that the handle graphs share at most one anchor vertex in common. It remains, then, for us to show that if η and η' are dangling handles on opposite sides of Π, then they do have the same anchor vertex. However, since η and η' are on opposite sides of Π, then L_η and $L_{\eta'}$ are on opposite layers and, by Lemma 5.20, these segments contain the anchor vertices of their respective handles.

Thus, we have proven that our handles do not overlap so have proven that our characterization is a necessary condition for 2-layer, 1-bend planar drawings:

Lemma 5.21 (Necessary Condition) *If a graph G is 2-layer, 1-bend planar, then G is sub-Hamiltonian-with-handles.*

5.5.2 Proof of Sufficiency

In this section we prove the sufficiency of the characterization given in Theorem 5.19 by constructing a 2-layer, 1-bend planar drawing of our sub-Hamiltonian-with-handles graph G.

Let Π be the base path in G. Since the end-vertices of Π lie on the external face of G, Π divides G into two subgraphs, a left subgraph and a right subgraph. Intuitively, the algorithm first draws Π on two layers and then draws the left subgraph above Π and the right subgraph below Π. The outline of our algorithm is as follows:

Drawing Π: We draw Π on two layers so that the co-handle of each handle in the left subgraph lies on the bottom layer, and the co-handle of each handle in the right subgraph lies on the top layer. Section 5.5.2.1 describes this step in more detail.

Removing the Dangling Handles: In order to simplify the algorithm, we replace the dangling handles with edges. We will use the drawings of these edges in the last step to guide our re-insertion of these handles into the drawing. Section 5.5.2.2 describes this step in more detail.

Drawing the Non-Dangling Handles: We draw each handle vertex on the layeropposite to its co-handle in Π, centered above or below its co-handle end-vertices. Section 5.5.2.3 describes this step in more detail.

Drawing the Non-Handle Graph Edges: We draw the edges that do not belong either to Π or to a handle graph. Recall that Π divides our embedded graph into two subgraphs, one to the left of Π and the other to the right of Π. Consequently, we draw the non-handle graph edges of each subgraph separately, the edges of the left subgraph above Π and the edges of the right subgraph below Π. Section 5.5.2.4 describes this step in more detail.

Drawing the Handle Graph Edges: Other than drawing the dangling handles, all that remains is to draw the edges inside each non-dangling handle. Section 5.5.2.5 describes this step in more detail.

Re-inserting the Dangling Handles: Finally, we re-insert the dangling handles back into the drawing after removing the edges we inserted earlier. We use the positions of these edges to guide our drawings of the dangling handles. Section 5.5.2.6 describes this step in more detail.

We now give a detailed description the steps above. In our drawing, the layers are horizontal and 1 unit apart.

5.5.2.1 Drawing Π.

Suppose that $\Pi = v_1, v_2, \ldots, v_p$. We assign preliminary x-coordinate i to each v_i in Π. We may modify some of these coordinates later.

We assign a layer to each vertex v_i so that the following two properties are satisfied:

Property 5.1 *Let $\Pi = v_1, v_2, \ldots, v_p$ be the base path.*

1. *If v_i is the anchor vertex of a dangling handle, then v_i is assigned the top layer if the handle is on the right side of Π; otherwise, if the handle is on the left side, then v_i is assigned the bottom layer.*

2. *If v_i is not a handle anchor but belongs to a co-handle, then v_i is assigned the top layer if the handle corresponding to the co-handle is on the right side of Π; otherwise, if the handle is on the left side, then v_i is assigned the bottom layer.*

These layer assignments are possible because the handles are non-overlapping. In particular, if v_i is the anchor of a dangling handle, then all dangling handles that it anchors lie on the same side of Π. If, on the other hand, v_i belongs to a co-handle but is not an anchor for its corresponding handle, then v_i belongs to exactly one co-handle and is not the anchor for any handle.

In order to simplify our drawing, we would like to draw the edges of Π as straight-line segments. Unfortunately, our two goals mentioned above are incompatible with this simplification when the co-handle of a non-dangling handle η consists of a single edge e, and both end-vertices of e are anchors of dangling handles on the opposite side of Π. In this situation, it is impossible to draw e as a straight line since the vertices of η must lie on the same track as the end-vertices of e. We resolve this problem by observing that each handle contains at least two edges. Therefore, since our handles are non-overlapping, we can eliminate η by replacing e in Π with η. Thus, we can assume, for the remainder of our drawing procedure the following property:

Property 5.2 *Each co-handle contains at least two edges.*

Now we are able to safely draw the edges of Π as straight-line segments between their end-vertices so that the drawing satisfies Properties 5.1 and 5.2, and the following:

Property 5.3 *If $\Pi = v_1, v_2, \ldots, v_p$, then, each edge (v_i, v_{i+1}) is drawn as a straight-line segment with slope 0 or $\pm\frac{1}{2}$.*

5.5.2.2 Removing the Dangling Handles

To simplify our drawing procedure, we replace dangling handles in the graph with edges. Let η be a dangling handle of G with anchor vertex s. Then, for each edge $e = (v, w)$ where w is not in η and $v \neq s$ is in η, we replace edge e with new edge $e' = (w, s)$.

It is possible that we remove the handle η here but do not replace it with any edges. In this case, s and a vertex $v \neq s$ of η belong to a face of G external to η. If this face contains no other vertices, then G is equal to η. In this case, obtaining a 2-layer, 1-bend planar drawing of G is trivial, so we assume that there is at least one other vertex w on this face. Adding edge (v, w) to G inside this face does not violate the planar embedding of G so add this edge and the reapply the replacement described above. Consequently, η is replaced by the edge (s, w).

Another potential problem is that two different dangling handles might contain adjacent vertices. We handle this problem by removing the dangling handles in a fixed order, and, then, when we want to re-insert them back into the drawing later, we re-insert them in the reverse order. The removal technique is illustrated in Figure 5.19.

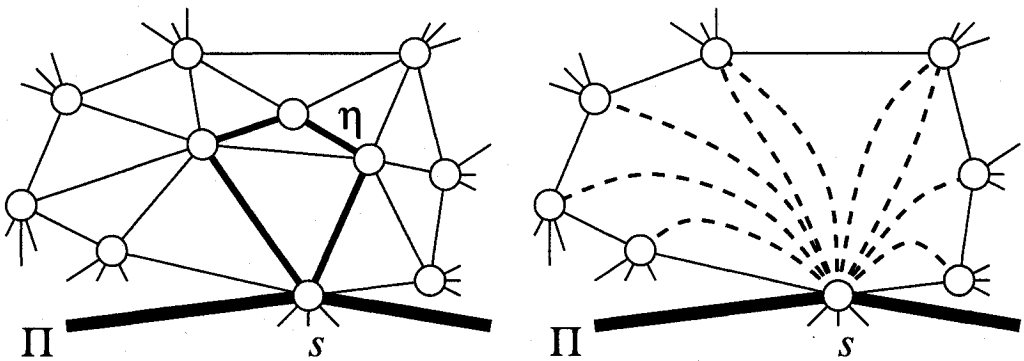


Figure 5.19: The removal of a dangling handle.

5.5.2.3 Drawing the Non-Dangling Handles

We assign a layer to each non-dangling handle vertex as follows: non-anchor vertices in handles on the left side of Π are assigned the top layer, and vertices in handles on the right side are assigned the bottom layer. We observe that this places each handle on the layer opposite its non-anchor co-handle vertices by Property 5.1.

Property 5.4 *The non-anchor vertices of each non-dangling handle on the left (right) side of* Π *in the embedding is drawn on the top (bottom) layer.*

When assigning x-coordinates, we must be careful not to make it impossible to draw edges outside the handle graph with at most one bend. Figure 5.20(a) illustrates the potential problem if we are not careful. The handle anchored at vertices s and t has been drawn

so that the handle is wider than its co-handle. Consequently, it is impossible to draw edge $e = (s, t)$ with one bend without crossing an edge in the handle graph.

Let $s = v_1, v_2, \ldots, v_p = t$ be the handle vertices in the order that they appear in the handle, and let $s = w_1, w_2, \ldots, w_q = t$ be the co-handle vertices, in the order that they appear in Π. We assume, without loss of generality, that vertices $v_2, v_3, \ldots, v_{p-1}$ are assigned the top layerand vertices $w_2, w_3, \ldots, w_{q-1}$ are assigned the bottom layer. By Property 5.2, we have that $q - 1 \geq 2$, so we avoid the problem mentioned above by shifting all of the vertices with x-coordinates smaller or equal to that of w_{q-1} in the drawing to the left. We shift them far enough so that we can assign x-coordinates to the handle vertices $v_2, v_3, \ldots, v_{p-1}$ such that $\mathrm{X}(v_i) + 1 \leq \mathrm{X}(v_{i+1})$ for each $1 \leq i \leq p - 1$. We observe that our drawing of Π still satisfies Property 5.3. See Figure 5.20(b).

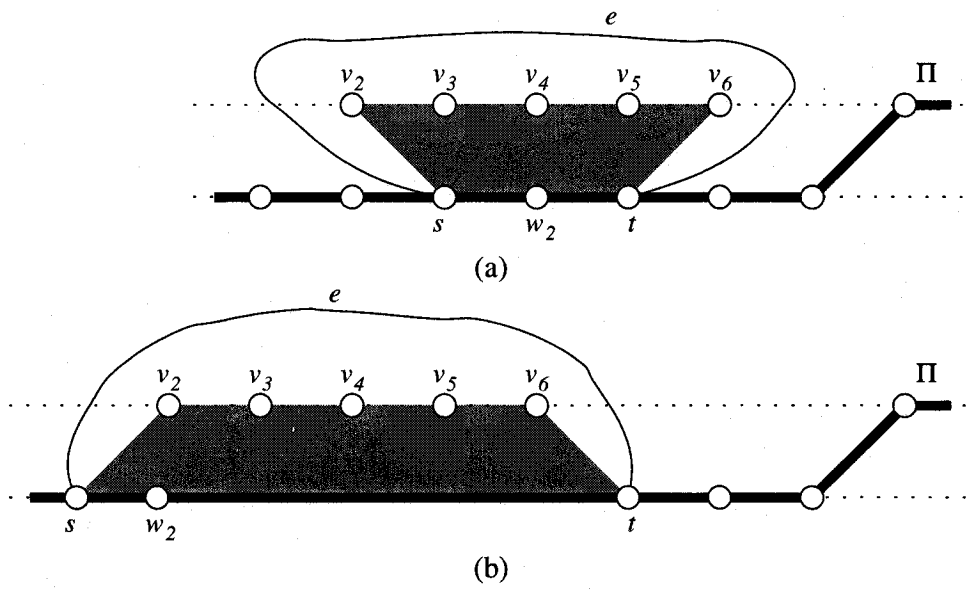


Figure 5.20: Solving the co-handle width problem.

Finally, we draw the handle edges as straight-line segments between their end-vertices. Because our handles are non-overlapping, our drawing of Π satisfies Properties 5.1 and 5.3, and because each co-handle has at least one non-anchor vertex by Property 5.2, our current drawing has the following property:

Property 5.5 *The drawing of* Π *and of the non-dangling handles is planar and, if* $P = v_1, v_2, \ldots, v_p$ *is the path composed of handles drawn above (below)* Π *along with the subpaths of* Π *connecting them together, then each edge* (v_i, v_{i+1}) *is drawn as a straight-line segment with slope* 0 *or* $\pm\frac{1}{2}$ *and* $\mathrm{X}(v_i) + 1 \leq \mathrm{X}(v_{i+1})$.

5.5.2.4 Drawing the Non-Handle Graph Edges

At this point, all vertices in the graph have been drawn so we are ready to draw the edges that do not belong to handle graphs. We draw these edges using the technique of Kaufmann and Wiese described in the proof of Lemma 5.1. Our situation here is similar in that each edge that we wish to draw must be drawn entirely above or entirely below Π by Property 5.4. It is different only in the fact that we have previously drawn not only a path Π but also handles. The handles, however, do not create any difficulties because, when drawing the edges above Π, we are actually drawing above a path P consisting of subpaths of Π and the handles drawn above Π. By Property 5.5, the x-coordinates of the vertices in P are monotonically increasing as we traverse P from one end to the other. The remainder of Π and the handles drawn below Π are drawn below P so they do not interfere with the drawings of these edges. Analogous comments apply when drawing the edges below Π.

Figure 5.21 illustrates how edges are drawn above Π, both with the overlaps and then without.

To simplify the step where we re-insert dangling handles, we would like each edge that we draw above Π and whose end-vertices are on the bottom layer to bend at a point above the top layer. To achieve this, we simply require that σ, our initial segment slope, be greater than $\frac{\Delta_y}{\Delta_x/2}$ where Δ_y is the distance between the layers and Δ_x is the minimum distance between the end-vertices of an edge. Since $\Delta_y = 1$ and, by Property 5.5, we have $\Delta_x \geq 1$, we assign σ a value greater than 2. Thus, when we are finished this step, our drawing satisfies the following property:

Property 5.6 *Each non-handle edge on the left (right) side of* Π *bends at a point above (below) the top (bottom) layer.*

5.5.2.5 Drawing the Handle Graph Edges

Next we complete the drawing of handle graphs. Consider a handle η, and let $v_1, v_2, \ldots, v_p$ be the vertices of its handle graph on the top layerand $w_1, w_2, \ldots, w_q$ be the vertices of its handle graph on the bottom layer. We assume that the order of these sequences corresponds to their left-to-right order on their respective layers. By Property 5.5, each edge (v_i, v_{i+1}) and each edge (w_i, w_{i+1}) is drawn as a horizontal line segment, and edges (v_1, w_1) and (v_p, w_q) are drawn as straight-line segments with slopes $\pm\frac{1}{2}$.

We first draw each edge (v_i, w_j) as a straight-line segment $\overline{v_i, w_j}$ as in Figure 5.22(a). We note that the resulting drawing has the same embedding as the original graph and is 2-layer, 1-bend planar.

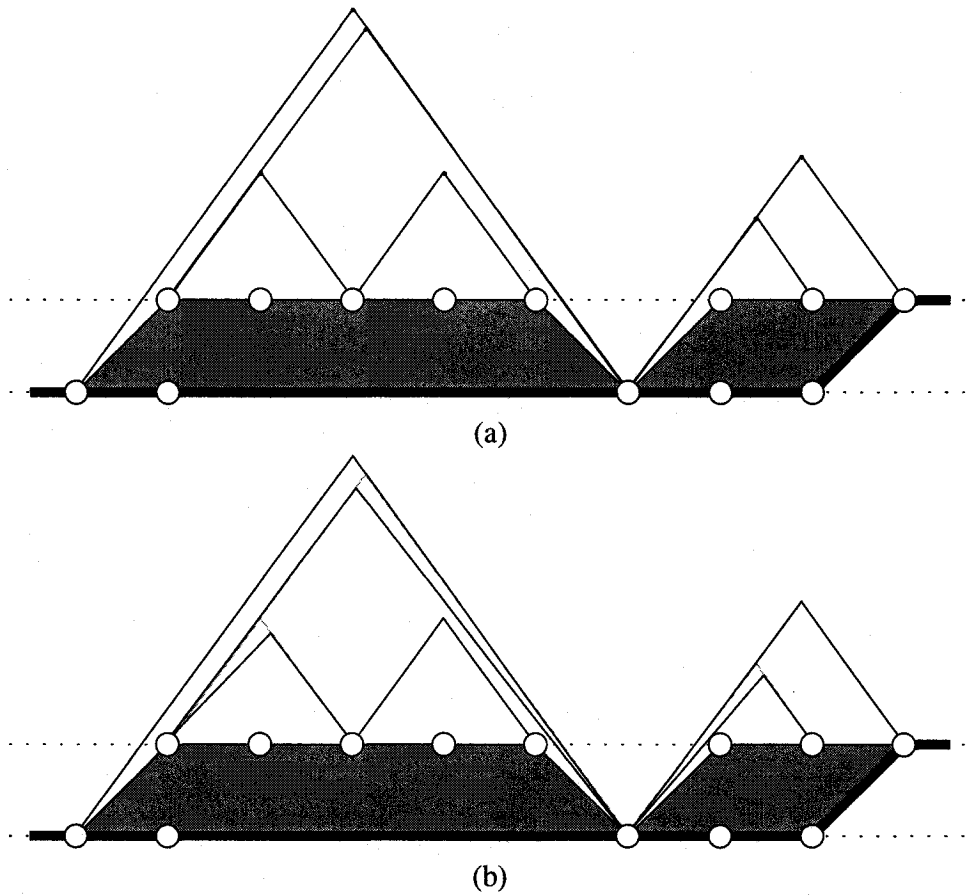


Figure 5.21: Non-handle graph edges drawn above Π with exactly two straight-line segments, (a) before and (b) after perturbation.

The remaining handle graph edges have both end-vertices on the same layer. Let T_η be the remaining edges with both end-vertices on the top layer, and let B_η be the remaining edges with both end-vertices on the bottom layer. We draw these edges using the technique of Kaufmann and Wiese described in Lemma 5.1. There are, however, differences to consider here. First of all, we must prevent the initial segment slope σ from being too large because, according to our planar embedding, the edges must be drawn entirely inside the handle graph, without crossing any of the previously drawn edges in the handle graph. More precisely, suppose that we are drawing edge $(w_i, w_j) \in B_\eta$ where $i + 1 < j$. Recall that we draw (w_i, w_j) with two segments, the segment incident on w_i with slope σ and the segment incident on w_j with slope $-\sigma$. To prevent the edge from crossing the top layer, σ must be smaller than $\frac{\Delta_y}{\frac{1}{2}(\mathrm{X}(w_j)-\mathrm{X}(w_i))}$, where Δ_y is the distance between the layers (recall that $\Delta_y = 1$). To guarantee this for each edge in B_η, we choose a σ smaller than $\frac{\Delta_y}{\frac{1}{2}(\mathrm{X}(w_q)-\mathrm{X}(w_1))}$. Of course, to additionally ensure that edges in T_η do not cross the bottom layer, we choose

a σ smaller than $\frac{\Delta_y}{\frac{1}{2}\max(\mathrm{X}(v_p)-\mathrm{X}(v_1),\mathrm{X}(w_q)-\mathrm{X}(w_1))}$. We must also prevent the edge from crossing any previously drawn edges in the handle graph, that is, those with end-vertices on different layers. If edge (v_k, w_l) has the smallest slope (in absolute value) of these previously drawn edges, then we choose a σ smaller than this slope.

A second difference from Lemma 5.1 is that we are also drawing edges in T_η between the two layers, so we must ensure that the edges in B_η do not intersect with those in T_η. The solution here is to ensure that each edge in T_η is drawn above and each edge in B_η is drawn below the horizontal line halfway between the two layers. In other words, for drawing edges in both B_η and T_η, we choose σ to be smaller than $\frac{\Delta_y}{\max(\mathrm{X}(v_p)-\mathrm{X}(v_1),\mathrm{X}(w_q)-\mathrm{X}(w_1))}$.

Figure 5.22(b) illustrates how we draw a typical handle graph.

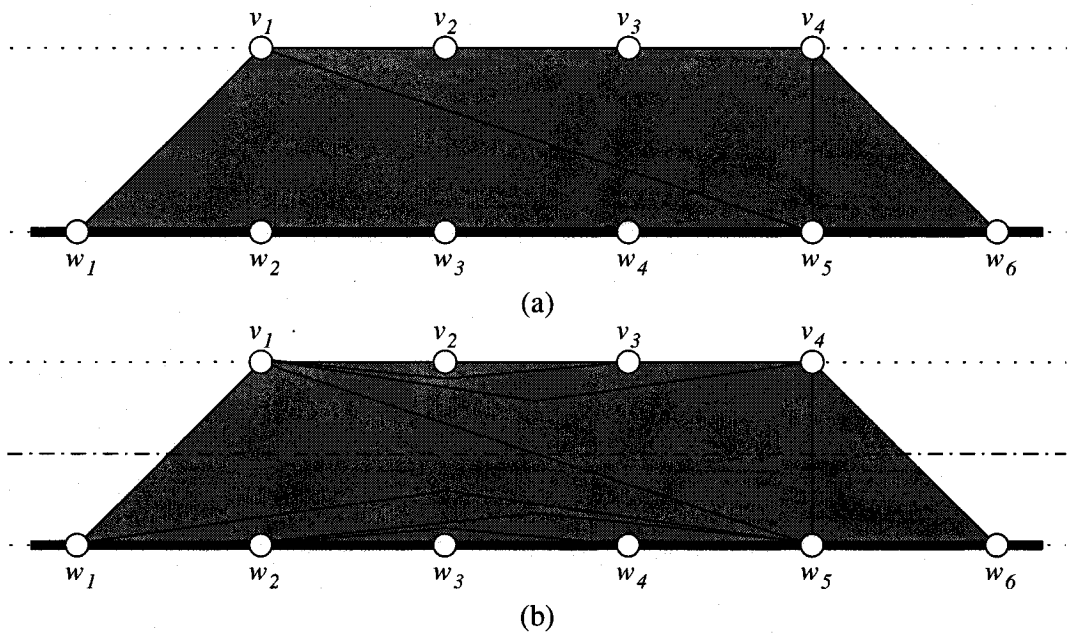


Figure 5.22: Drawing of a handle graph.

5.5.2.6 Re-inserting the Dangling Handles

In an earlier step described in Section 5.5.2.2, we replaced each dangling handle with a set of edges. Here, we describe how to re-insert these handles back into the drawing as a final step in our algorithm.

Let η be a dangling handle that lies on the left side of Π. By Property 5.1, then, anchor s of η lies on the bottom layer, so, by Property 5.6, the edges incident on s that we used to replace η bend above the top layer. Let $e = (s, v)$ be one such edge, and let c be the point nearest s in e where e crosses the top layer. We observe that e contains line segment $\overline{sc}$. We

also recall, from Section 5.5.2.2, that we inserted e to replace an edge between v and vertex w in η. Let b be the point where the two segments of e intersect. Then, we remove edge e and replace it with edge (v, w) by drawing vertex w at point c and drawing (v, w) with line segments $\overline{vb}$ and $\overline{bw}$ (see Figure 5.23).

Now suppose that some other vertex v' is adjacent to w in the original graph. We replace edge $e' = (s, v')$ with edge (v', w) by removing e' and then drawing (v', w) with segments $\overline{v'b'}$ and $\overline{b'w}$, where b' is the point where e' bends. We apply the same replacement procedure for every other vertex that is adjacent to w in the original graph. Figure 5.23 illustrates the result of replacing edges incident on s with vertex w and its incident edges.

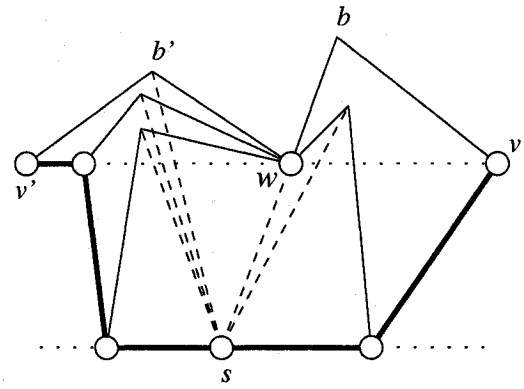


Figure 5.23: Reinserting a dangling handle vertex.

We show that these reinsertions do not create any edge crossings. Suppose, by way of contradiction, that we do create an edge crossing with an edge e_c (see Figure 5.24(a)). Clearly, edge (v, w) does not create the crossing because we can obtain its drawing by simply erasing the segment $\overline{sc}$ from edge $e = (s, v)$. We assume then, without loss of generality, that edge (v', w) creates the crossing. Segment $\overline{v'b'}$ belongs to edge $\overline{v's}$, so edge e_c crosses segment $\overline{b'w}$. This implies that e_c is incident on a vertex inside the triangle defined by s, c and b'. In our embedding, all edges incident on s between (v', s) and (v, s) are edges that replace the original edges incident on w. Therefore, if e_c is incident on s, then we have just replaced e_c by an edge incident on w so no crossing exists. Otherwise, e_c is incident on a vertex v_c on the top layerbetween (v', s) and (v, s). According to the embedding and because s is in Π, v_c does not belong to Π, so v_c belongs to a handle η'. Because v_c is between (v', s) and (v, s), handle η' is a dangling handle anchored at s. We observe that $\eta = \eta'$ because the edges of η' are between edges (v', s) and (v, s) which correspond to handle η in our embedding. Thus, v_c is in η and since it is between (v', s) and (v, s), it is in fact equal to w so no crossing exists (see Figure 5.24(b)).

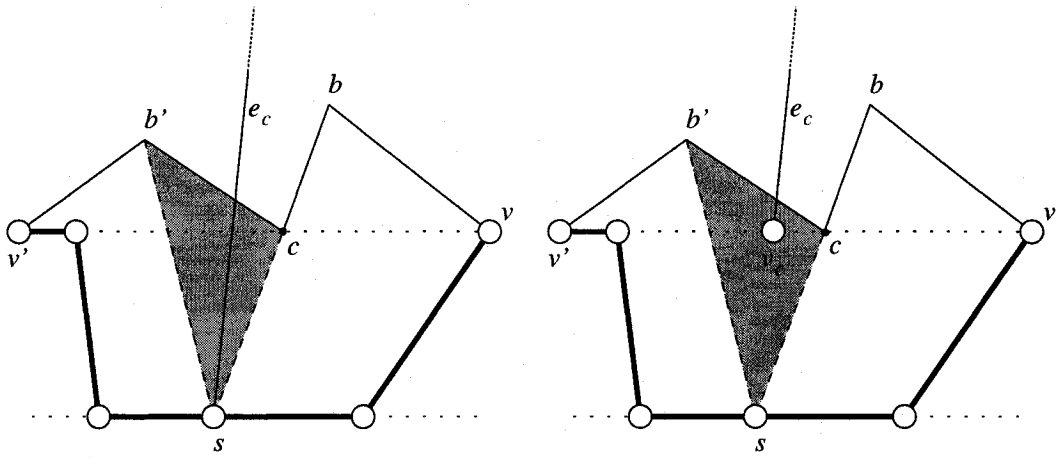


Figure 5.24: Edge e_c crosses reinserted edge $e' = (v'w)$ and is incident either on (a) anchor s on the bottom layer, or else on (b) vertex v_c on the top layer.

To complete the drawing of η, it remains for us to draw the following edges connecting pairs of vertices in η:

Edges that are inside the handle graph. This case is identical to the earlier step where we draw edges inside a non-dangling handle. Once again, then, we draw these edges using the technique of Kaufmann and Wiese as described in the proof of Lemma 5.1, but with additional upper bounds on σ described in Section 5.5.2.5.

Edges that are outside the handle graph and incident on s. Let e be such an edge, let $v_1, v_2, \ldots, v_p$ be the edges of η in the order that they appear on the top layer. Then, $e = (s, v_i)$ appears between (s, v_1) and Π or between (s, v_p) and Π in the cyclic ordering of the edges incident on s in the embedding of our graph. We assume, without loss of generality, the former case. We draw e with two segments that meet at a point b arbitrarily close v_1, higher than and to the left of v_1. We note that e does not cross any previously drawn edges and its drawing corresponds to our graph embedding.

Now consider another such edge $e' = (s, v_j)$ that also appears between (s, v_1) and Π in the cyclic ordering of the edges incident on s. It either appears before or after e in this ordering. In the first case, we draw e' with two segments that meet at a point b' that is arbitrarily close to b, higher than and to the left of b'. In the second case, b' is arbitrarily close to b, lower than and to the right of b', and, at the same time, higher than and to the left of v_1.

We draw other such edges in the same manner. Figure 5.25 illustrates how we draw two edges (s, v_2) and (s, v_3) outside the handle.

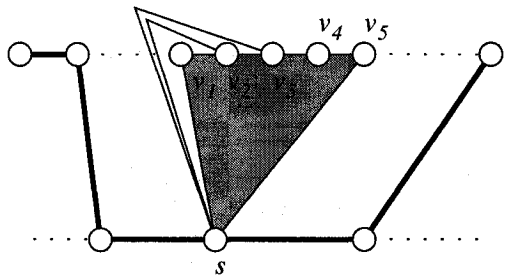


Figure 5.25: Drawing edges (s, v_2) and (s, v_3) outside the handle.

Edges outside the handle graph and not incident on s. These are edges between vertices in η that lie on the top layer. We draw these edges using the same technique as we used in Section 5.5.2.5 to draw edges inside non-dangling handles. The only change here is that the slope σ must be small enough that we don't create any crossings with other edges drawn outside the handle graph of η, particularly those between vertices in η and s. This is not a problem, however, because we can make σ arbitrarily close to 0.

Thus, we have proven the sufficiency of the characterization given in Theorem 5.19.

Lemma 5.22 (Sufficient Condition) *If a graph G is sub-Hamiltonian-with-handles, then G is 2-layer, 1-bend planar.*

Together, Lemmas 5.21 and 5.22 complete the proof of Theorem 5.19.

5.6 2-Outerplanar Graphs

Given the characterization of 2-layer, 1-bend planarity, it is natural to ask if there are families of planar graphs that satisfy the characterization. In this section, we consider 2-outerplanar graphs . A planar embedding of a graph is *outerplanar* if each vertex lies on the external face. A planar embedding of a graph is *2-outerplanar* if removing all vertices on the external face yields an outerplanar embedding of the remaining subgraph. Thus, a graph is *outerplanar* if it has an outerplanar embedding, and *2-outerplanar* if it has a 2-outerplanar embedding.

First, we observe that not all 2-outerplanar graphs are 1-layer, 1-bend planar. An example is shown in Figure 5.26 and we recall from Section 5.3 that this graph is not 1-layer, 1-bend planar.

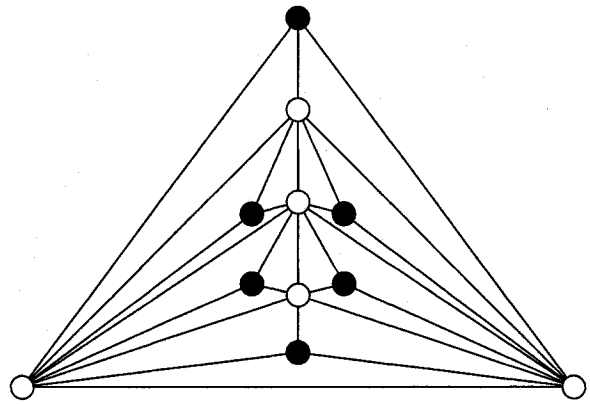


Figure 5.26: A 2-outerplanar graph that is not 1-layer, 1-bend planar. Called N^1 in Section 5.3.

Next, we use our characterization of 2-layer, 1-bend planar graphs to show that all 2-outerplanar graphs are 2-layer, 1-bend planar. This result is an interesting contrast to that of Cornelsen *et al.* [17, 18] which states that not all outerplanar graphs have planar drawings on two layers when edges must be drawn as straight-line segments.

Theorem 5.23 *Every 2-outerplanar graph is 2-layer, 1-bend planar and a 2-layer, 1-bend planar drawing can be computed in linear time in the number of vertices in the graph.*

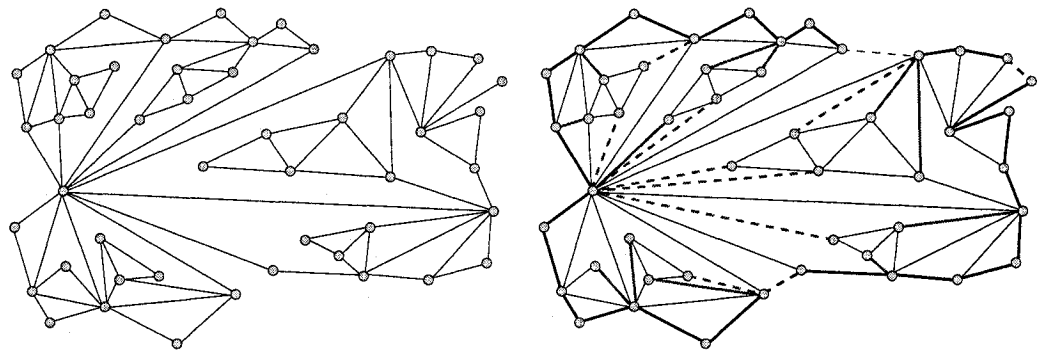
Proof: By Theorem 5.19 it is sufficient to prove the graph G is sub-Hamiltonian-with-handles. We assume that G is embedded with a 2-outerplanar embedding.

For convenience, we would like G to be biconnected. If G is not biconnected, then we make it biconnected by adding edges, so consider a cut-vertex v in G and suppose that it belongs to h biconnected components. Then, in the cyclic ordering of the edges incident on v, there are h consecutive pairs that belong to different biconnected components. Consider one such pair (v, u) and (v, w). We augment G by adding edge (u, w) for $h - 1$ of these pairs. We observe that the embedding of G remains 2-outerplanar because each edge that we add lies on the external face and each vertex that was on the external face before adding the edges remains on the external face. If we repeat this augmentation until there are no more cut-vertices, then we obtain a 2-outerplanar biconnected graph.

Also for convenience, we would like each vertex not on the external face to be adjacent to a vertex on the external face. If this is not the case for some vertex w not on the external

face, then, since G is 2-outerplanar, w belongs to a face containing at least one external face vertex v. Thus, we add the edge (v, w) to G and observe that G remains 2-outerplanar. We repeat this augmentation until each vertex not on the external face is adjacent to a vertex on the external face.

Since G is biconnected, the external face of G is bounded by a simple cycle C. Thus, we can cover the vertices of G with C minus one edge as our base path and one handle for each vertex w not in C consisting of a single edge connecting w to a vertex in C. Since these handles all lie inside C, they do not overlap (see Figure 5.27 for an example).



(a) (b)

Figure 5.27: (a) A 2-outerplanar embedded graph G. (b) A base path and a set of non-interleaving handles that cover the vertices of G. Dashed edges are dummy edges.

□

5.7 Conclusions and Open Problems

In this chapter, we have introduced the k-layer, 1-bend planar drawing convention for planar graphs. This convention can be used to describe, from a homogeneous perspective, several graph drawing problems studied in the literature. For example, the literature shows a connection between 2-page book embeddings and Hamiltonicity. Therefore, since 2-page book embeddings can be seen as 1-layer, 1-bend planar drawings, this leads to a new connection between 2-layer, 1-bend planar drawings and an extension of Hamiltonicity.

Given previous results, we asked in the introduction if all planar graphs are 2-layer, 1-bend planar, and then proved in this chapter that the answer is 'no', showing, in fact, that it is $\mathcal{NP}$-hard to determine whether or not a planar graph is 2-layer, 1-bend planar. Consequently, we have shown the surprising result that no number of layers lines is large enough to match the power of a single convex curve or allowing two bends per edge. Thus, our results suggest several new questions:

- What happens if we have two non-parallel layers?
- Are two non-parallel layers as powerful as three or more?
- What happens if we use a layer defined by a convex polyline? Here the layer approximates a convex layer so we might ask how many layer segments are necessary to draw all planar graphs. Certainly, if the number of segments is equal to $|V(G)| - 1$ and at most two vertices are placed on a single segment in the drawing, then there is a convex curve containing all the vertices.

Finally, we have also characterized 2-layer, 1-bend planarity by generalizing a characterization for 1-layer, 1-bend planarity. Consequently, we are interested in discovering further generalizations to k-layer, 1-bend planarity for $k \geq 3$. We gave one example of using our characterization to obtain efficient algorithms for computing 2-layer, 1-bend planar drawings of 2-outerplanar graphs. Perhaps there are other interesting classes that can also be drawn efficiently using this characterization.

Part II

Non-Planar Drawings

Overview of Part II

In this part of the thesis, we consider drawings of graphs on two layers that may contain edge crossings. Our approach to non-planar drawings is different than our approach to planar drawings. For planar drawings in Part I, given an $\mathcal{NP}$-hard problem, we generally found a tractable subproblem and obtained an efficient solution to it. Our hope was that the subproblem solution would give insight into solutions to more general problems.

In the case of drawings with edge crossings, however, there is a natural and systematic way to break the problems into manageable subproblems: by bounding a problem parameter by a constant. More specifically, for the 1-SIDED CROSSING MINIMIZATION problem, if we use the number of allowable edge crossings as a parameter and bound it by a constant, then the resulting problem has a polynomial solution. Similarly, for the 2-LAYER PLANARIZATION problem, if we use the number of allowable edge removals as a parameter and bound it by a constant, then this problem can also be solved in polynomial time. As discussed in Chapter 1, this approach is formalized in a relatively recent theory called parameterized complexity and the parameterized problems just described belong to the class $\mathcal{FPT}$.

The final chapter in this part presents experimental results based on our implementations of a few of these $\mathcal{FPT}$ algorithms. The purpose of the experiments is to compare the performance of these $\mathcal{FPT}$ algorithms in practice to previously implemented approaches like integer linear programming.

Chapter 6

Biplanarization Algorithms

To Sparrow—*your eyes present me with the most delightful mystery.*

In this chapter, we derive currently the most efficient $\mathcal{FPT}$ algorithms for solving biplanarization problems for two layers. We recall that the 2-LAYER PLANARIZATION problem is defined as follows:

> **Given:** A bipartite graph G and an integer $k \geq 0$.
> **Question:** Is there a set of edges $S \subseteq E$ of size at most k such that $G - S$ is biplanar?

and the 1-LAYER PLANARIZATION problem is defined as follows:

> **Given:** A bipartite graph $G = (A, B; E)$, an integer $k \geq 0$, and a linear ordering π of the vertices in A.
> **Question:** Is there a set of edges $S \subseteq E$ of size at most k such that $G - S$ and π is biplanar?

We also recall from Chapter 1 that both of these problems have many applications but that both are $\mathcal{NP}$-hard; consequently, we would like to find algorithms that can be used in at least some of the applications. Along these lines, Dujmović *et al.* [25] have already shown that both of these problems are fixed-parameter tractable when k is the problem parameter. They describe a $\mathcal{O}(k^2 \cdot 3^k + |G|^2)$ time algorithm for 1-LAYER PLANARIZATION, and a $\mathcal{O}(k \cdot 6^k + |G|)$ time algorithm for 2-LAYER PLANARIZATION. Using heuristics from efficient algorithms for the h-HITTING SET problem, Fernau [39] derives improved algorithms for both these problems with running times of $\mathcal{O}(k^3 \cdot 2.5616^k + |G|^2)$ and $\mathcal{O}(k^2 \cdot 5.1926^k + |G|)$, respectively.

In this chapter, we further improve these results by deriving a $\mathcal{O}(2^k + |G|)$-time algorithm for 1-LAYER PLANARIZATION, and a $\mathcal{O}(3.562^k + |G|)$-time algorithm for 2-LAYER PLANARIZATION. Like their predecessors, our algorithms organize the problem search space as a tree whose size is bounded by a function of k (see Section 1.3.5.1 for a general

description of bounded search trees). To each node in our bounded search trees, we associate a subgraph H of the original input graph G, and we generate children for the node by identifying small subgraphs of H that are not biplanar. Each child then corresponds to a different way of removing edges from the small subgraph so that it becomes biplanar. A leaf node in the search tree is one whose corresponding problem instance can easily be identified as either a no-instance or a yes-instance.

Our improvements are based on extending some of the heuristics used by Fernau [39] for efficiently biplanarizing dense subgraphs, and adding new heuristics for efficiently biplanarizing sparse induced subgraphs. Chen, Kanj and Jia [15] use a technique similar to this to obtain the fastest $\mathcal{FPT}$ algorithm for the VERTEX COVER problem. The main idea behind dense subgraph heuristics is that small dense subgraphs can only be made biplanar in a small number of ways, but each way involves removing a large number of edges. In terms of the search tree, this means that corresponding search tree nodes have a fewer children, and the subtree rooted at each child has reduced size. Indeed, Fernau [39] uses heuristics based on this observation to reduce the search tree size for 2-LAYER PLANARIZATION from approximately 6^k nodes to approximately 5.2^k nodes.

For sparse induced subgraphs, the situation is quite different because the number of edges that must be removed is quite small in comparison to its size. In this case, we show that we can obtain a correct solution without considering all possible ways of biplanarizing the subgraph. In terms of the search tree, this means that corresponding search tree nodes have fewer children.

In this chapter, we also describe how to incorporate our new divide-and-conquer heuristic to construct and traverse a bounded search tree. The idea is that sometimes the input graph can be split into two subgraphs whose minimum biplanarizing sets together form a minimum biplanarizing set for the main graph. Hence, it is more efficient to biplanarize the subgraphs separately rather than as a single graph. Though we do not yet know how to use this heuristic to improve the worst-case running time of our algorithms, we will show in Chapter 8 that it dramatically improves performance in experiments.

In Section 6.1, we derive a $\mathcal{O}(2^k \cdot |G|)$-time algorithm for 1-LAYER PLANARIZATION, followed by a $\mathcal{O}(3.562^k \cdot |G|)$-time algorithm for 2-LAYER PLANARIZATION in Section 6.2. Then, in Section 6.3, we show how to improve these running times to $\mathcal{O}(2^k + |G|)$ and $\mathcal{O}(3.562^k + |G|)$, respectively. Finally, in Section 6.4, we describe our divide-and-conquer approach.

6.1 One-Layer Planarization in $\mathcal{O}(2^k \cdot |G|)$ Time

The first $\mathcal{FPT}$ algorithm for solving 1-LAYER PLANARIZATION is described by Dujmović *et al.* [23]. We describe their algorithm here because our algorithm elaborates on their basic approach. It is based on the following result:

Lemma 6.1 (Dujmović *et al.* [23]) *Let $G = (A, B; E)$ be a bipartite graph and π a linear ordering of A. Then,* $\mathsf{bpn}(G, \pi) = 0$ *iff G is acyclic and the following condition holds:*

> *For every path (x, v, y) of G with $x, y \in A$, if $a \in A$ is between x and y in π, then the only edge incident to a (if any) is (a, v).* $(\star)$

A violation of $(\star)$ is illustrated in Figure 6.1.

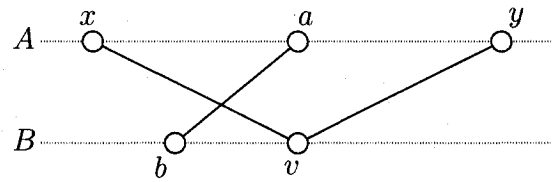


Figure 6.1: A violation of $(\star)$.

Using this result, they derive a bounded search tree algorithm for solving the 1-LAYER PLANARIZATION problem which we briefly describe. Each node N in the search tree corresponds to an instance of 1-LAYER PLANARIZATION whose input is a subgraph H_N of G, an integer $0 \leq k_N \leq k$, and, of course, linear ordering π. The root R of the tree corresponds to the original problem instance so $H_R = G$ and $k_R = k$. A node N has children if $k_N > 0$ and H_N does not satisfy $(\star)$. In fact, N has exactly three children, say N_1, N_2 and N_3, that correspond to some violation of $(\star)$ in H_N. In particular, H_N contains some path x, v, y for $x, y \in A$ and an edge (a, b) such that $a \in A$, $x <_\pi a <_\pi y$ and $b \neq v$. By Lemma 6.1, every biplanarizing set of H_N and π contains (a, b), (x, v) or (y, v). Therefore, we set $H_{N_1} = H_N - (a, b)$, $H_{N_2} = H_N - (x, v)$ and $H_{N_3} = H_N - (y, v)$, and each $k_{N_i} = k_N - 1$.

On the other hand, if $k_N = 0$ or H_N and π satisfies $(\star)$, then N is a leaf in the search tree. By Lemma 6.1, H_N, π and k_N is a yes-instance to the 1-LAYER PLANARIZATION problem if and only if H_N can be made acyclic by removing at most k_N edges. By the construction of the search tree, if H_N, π and k_N is a yes-instance, then this implies that G, π and k is a yes-instance to the 1-LAYER PLANARIZATION problem. In this case, the algorithm returns this positive result. On the other hand, if no such node is found, then, by the construction of the search tree, G, π and k are a no-instance so the algorithm returns this negative result.

By construction, the height of our search tree is at most k, and each non-leaf search node has exactly three children; therefore, it has size at most $\mathcal{O}(3^k)$. Dujmović *et al.* [23] show that the tree can be constructed and searched by spending $\mathcal{O}(|G|)$ time at each node. Consequently, their algorithm solves the 1-LAYER PLANARIZATION problem in $\mathcal{O}(3^k \cdot |G|)$ time.

In the remainder of this section, we describe more sophisticated branching rules to reduce the size of the search tree from $\mathcal{O}(3^k)$ down to $\mathcal{O}(2^k)$.

As in the original algorithm, branching is described in terms of some violation of $(\star)$. Consequently, suppose that we have a path (x, v, y) such that $x, y \in A$ and that there is a vertex $a \in A$ incident on a vertex $b \in B$ such that $x <_\pi a <_\pi y$ and $b \neq v$. We assume that x is the minimum such vertex in π and, given x, that y is maximum and a is minimum in π. Also, let $x' \in A$ be maximum in π and $y' \in A$ be minimum in π such that x' and y' are adjacent to v and $x' <_\pi a <_\pi y'$. Here is the first branching rule:

1EDGE If $\text{Adj}(a) \subseteq \{b, v\}$ and a is the only vertex strictly between x' and y' with degree greater than 0, then branch on removing (a, b) or (y', v).

See Figure 6.2 for an illustration. In the figure, a is the only vertex between x' and y' that is adjacent to another vertex. The dashed line between a and v shows that a may be adjacent to v. If this rule cannot be applied, then there is another edge $(a', b') \neq (a, b)$ such that

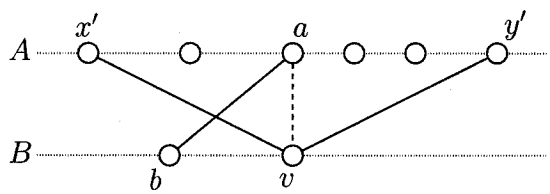


Figure 6.2: Illustration for branching rule 1EDGE.

$x' <_\pi a \leq_\pi a' <_\pi y'$ and $b' \neq v$. In this case, we apply one of the following rules:

X2EDGE If $x \neq x'$, then branch on removing $\{(v, x), (v, x')\}$, $\{(a, b), (a', b')\}$ or (y, v).

Y2EDGE If $y \neq y'$, then branch on removing $\{(v, y), (v, y')\}$, $\{(a, b), (a', b')\}$ or (x, v).

2EDGE If $x = x'$ and $y = y'$, then branch on removing $\{(a, b), (a', b')\}$ or (y, v).

See Figures 6.3, 6.4 and 6.5 for illustrations of these rules. We note that, in each case, we might have $a = a'$ or $b = b'$ but we cannot have both at the same time since $(a, b) \neq (a', b')$.

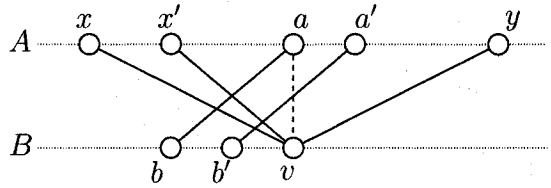


Figure 6.3: Illustration for branching rule X2EDGE.

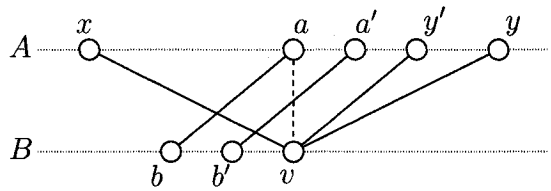


Figure 6.4: Illustration for branching rule Y2EDGE.

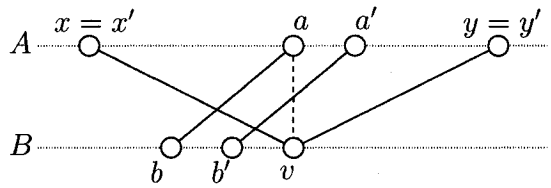


Figure 6.5: Illustration for branching rule 2EDGE.

Straight-forward analysis shows that the resulting search tree has size $T(k)$ bounded by:

$$T(k) \leq T(k-1) + 2T(k-2).$$

With $T(k)$ bounded by some constant for sufficiently small k, a simple proof by induction on k shows that $T(k) \in \mathcal{O}(2^k)$.

Next we show that any bounded search tree computed by applying these branching rules contains a node with a yes-instance to the 1-LAYER PLANARIZATION problem if and only if the original input problem is a yes-instance. In other words, for each tree node N with children obtained by applying one of the branching rules, we must show that the problem instance of N is a yes-instance if and only if the problem of at least one child of N is a yes-instance. This is equivalent to saying that there is a minimum biplanarizing set for H_N, the graph associated with N, that contains the edges removed in the branch corresponding to at least one child of N.

Lemma 6.2 *Let $G = (A, B; E)$ be a bipartite graph, and let π be a linear ordering of A.*

1. *If 1EDGE can be applied to G and π, then there exists a minimum biplanarizing set S for G and π such that $(a, b) \in S$ or $(y', v) \in S$.*

2. *If X2EDGE can be applied to G and π, then, for every minimum biplanarizing set S for G and π, $\{(v, x), (v, x')\} \subseteq S$, $\{(a, b), (a', b')\} \subseteq S$ or $(y, v) \in S$.*

3. *If Y2EDGE can be applied to G and π, then, for every minimum biplanarizing set S for G and π, $\{(v, y), (v, y')\} \subseteq S$, $\{(a, b), (a', b')\} \subseteq S$ or $(x, v) \in S$.*

4. *If 2EDGE can be applied to G and π, then there exists a minimum biplanarizing set S for G and π such that $\{(a, b), (a', b')\} \subseteq S$ or $(y, v) \in S$.*

Proof:

1. Let $S^\star$ be a minimum biplanarizing set such that $(a, b), (y', v) \notin S^\star$. By Lemma 6.1, then, we have $(x', v) \in S^\star$. By definition, there is a biplanar drawing Γ of subgraph $H = G - S^\star$, and, in Γ, we have $b < v$. Since a is the only vertex between x' and y' with degree greater than 0, (y', v) is in Γ, and $\mathrm{Adj}(a) \subseteq \{b, v\}$, then (a, b) is the only edge in Γ that crosses line segment $\overline{x'v}$. Therefore, we can obtain another biplanar drawing from Γ by removing edge (a, b) and inserting edge (x', v). In other words, $S^\star - (x', v) + (a, b)$ is a minimum biplanarizing set for G and π.

2. This statement follows directly from Lemma 6.1.

3. This statement also follows directly from Lemma 6.1.

4. Due the minimality of x and the maximality of y given x, each neighbor of v is between x and y in π. Since, in addition, we have that $x = x'$ and $y = y'$, then $\mathrm{Adj}(v) \subseteq \{x, y, a\}$.

   Let $S^\star$ be a minimum biplanarizing set such that $\{(a, b), (a', b')\} \not\subseteq S^\star$ and $(y, v) \notin S^\star$. By Lemma 6.1, then, we have $(x, v) \in S^\star$ and there is a biplanar drawing Γ of subgraph $H = G - S^\star$. Let X be the set of edges that cross line segment $\overline{xv}$ in Γ. We consider two cases:

   (a) *Each edge in X is incident on a.*
   In other words, we have $X = \{(a, b_1), (a, b_2), \ldots, (a, b_p)\}$ where $b_1 < b_2 < \ldots < b_p$ in Γ. Because of the minimality of x, the neighbors of each b_i are greater than or equal to x in π. Because (y, v) is in Γ, we also have $b_i < v$. Now consider the following drawing obtained from Γ: remove edge (y, v) (and edge

(v, a) if it exists), move v so that it lies immediately before b_1, and then insert edge (x, v) (and edge (v, a) if it exists). The resulting drawing is biplanar so $S^\star - (x, v) + (y, v)$ is a minimum biplanarizing set for G and π. See Figure 6.6 for an illustration of this modification.

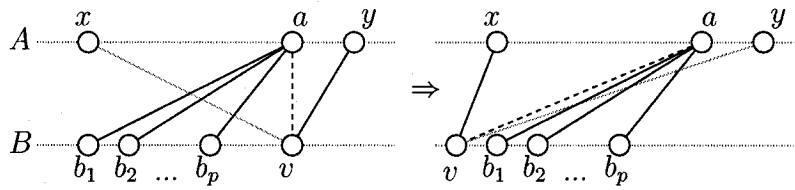


Figure 6.6: How to modify the drawing when each edge in X is incident on a.

(b) *An edge in X is not incident on a.*

In other words, we have $X = \{(a_1, b_1), (a_2, b_2), \ldots, (a_p, b_p)\}$ where $x <_\pi a_1 \leq_\pi a_2 \leq_\pi \cdots \leq_\pi a_p <_\pi y$ and $b_1 \leq b_2 \leq \ldots \leq b_p$ in Γ. In this case, we have $\mathrm{Adj}(v) \subseteq \{x, y\}$ because Γ contains edge (y, v) and the edge in X not incident on a. Because of the minimality of x, the neighbors of each b_i are greater than or equal to x in π. Therefore, the drawing obtained from Γ by removing (y, v), moving v immediately before b_1 and then inserting edge (x, v) is biplanar. In other words, $S^\star - (x, v) + (y, v)$ is a minimum biplanarizing set for G and π. See Figure 6.7 for an illustration of this modification.

□

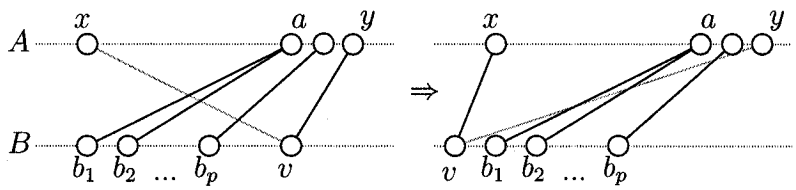


Figure 6.7: How to modify the drawing when an edge in X is not incident on a.

To compute and traverse the search tree, it is necessary to spend at most $\mathcal{O}(|G|)$ time at each search tree node so we have a $\mathcal{O}(2^k \cdot |G|)$ time algorithm for 1-LAYER PLANARIZATION.

Theorem 6.3 *For any given bipartite graph $G = (A, B; E)$, linear ordering π of A and integer $k \geq 0$, there is an algorithm that determines whether or not* $\mathsf{bpn}(G, \pi) \leq k$ *in $\mathcal{O}(2^k \cdot |G|)$ time.*

6.2 Two-Layer Planarization in $\mathcal{O}(3.562^k \cdot |G|)$ Time

The first $\mathcal{FPT}$ algorithm for 2-LAYER PLANARIZATION is also described by Dujmović *et al.* [23]. Once again, we describe their algorithm here because our algorithm elaborates on their basic approach. Based on Lemma 2.1, they describe a bounded search tree algorithm, much like their algorithm for 1-LAYER PLANARIZATION described in the previous section, that selects a "forbidden structure" at each search tree node, and then creates one child for each edge in the structure. The next lemma describes these structures:

Lemma 6.4 (Dujmović *et al.* [23]) *If there exists a vertex v in a graph G such that $\deg'(v) \geq 3$, then v belongs to a 2-claw or a 3- or 4-cycle in G.*

Proof: Let u_1, u_2, u_3 be three distinct non-leaf neighbors of v, and let w_1, w_2, w_3 be neighbors of u_1, u_2, u_3, respectively, that are distinct from v. If $w_i = u_j$ for some i and j, then vu_ju_i is a 3-cycle. On the other hand, if $w_i \neq u_j$ for each i and j but $w_i = w_j$ for some $i \neq j$, then $vu_iw_iu_j$ is a 4-cycle. If neither of these is true, then vertices $v, u_1, u_2, u_3, w_1, w_2, w_3$ form a 2-claw rooted at v. □

We call a 2-claw or a 3- or 4-cycle in a graph a *forbidden structure*.

We construct the bounded search tree recursively, beginning at the root. Given a graph G and integer $k \geq 0$ as input to the algorithm, we associate a subgraph H_N of G and an integer $0 \leq k_N \leq k$ to each search tree node N. For root node R, $H_R = G$ and $k_R = k$. A node N has children if $k_N > 0$ and H_N contains a forbidden structure S. By Lemma 2.1, at least one edge in S is in every biplanarizing set of H_N; consequently, the node has $|S|$ children, one child N' for each edge e in S. Thus, we set $H_{N'} = H_N - e$ and $k_{N'} = k_N - 1$. A node N is a leaf if $k_N = 0$ or H_N contains no forbidden structures. If H_N contains no forbidden structures, then, by Lemma 6.4, every vertex v in H_N has $\deg'_{H_N}(v) \leq 2$; in other words, each connected component in H_N is either a caterpillar or a wreath (see Chapter 2). By Lemma 2.1, every minimum biplanarizing set contains exactly one cycle edge from each component wreath in H_N. Thus, a leaf node represents a yes-instance to the problem if its subgraph H_N does not contain any forbidden structures and contains at most k_N component wreaths.

The resulting tree has at most $\mathcal{O}(6^k)$ nodes because each node has at most 6 children, and the height of the tree is at most k. It is possible to construct and traverse this tree while spending at most $\mathcal{O}(|G|)$ time at each node; therefore, we have an $\mathcal{O}(6^k \cdot |G|)$ time algorithm for solving the 2-LAYER PLANARIZATION problem.

We can dramatically improve the running time of this algorithm by refining our branching rules. For each rule, let v be a vertex with three non-leaf neighbors v_1, v_2 and v_3. We

assume then that $\deg(v_1) \geq \deg(v_2) \geq \deg(v_3) \geq 2$, and, for each v_i, we let $v_{i1} \neq v$ be a neighbor of v_i, and, if $\deg(v_i) \geq 3$, then $v_{i2} \neq v$ is another neighbor of v_i. We let $e_i = (v, v_i)$ and $e_{ij} = (v_i, v_{ij})$ for each $i = 1, 2, 3$ and $j = 1, 2$ (see Figure 6.8). The

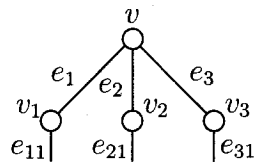


Figure 6.8: A vertex v with three non-leaf neighbors.

following rules are illustrated in Figures 6.9-6.13.

3CYC If $v_i = v_{i'j'}$ for some $i \neq i'$ and $j' \in \{1, 2\}$, then branch on removing edges e_i, $e_{i'j'}$ or $e_{i'}$.

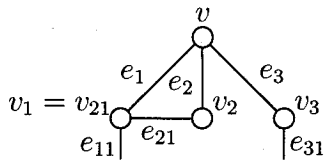


Figure 6.9: 3CYC: a 3-cycle because $v_1 = v_{21}$.

CLAW0 If $\deg(v_1) = 2$, then branch on removing edges e_{11}, e_{21}, e_{31} or $\{e_1, e_2\}$.

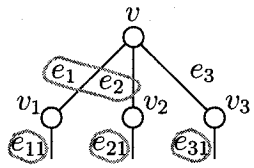


Figure 6.10: CLAW0: $\deg(v_1) = \deg(v_2) = \deg(v_3) = 2$.

CLAW1 If $\deg(v_1) > 2$ and $\deg(v_2) = 2$, then branch on removing e_1, $\{e_{11}, e_{12}\}$, e_{21}, e_{31}, or $\{e_2, e_3\}$.

CLAW2 If $\deg(v_2) > 2$ and $\deg(v_3) = 2$, then branch on removing $\{e_{11}, e_{12}\}$, $\{e_{21}, e_{22}\}$, e_{31}, e_1, or e_2.

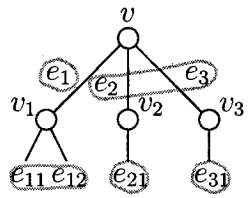


Figure 6.11: CLAW1: $\deg(v_1) > 2$ and $\deg(v_2) = \deg(v_3) = 2$.

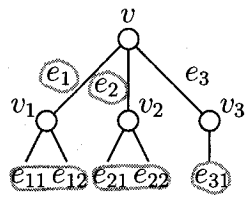


Figure 6.12: CLAW2: $\deg(v_1), \deg(v_2) > 2$ and $\deg(v_3) = 2$.

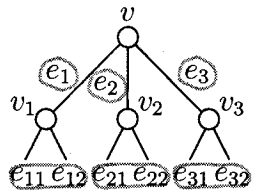


Figure 6.13: CLAW3: $\deg(v_1), \deg(v_2), \deg(v_3) > 2$.

CLAW3 If $\deg(v_3) > 2$, then we branch on removing e_1, e_2, e_3, $\{e_{11}, e_{12}\}$, $\{e_{21}, e_{22}\}$, or $\{e_{31}, e_{32}\}$.

We apply 3CYC whenever it is applicable so that, in each of the remaining rules, we can assume that $v_i \neq v_{i',j'}$ for each $i \neq i'$ and $j \in \{1, 2\}$.

Our proof of the correctness of these rules is similar to Lemma 6.2 for the 1-LAYER PLANARIZATION problem. In other words, given a search tree node N and any applicable branching, we prove that there is a minimum biplanarizing set for H_N that contains the edge(s) of a branch corresponding to at least one of the children of N. We will use the following lemma repeatedly to obtain our result:

Lemma 6.5 *Let $e = (u, v)$ be an edge in a graph G. For each vertex w in $V(G)$, $\deg'_{G-e}(w) < \deg'_G(w)$ if and only if:*

1. *$w = u$ and $\deg_G(v) \geq 2$; or*

2. $w = v$ *and* $\deg_G(u) \geq 2$; *or*

3. $(u, w) \in E(G)$ *and* $\deg_G(u) = 2$; *or*

4. $(v, w) \in E(G)$ *and* $\deg_G(v) = 2$.

Proof: We observe that only u and v can become leaves by removing edge e. Therefore, $\deg'(w)$ is reduced only if $w \cap \{u, v\} \neq \emptyset$ or $\text{Adj}(w) \cap \{u, v\} \neq \emptyset$. □

Lemma 6.6 *Let G be a graph.*

1. *If 3CYC can be applied to G, then for every minimum biplanarizing set S of G we have $e_{i'} \in S$, $e_{i'j'} \in S$, or $e_i \in S$.*

2. *If CLAW0 can be applied to G, then there is a minimum biplanarizing set S of G such that $e_{11} \in S$, $e_{21} \in S$, $e_{31} \in S$, or $\{e_1, e_2\} \subseteq S$.*

3. *If CLAW1 can be applied to G, then there is a minimum biplanarizing set S of G such that $e_1 \in S$, $\{e_{11}, e_{12}\} \subseteq S$, $e_{21} \in S$, $e_{31} \in S$, or $\{e_2, e_3\} \subseteq S$.*

4. *If CLAW2 can be applied to G, then there is a minimum biplanarizing set S of G such that $\{e_{11}, e_{12}\} \subseteq S$, $\{e_{21}, e_{22}\} \subseteq S$, $e_{31} \in S$, $e_1 \in S$, or $e_2 \in S$.*

5. *If CLAW3 can be applied to G, then for every minimum biplanarizing set S of G we have $e_1 \in S$, $e_2 \in S$, $e_3 \in S$, $\{e_{11}, e_{12}\} \subseteq S$, $\{e_{21}, e_{22}\} \subseteq S$, or $\{e_{31}, e_{32}\} \subseteq S$.*

Proof:

1. Follows immediately from Lemma 2.1 because $v, v_{i'}, v_{i'j'}$ is a 3-cycle.

2. Let $S^\star$ be a minimum biplanarizing set such that $e_{i1} \notin S^\star$ and $\{e_1, e_2\} \not\subseteq S^\star$. Since $S^\star$ is a biplanarizing set, then, without loss of generality, $S^\star$ contains some e_i. Let H be the subgraph of G obtained by removing the edges of $S^\star$ from G, and let H'' be the subgraph of H obtained by removing edge e_{i1} from H. Since $S^\star$ is a biplanarizing set, H and H'' contain no 2-claws or cycles. If we let $H' = H'' + e_i$, then vertex v_1 is a leaf in H'; thus, H' contains no cycles because H'' contains no cycles. Since H'' contains no 2-claws, then, by Lemma 6.5, H' may contain a 2-claw only if v is a leaf in H'' and its only neighbor in H'' is the 2-claw root. Since edge e_1 or e_2 belongs to H'', then v_1 or v_2 is the only neighbor of v in H''. However, neither of these vertices is the root of a 2-claw because each has degree at most 2 in H'. Therefore, H' is biplanar so $S^\star - e_i + e_{i1}$ is a minimum biplanarizing set.

3. Let $S^\star$ be a minimum biplanarizing set such that $e_1, e_{21}, e_{31} \notin S^\star$, $\{e_{11}, e_{12}\} \not\subseteq S^\star$ and $\{e_2, e_3\} \not\subseteq S^\star$. Without loss of generality, we assume that $e_{11}, e_2 \notin S^\star$. Since $S^\star$ is a biplanarizing set, $S^\star$ contains edge e_3. Let H be the subgraph of G obtained by deleting the edges of $S^\star$, and let H'' be the subgraph of H obtained by deleting edge e_{31}. Since $S^\star$ is a biplanarizing set, H and H'' are acyclic and do not contain any 2-claws. Let $H' = H'' + e_3$. Since v_3 is a leaf in H', then H' is acyclic because H'' is acyclic. Since H'' contains no 2-claws, then, by Lemma 6.5, H' may contain a 2-claw only if v is a leaf in H''. By assumption, however, v is adjacent to v_1 and v_2 in H and H''; therefore, H' contains no 2-claws. In other words, H' is biplanar so $S^\star - e_3 + e_{31}$ is a minimum biplanarizing set for G.

4. Let $S^\star$ be a minimum biplanarizing set such that $e_1, e_2, e_{31} \notin S^\star$, $\{e_{11}, e_{12}\} \not\subseteq S^\star$ and $\{e_{21}, e_{22}\} \not\subseteq S^\star$. Since $S^\star$ is a biplanarizing set, then $S^\star$ contains edge e_3. Since we have $e_1, e_2 \notin S^\star$, the situation is the same as for CLAW2 so $S^\star - e_3 + e_{31}$ is a minimum biplanarizing set for G.

5. Follows immediately from Lemma 2.1.

□

Let $T(k)$ denote the size of the search tree created by applying these rules. For sufficiently small k, $T(k)$ is bounded by a small constant, so for larger k, we obtain the following upper bound for $T(k)$:

$$T(k) \leq 3T(k-1) + 3T(k-2).$$

A simple proof by induction shows that $T(k) \in \mathcal{O}\left(\left(\frac{3+\sqrt{21}}{2}\right)^k\right) \approx \mathcal{O}(3.8^k)$, so we have the following result:

Lemma 6.7 *For any given graph G and integer $k \geq 0$, there is an algorithm that determines whether or not* $\mathsf{bpn}(G) \leq k$ *in $\mathcal{O}(3.8^k \cdot |G|)$ time.*

To further reduce the running time to $\mathcal{O}(3.562^k \cdot |G|)$ time, we replace branching rule CLAW3 with new rules since it is the rule that leads to largest branching factor. In each, we assume, as in rule CLAW3, that $\deg(v_1), \deg(v_2), \deg(v_3) > 2$. The first new rule handles the case where some v_{ij} is a leaf vertex. Like Dujmović *et al.* [25], we show that search tree branches corresponding to removing an edge incident on a leaf vertex can be ignored.

LEAF If some vertex v_{ij} is a leaf for some $i \in \{1,2,3\}$ and $j \in \{1,2\}$, then branch on removing the same sets of edges as in rule CLAW3 except the set containing e_{ij} which we omit.

In the following proof of correctness, we consider only the case where $i = 1$ because the other cases are symmetric.

Lemma 6.8 *Let G be a graph. If LEAF can be applied to G because v_{1j} is a leaf for some $j \in \{1,2\}$, then there exists a minimum biplanarizing set S for G such that $e_1 \in S$, $e_2 \in S$, $e_3 \in S$, $\{e_{21}, e_{22}\} \subseteq S$, or $\{e_{31}, e_{32}\} \subseteq S$.*

Proof: Let $S^\star$ be a minimum biplanarizing set that does not satisfy the lemma statement, and let H be the subgraph of G obtained by deleting the edges of $S^\star$. Since $S^\star$ is a biplanarizing set, we have $\{e_{11}, e_{12}\} \subseteq S^\star$ and, furthermore, in H, $\mathrm{Adj}(v_1) = \{v\}$ and $\mathrm{Adj}(v_{1j}) = \emptyset$. Thus, $H - e_1 + e_{1j}$ is biplanar, so $S^\star - e_{1j} + e_1$ is a minimum biplanarizing set. □

The next branching rule applies whenever we have $v_{ij} = v_{i'j'}$. When this is the case, our structure contains a 4-cycle $vv_iv_{ij}v_{i'}$. By Lemma 2.1, every biplanarizing set for the graph contains at least one of the cycle edges. Therefore, we obtain a new branching rule that extends the CLAW3 rule so that each set removes at least one edge of the 4-cycle. This rule is illustrated in Figure 6.14:

4CYC If $v_{ij} = v_{i'j'}$ for some $i \neq i' \in \{1,2,3\}$ and $j, j' \in \{1,2\}$, then branch on removing the same sets of edges as in rule CLAW3 except that, for each set X that does not contain e_i, e_{ij}, $e_{i'}$ or $e_{i'j'}$, we create two sets $X + e_{ij}$ and $X + e_{i'j'}$.

e.g. if $v_{1j} = v_{2j'}$, then branch on removing $\{e_1\}$, $\{e_2\}$, $\{e_3, e_{1j}\}$, $\{e_3, e_{2j'}\}$ $\{e_{11}, e_{12}\}$, $\{e_{21}, e_{22}\}$, $\{e_{31}, e_{32}, e_{1j}\}$, or $\{e_{31}, e_{32}, e_{2j'}\}$.

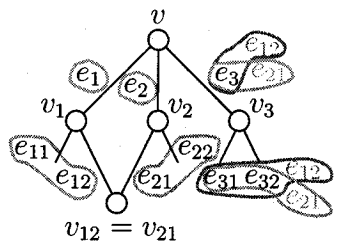


Figure 6.14: 4CYC: $v_{12} = v_{21}$.

In the following proof of correctness, we consider only the case where $i = 1$ and $i' = 2$ since the other cases are symmetric.

Lemma 6.9 *Let G be a graph. If 4CYC can be applied to G for because $v_{1j} = v_{2j'}$ for some $j, j' \in \{1,2\}$, then, for every minimum biplanarizing set S for G, $e_1 \in S$, $e_2 \in S$,*

$\{e_3, e_{1j}\} \subseteq S$, $\{e_3, e_{2j'}\} \subseteq S$, $\{e_{11}, e_{12}\} \subseteq S$, $\{e_{21}, e_{22}\} \subseteq S$, $\{e_{31}, e_{32}, e_{1j}\} \subseteq S$, *or* $\{e_{31}, e_{32}, e_{2j'}\} \subseteq S$.

Proof: By way of contradiction, let $S^\star$ be a minimum biplanarizing set that does not satisfy the lemma statement, and let H be the subgraph of G obtained by deleting the edges of $S^\star$. Thus, we have e_{21} or $e_{22} \notin S^\star$, e_{11} or $e_{12} \notin S^\star$, and $e_1, e_2 \notin S^\star$. Since $S^\star$ is a biplanarizing set, then $e_3 \in S^\star$ or $\{e_{31}, e_{32}\} \subseteq S^\star$. In either case, we have $e_{1j}, e_{2j'} \notin S^\star$ by assumption so H contains a 4-cycle $vv_1v_{1j}v_2$. □

If neither of these two rules is applicable, then each vertex v_{ij} is adjacent to a vertex $v_{ij1} \neq v_i$ and possibly another vertex $v_{ij2} \neq v_i$. We let $e_{ij1} = (v_{ij}, v_{ij1})$, and, if v_{ij2} exists, then we let $e_{ij2} = (v_{ij}, v_{ij2})$. We observe that each v_i has three non-leaf neighbors v, v_{i1} and v_{i2} so we may be able to apply one of the branching rules above to the structure centered at v_i instead of v. For example, for $i = 1$, we would consider the structure consisting of vertices v_1, v, v_2, v_3, v_{11}, v_{111}, v_{12}, and v_{121} (and v_{112} and v_{122} if they exist).

In the final rule below, then, we assume none of the rules above are applicable for any of these structures. Thus, since CLAW0, CLAW1 and CLAW2 are not applicable, then each v_i has at least three neighbors including v, v_{i1} and v_{i2}, and similarly, each v_{ij} has at least three neighbors including v_i, v_{ij1} and v_{ij2}. Since, in addition, rule 4CYC is not applicable, we have that $|\{v, v_1, v_2, v_3, v_{11}, v_{12}, v_{21}, v_{22}, v_{31}, v_{32}\}| = 10$. Since 3CYC is not applicable, $e_{i1l} \neq e_{i2l'}$ for each $i \in \{1, 2, 3\}$ and $l, l' \in \{1, 2\}$.

With these assumptions, we describe the final branching rule. As mentioned above, we not only have a CLAW3 structure centered at v but also at each v_i for each $i \in \{1, 2, 3\}$. In Figure 6.15, we highlight the CLAW3 structure centered at v_1. Consequently, we have

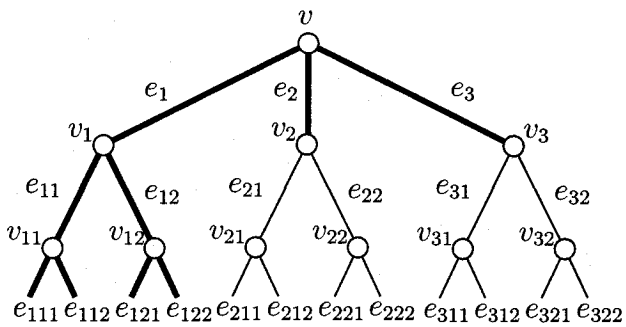


Figure 6.15: CLAWS subgraph.

CLAW3 structures that overlap so, like branching rule 4CYC, we branch on the two structures at the same time rather than consecutively. As we will see, this results in a smaller

overall branching factor in the search tree. To obtain a correct branching rule, we first recall that rule CLAW3 generates the following sets of edges: $\{e_1\}$, $\{e_2\}$, $\{e_3\}$, $\{e_{11}, e_{12}\}$, $\{e_{21}, e_{22}\}$, and $\{e_{31}, e_{32}\}$. Similarly, when we apply CLAW3 to the structure centered at v_1, it generates the following sets of edges: $\{e_1\}$, $\{e_{11}\}$, $\{e_{12}\}$, $\{e_2, e_3\}$, $\{e_{111}, e_{112}\}$, and $\{e_{121}, e_{122}\}$. We combine these two applications into a single rule by combining all pairs of sets, one from the application for v and the other from the application for v_1. Because the applications overlap on sets of edges, some of the resulting pairs are redundant. For example, both $\{e_1\}$ and $\{e_1, e_{11}\}$ are sets obtained by combining a set from the application for v with a set from the application for v_1. The second set $\{e_1, e_{11}\}$ is redundant because it corresponds to a solution only if $\{e_1\}$ corresponds to a solution. More generally, a set is redundant whenever it is a superset of another set. Thus, whereas there are $6 \cdot 6 = 36$ possible sets from the two applications and therefore 36 possible branches, 17 of them are redundant so we actually only consider 19 branches.

CLAWS If $|\{v, v_1, v_2, v_3, v_{11}, v_{12}, v_{21}, v_{22}, v_{31}, v_{32}\}| = 10$ and $e_{il} \neq e_{i2l'}$ for each $i \in \{1, 2, 3\}$ and $l, l' \in \{1, 2\}$, then branch on removing $\{e_1\}$, $\{e_{11}, e_{12}\}$, $\{e_2, e_3\}$, $\{e_2, e_{11}\}$, $\{e_2, e_{12}\}$, $\{e_2, e_{111}, e_{112}\}$, $\{e_2, e_{121}, e_{122}\}$, $\{e_3, e_{11}\}$, $\{e_3, e_{12}\}$, $\{e_3, e_{111}, e_{112}\}$, $\{e_3, e_{121}, e_{122}\}$, $\{e_{21}, e_{22}, e_{11}\}$, $\{e_{21}, e_{22}, e_{12}\}$, $\{e_{21}, e_{22}, e_{111}, e_{112}\}$, $\{e_{21}, e_{22}, e_{121}, e_{122}\}$, $\{e_{31}, e_{32}, e_{11}\}$, $\{e_{31}, e_{32}, e_{12}\}$, or $\{e_{31}, e_{32}, e_{111}, e_{112}\}$, $\{e_{31}, e_{32}, e_{121}, e_{122}\}$,

We prove that this rule is correct:

Lemma 6.10 *Let G be a graph. If CLAWS can be applied to G, then every minimum biplanarizing set for G contains at least one subset listed by rule CLAWS.*

Proof: By way of contradiction, let $S^\star$ be a minimum biplanarizing set that does not satisfy the lemma statement, and let H be the subgraph of G obtained by deleting the edges of $S^\star$. By this assumption, then, $e_1 \notin S^\star$ and either e_{11} or $e_{12} \notin S^\star$. However, since $S^\star$ is a biplanarizing set, then $S^\star$ contains e_2, e_3, $\{e_{21}, e_{22}\}$, or $\{e_{31}, e_{32}\}$. By assumption, however, each of these four cases implies that v, v_{11} and v_{12} are non-leaf neighbors of v in H, a contradiction by Lemma 2.1. □

Using these rules, we obtain the following bound for the size $T(k)$ of the resulting search tree:

$$T(k) \leq \max \{ 3T(k-1) + 2T(k-2), 2T(k-1) + 4T(k-2) + 2T(k-3), \\ T(k-1) + 6T(k-2) + 8T(k-3) + 4T(k-4)\}.$$

A simple proof by induction shows that $T(k) \in \mathcal{O}\left(\left(\frac{3+\sqrt{17}}{2}\right)^k\right) \approx \mathcal{O}(3.562^k)$.

Thus, we have the following result:

Theorem 6.11 *For any given graph G and integer $k \geq 0$, there is an algorithm that determines whether or not* $\mathsf{bpn}(G) \leq k$ *in* $\mathcal{O}(3.562^k \cdot |G|)$ *time.*

6.3 Achieving Constant Time Per Node

In this section, we show that the running times of the algorithms described in the previous sections can be improved so that, on average, constant time is spent at each search tree node. In particular, we show how to solve the 1-LAYER PLANARIZATION problem in $\mathcal{O}(2^k + |G|)$ time, and the 2-LAYER PLANARIZATION problem in $\mathcal{O}(3.562^k + |G|)$ time.

6.3.1 1-LAYER PLANARIZATION in $\mathcal{O}(2^k + |G|)$ Time

To achieve a running time of $\mathcal{O}(2^k + |G|)$, we show how to utilize $\mathcal{O}(|G|)$ initialization time in order to spend at most $\mathcal{O}(k_N)$ time at each search tree node N. If we can do this, then the running time $T(k)$ after initialization has the following bound:

$$T(k) \leq T(k-1) + 2T(k-2) + \mathcal{O}(k).$$

It is well-known that the solution to this recurrence is dominated by the exponential part so $T(k) \in \mathcal{O}(2^k)$ (see e.g. [80]). Therefore, assuming that we can indeed use $\mathcal{O}(k_N)$ time at each search tree node, we have obtained an algorithm that runs in $\mathcal{O}(2^k + |G|)$ time.

To achieve this bound, we require some additional data structures. To this end, we present a few definitions. Consider a problem instance consisting of a bipartite graph $G = (A, B; E)$, a linear ordering π of A, and an integer $k \geq 0$. A vertex $v \in B$ is called a star-violator if it has two neighbors $x, y \in A$ and there exists an edge (a, b) such that $a \in A$, $x <_\pi a <_\pi y$, and $b \neq v$. A vertex $v \in B$ is called a *diamond-violator* if it has exactly two distinct neighbors $x, y \in A$ and there exists another vertex $v' \in B$ also adjacent only to x and y. The term *violator* then applies to any star-violator or diamond-violator.

To each node N, we not only associate a subgraph H_N of G and an integer $0 \leq k_N \leq k$, but we also associate three additional data structures:

1. S_N, the set of star-violators in H_N;
2. D_N, the set of diamond-violators in H_N that do not belong to S_N; and
3. T_N, the induced subgraph of H_N composed of vertices in $B - S_N$ that have degree two along with their neighbors in H_N.

Thus, $S_N \cup D_N$ is the set of violators in H_N. We first show H_N contains no violators if and only if H_N and π is biplanar.

Lemma 6.12 *A bipartite graph $G = (A, B; E)$ and linear ordering π of A contains no violators if and only if* $\mathsf{bpn}(G, \pi) = 0$.

Proof: ($\Rightarrow$) By definition, G and π satisfy ($\star$), so, by Lemma 6.1, it remains for us to prove that G is acyclic. Assume, by way of contradiction, that G contains a cycle C. If C is a 4-cycle, then C contains two diamond-violators; therefore, C contains at least five vertices so let a_1, b_1, a_2, b_2, a_3 be a subpath of C such that a_2 is the minimum vertex of C in π. If $a_3 <_\pi a_1$, then path a_1, b_1, a_2 and edge (b_2, a_3) show that b_1 is a star-violator. Thus, $a_3 >_\pi a_1$; however, in that case, path a_2, b_2, a_3 and edge (a_1, b_1) show that b_2 is a star-violator.

($\Leftarrow$) By Lemma 6.1, G satisfies ($\star$), so G contains no star-violators. The existence of a diamond-violator implies the existence of a cycle, so G contains no violators. □

For each node N in the search tree, we look for an application of a branching rule using the data structures described above. The next lemma, then, is a key to spending at most $\mathcal{O}(k_N)$ at node N because it shows that $|S_N| + |D_N| \leq 2k_N$ whenever $\mathsf{bpn}(H_N) \leq k_N$.

Lemma 6.13 *Let $G = (A, B; E)$ be a bipartite graph and π a linear ordering of A. If* $\mathsf{bpn}(G, \pi) \leq k$ *for some integer $k \geq 0$, then G and π contain at most $2k$ violators.*

Proof: Let $e = (a, b)$ be an edge in G, $a \in A$ and $b \in B$, and let $H = G - e$. Also, let B_G be the set of violators in G and π, and let B_H be the set of violators in H and π. It is sufficient to prove that $|B_H| \geq |B_G| - 2$.

Let $v, v' \neq b$ be vertices in B_G but not in B_H. We will show that $v = v'$ so, suppose, by way of contradiction, that $v \neq v'$. Since $v, v' \in B_G$, v has two neighbors $x, y \in A$ in G and v' has two neighbors $x', y' \in A$ in G. We assume that x and x' are minimum and y and y' are maximum possible. Since $v, v' \neq b$, v and v' have the same neighbors in H as in G. We assume without loss of generality that $x \leq_\pi x'$. By assumption, then, we have that $y \leq_\pi x'$. Since $v, v' \in B_G$, then, we cannot have $a <_\pi x'$ because then $v' \in B_H$. Similarly, we cannot have $a >_\pi y$ because then $v \in B_H$. Thus, $y \leq_\pi a \leq_\pi x'$. However, in this case, it is not possible that edge (a, b) makes v and v' star-violators in G, so (a, b) must make them both diamond-violators. This is possible for v if b is adjacent to x and for v' if b is adjacent to y'. However, this implies that $\deg(b) = 3$ in G meaning that (a, b) makes neither of them diamond-violators. Thus, we must have $v = v'$. □

Based on this result, we show that the children of node N and their associated data structures can be generated in $\mathcal{O}(k_N)$ time from the data structures of N.

Applying a branching rules to create children. We first show how to select and apply a branching rule at a non-leaf search tree node N in $\mathcal{O}(k_N)$ time. If S_N is not empty, then H_N and π violate ($\star$). In this case, we select a vertex v from S_N with the minimum neighbor x in π. Unfortunately, selecting v could require $\Omega(|H_N|)$ time if we are not careful. We bound the search time by $\mathcal{O}(k_N)$ by representing H_N so that the adjacency lists of the vertices in B are ordering according to π. Creating and maintaining these ordered lists uses only $\mathcal{O}(|G|)$ time during the initialization phase of the algorithm. During the the search tree construction and traversal times, we need not concern ourselves with these orderings because edge removals only remove items from adjacency lists. Given that the adjacency lists are ordered according to π, and that $|S_N| \in \mathcal{O}(k_N)$ by Lemma 6.13, it is possible to select v and x in $\mathcal{O}(k_N)$ time. Vertex y is then the last vertex in the adjacency list of v. Next we select vertex a between x and y that is adjacent to a vertex $b \neq v$. Unfortunately, if v is adjacent to many leaves or there are vertices with degree equal to zero, then this could take $\Omega(|A|)$ time. We handle degree zero vertices by simply removing them from the graph during initialization or whenever they are created by an edge removal. Thus, in the following, we will assume that there are no such vertices. We handle consecutive leaves incident on v using the following observation:

Observation 6.1 *Let $G = (A, B; E)$ be a bipartite graph and π a linear ordering of A. Let S be a biplanarizing set of G and π. If $v \in B$ has two leaf neighbors a_1 and a_2 that are consecutive in π and $(a_1, v) \notin S$, then $S - (a_2, v)$ is a biplanarizing set of G and π.*

In other words, a minimum biplanarizing set either contains both (a_1, v) and (a_2, v) or neither of them. Therefore, we can treat these leaf edges like a single edge whose removal has the cost of two edges. During the initialization phase, we simply combine consecutive leaves in π that are adjacent to the same vertex in B and mark them with an extra cost. During the course of the algorithm, we remove edges from the graph so additional leaves may be created. In these cases, we simply merge them with any other consecutive leaf edges that are incident on the same vertex and increase the cost of removing the resulting edge. In the following, then, we will assume that there are no consecutive leaves incident on the same vertex.

Based on these two simplifying assumptions, then, vertex a is the first or second vertex immediately following x in π (we recall that a is minimum). Vertex x', the maximum neighbor of v before a in π, and vertex y', the minimum neighbor of v after a in π, are also easy to find. Vertex x' is the first or second vertex immediately preceding a in π (we note that $x' = x$ or x' is a leaf by the minimality of a) and y' is the vertex immediately following x' in the adjacency list of v. Having selected these vertices, it is now easy to select and

apply an appropriate branching rule in constant time. Thus, we have shown how to select and apply a branching rule in $\mathcal{O}(k_N)$ time (recall that selecting v may use $\mathcal{O}(k_N)$ time) whenever H_N contains a star-violator.

For each set of edges generated by a branching rule, we create a child N' for N and generate a corresponding set of data structures.

Computing $H_{N'}$ for child N'. Unfortunately, subgraph $H_{N'}$ may be larger than $\mathcal{O}(k_{N'})$. Therefore, we cannot actually generate a completely new graph for the child. Instead, we construct and search the tree simultaneously in a depth-first search order, so that we actually only need to explicitly maintain one graph. To construct $H_{N'}$, we delete the necessary edges from H_N to obtain $H_{N'}$. Then, if in our search we later backtrack from N' up to N, we simply undo the changes made to obtain $H_{N'}$ in order to obtain H_N once again. This uses only constant time per edge removal.

Computing $S_{N'}$ for child N'. To obtain $S_{N'}$, we observe that $S_{N'} \subseteq S_N$ because $H_{N'}$ is a subgraph of H_N. Thus, we obtain $S_{N'}$ by determining which vertices of S_N are star-violators in $H_{N'}$. This requires only constant time per vertex in $S_{N'}$ using a technique similar to the one described above for selecting a branching rule. Since S_N has size at most $\mathcal{O}(k_N)$, then we can compute $S_{N'}$ in $\mathcal{O}(k_N)$ time.

Computing $T_{N'}$ for child N'. Just like $H_{N'}$, subgraph $T_{N'}$ may be larger than $\mathcal{O}(k_{N'})$, but, because we construct and traverse our search tree using a depth-first search order, we actually only need to maintain one copy of the subgraph at a time. We obtain $T_{N'}$ from T_N, then, by adding or removing vertices as necessary.

We add a vertex of $B - S_{N'}$ to T_N obtain $T_{N'}$ whenever it has degree equal to two in $H_{N'}$ and either belongs to S_N or has degree greater than two in H_N. We detect the first case during the construction of $S_{N'}$, each vertex of S_N not added to $S_{N'}$ is added to $T_{N'}$ if it has degree equal to two. We detect the second case during the construction of $H_{N'}$. For each edge that is removed from H_N to obtain $H_{N'}$, we check whether or not the degree of its end-vertex in B has been reduced to two.

We remove a vertex from T_N to obtain $T_{N'}$ whenever the vertex has degree less than one in $H_{N'}$. We do not need to concern ourselves with vertices of degree two in T_N that belong to $S_{N'}$ because, as we observed above, $S_{N'} \subseteq S_N$. Thus, we check for vertices to remove from T_N during the construction of $H_{N'}$. For each edge that is removed from H_N to obtain $H_{N'}$, we check whether or not the degree of its end-vertex in B belongs to T_N and has had its degree reduced to less than two.

Computing $D_{N'}$ for child N'. The final data structure constructed is $D_{N'}$, the set of

diamond-violators that are not star-violators. To construct $D_{N'}$, then, we must be able to determine whether or not a given vertex is a diamond-violator but not a star-violator. Unfortunately, this could require searching through each vertex in B with degree equal to two in $H_{N'}$ if we are not careful. We can perform such a test in constant time if modify the representation of $T_{N'}$ so that each vertex u in A has two adjacency lists, one list for degree two neighbors in B whose other neighbor is before u in π, and another list for degree two neighbors in B whose other neighbor is after u in π. Representing $T_{N'}$ in this way and maintaining this representation for each node N' does not increase our asymptotic running time.

Given this representation, then, we perform our test on vertex $w \in B$ as follows: w belongs to $D_{N'}$ if and only if w belongs to an adjacency list in $T_{N'}$ containing more than one vertex. The correctness of this test follows by definition. If w does not belong to $T_{N'}$, then w does not satisfy the definition of $D_{N'}$ so we should reject w. Otherwise, w belongs to $T_{N'}$ so let u be one of its neighbors in A. If a vertex w' belongs to the same adjacency list of u that w belongs to, then we observe that w' has the same neighbors as w because, otherwise, w or w' would be star-violators. Therefore, w is a diamond-violator but not a star-violator so it belongs to $D_{N'}$. If, on the other hand, no other vertex belongs to the adjacency list of u that contains w, then no other vertex in B has exactly the same two neighbors as w. No vertex in $S_{N'}$ has the same two neighbors because, otherwise, w would be a star-violator. No vertex outside $S_{N'}$ has the same two neighbors because, otherwise, there would be another vertex in the adjacency list of u. Therefore, w is not a diamond-violator so it does not belong to $D_{N'}$.

We note that this test can be performed in constant-time, so, in $\mathcal{O}(k_N)$ time, it is possible determine which vertices of $S_N \cup D_N$ belong to $D_{N'}$. It remains, then, for us to find each vertex w that belongs to $D_{N'}$ but does not belong to S_N or D_N. If w does not belong to T_N, then w has degree greater than two in H_N and, by the test above, is added to an adjacency list in $T_{N'}$ that already contains at least one vertex. On the other hand, if w belongs to T_N, then, by the test described above, w belongs to adjacency lists of size equal to one in T_N, and to adjacency lists of size greater than one in $T_{N'}$. In each case, vertex w is the result of an edge removal. In the first case, w is an end-vertex and, in the second, an end-vertex is added to an adjacency list containing only w. Using the test above, we can determine in constant time per edge removal which vertices outside S_N and D_N belong to $D_{N'}$.

Handling leaf nodes. We have now described how to construct and traverse the search tree by spending at most $\mathcal{O}(k_N)$ time per at each non-leaf search tree node N. To complete the algorithm description, we describe how to detect leaf nodes and whether or not they correspond to yes-instances. By definition, a node N is a leaf if either $k_N = 0$ or $S_N = \emptyset$

and corresponds to a yes-instance if and only if $\mathsf{bpn}(H_N) \leq k_N$. Of course, if $k_N = 0$ but $S_N \neq \emptyset$, then the node corresponds to a no-instance because $\mathsf{bpn}(H_N) > 0$. On the other hand, if H_N satisfies $(\star)$, then, by Lemma 6.12, we must determine whether or not we can remove all diamond-violators from H_N by removing at most k_N edges. Because S_N is empty, then, by definition, D_N contains all diamond-violators and T_N contains all vertices of H_N with degree equal to two. From our test for membership in D_N described above, then, we remove diamond-violators by removing enough edges so that the adjacency lists of each vertex of A in T_N contains at most one vertex. Dujmović *et al.* [25] shows that we can remove all diamond-violators from H_N with a minimum number of edge-removals by repeatedly selecting a diamond-violator from D_N, removing an incident edge and then updating D_N and T_N. As described above, these updates require constant time per edge-removal and $|D_N| \leq 2k_N$ so we can test whether or not a leaf node corresponds to a yes-instance in $\mathcal{O}(k_N)$ time.

We call the resulting algorithm OLP. By Theorem 6.3, then, we have the following result:

Theorem 6.14 *For any bipartite graph $G = (A, B; E)$, linear ordering π of A and integer $k \geq 0$, algorithm* OLP *determines whether or not $\mathsf{bpn}(G, \pi) \leq k$ in $\mathcal{O}(2^k + |G|)$ time.*

6.3.2 2-LAYER PLANARIZATION in $\mathcal{O}(3.562^k + |G|)$ Time

For the 2-LAYER PLANARIZATION problem, it is possible to spend constant time at each each search tree node after $\mathcal{O}(|G|)$ time initialization. Our approach is very similar to the one we use in [90] to reduce the running time of the $(6^k \cdot |G|)$ time algorithm to $\mathcal{O}(6^k + |G|)$. In the algorithm for 1-LAYER PLANARIZATION, we noted that the size of each graph H_N associated with each search tree node may be larger than even $\mathcal{O}(k_N)$ so, rather than create a new graph at each node, we construct and traverse the tree simultaneously in a depth-first search order, maintaining only one explicit copy of the graph.

It is possible to choose an appropriate branching rule in constant time if, in addition to the graph, we maintain a list F of vertices with three or more non-leaf neighbors and, for each vertex v, the list $f(v)$ of edges incident on v that are incident to the non-leaf neighbors of v. We first select any vertex v in F, then we select the first three edges e_1, e_2 and e_3 in $f(v)$, sorted so that $\deg(v_1) \geq \deg(v_2) \geq \deg(v_3)$, and finally, for each v_i, we select an incident edge other than e_i. It is then easy to select an appropriate branching rule in constant time. Of course, if F is empty, then, we are at a leaf node in the tree.

We now show how to update set F and mapping f appropriately as we move from a parent to a child node during the traversal of the search tree. By Lemma 6.5, a vertex v

must be added to or removed from F or the value of $f(v)$ modified, if it is the end-vertex of a removed edge or if it is adjacent to an end-vertex of a removed edge with degree equal to two. Thus, we need only update F and f with respect to at most four vertices for each edge removal, the end-vertices of the removed edge and, if an end-vertex becomes a leaf, then its only remaining neighbor.

As mentioned in the simple $\mathcal{O}(6^k \cdot |G|)$ time algorithm, a leaf node N in the search tree corresponds to a yes-instance if H_N contains contains no 2-claws and at most k_N component wreaths. Unfortunately, detecting component wreaths in H_N could require more than constant time so we must find a way to avoid this. The solution is simply to detect and planarize component wreaths as soon as they are created by an edge removal. While this creates extra work at each search tree node, it is still possible to perform the extra work in constant time per edge removal.

To detect a component wreath in constant time, we cannot expect to do so by visiting each vertex in the component. Instead, we rely on pointers called *cheaters*. Cheaters link the first and last vertices on the spine of every internal caterpillar.

Suppose that the subgraph H_N of a node N contains no component wreaths but that, for some edge $e = (u, v)$ in H_N, $H_N - e$ contains at least one component wreath W. If $\deg(u) = 2$, then let u' be the neighbor of u not equal to v; otherwise, let $u' = u$. Similarly, if $\deg(v) = 2$, then let v' be the neighbor of v not equal to u; otherwise, let $v' = v$. By Lemma 6.5, u, v, u' and v' are the only vertices in H_N for which $\deg'_{H_N - e} < \deg'_{H_N}$. Each vertex in a component wreath has at most two non-leaf neighbors. Considering W as a subgraph of H_N, we have that either W consists of an internal caterpillar and its single connection point u' or v' to the rest of the graph, or else W belongs to a connected components of H_N consisting of W and the edge (u, v). We detect the first case by noticing that the internal caterpillar in H_N has a single connection point u' or v' and, following the removal of (u, v), the vertex is not longer a connection point. We detect the second case by noticing that two internal caterpillars of H_N have the same connection points, the connection points belong to $\{u, v, u', v'\}$, and, following the removal of (u, v), neither of these vertices are connection points. Thus, in either case, any component wreath created by an edge removal can be detected in constant time using cheaters and the end-vertices of the removed edge.

We now describe how to update cheaters in constant time following the removal of an edge e from subgraph H_N. If a new internal caterpillar is created by an edge removal, then $\deg'_{H_N}(v) > \deg'_{H_N - e}(v) = 2$ for some vertex v. If v is a connection point for two internal caterpillars P_1 and P_2 before the edge removal, then the new internal caterpillar is the concatenation of P_1 and P_2 along with v and its incident edges. If v is a connection point

for only one internal caterpillar P, then the new internal caterpillar is the concatenation of P with v and its leaf neighbors. Otherwise, the new internal caterpillar is composed only of v and its leaf neighbors. In each of these three cases, it is a simple matter to update the cheaters in constant time after an edge removal using existing cheaters. By Lemma 6.5, we need to consider these cases with respect at most two vertices, either the end-vertices of the removed edge, or, if one or both becomes a leaf, then its only neighbor.

Thus, we have shown how to construct and traverse the search tree by spending at most constant time at each search tree node. By Theorem 6.11, we have an algorithm which we call TLP that solve the 2-LAYER PLANARIZATION problem in $\mathcal{O}(3.562^k + |G|)$ time.

Theorem 6.15 *For any graph G and integer $k \geq 0$, algorithm* TLP *determines whether or not* $\mathsf{bpn}(G) \leq k$ *in* $\mathcal{O}(3.562^k + |G|)$ *time.*

6.4 Incorporating Divide-And-Conquer

One way that we might improve the performance of algorithm TLP on large sparse graphs is to integrate a divide-and-conquer approach into the bounded search tree. For example, if we biplanarize two subgraphs G_1 and G_2 as a single graph, then we could use up to $\mathcal{O}(3.562^{\mathsf{bpn}(G_1)+\mathsf{bpn}(G_2)} + |G_1| + |G_2|)$ time. If, on the other hand, it is possible to biplanarize them separately, then we would use at most $\mathcal{O}(3.562^{\mathsf{bpn}(G_1)} + 3.562^{\mathsf{bpn}(G_2)} + |G_1| + |G_2|)$ time. Clearly, the second option is preferable. In sparse graphs, we would expect this to be possible quite often because the edge-removals will tend to disconnect the graph being planarized.

Certainly, if a graph is disconnected, then the minimum biplanarizing set for the whole graph is simply the union of the minimum biplanarizing sets for the connected components. We can, however, do slightly better than this by dividing the graph into p-components. A *p-component* of a graph is a maximal connected subgraph consisting of biconnected components that are connected by internal paths of length at most three, and any internal caterpillars that connect it to other p-components. We note, then, that two p-components are not necessarily disjoint since they may share a single internal caterpillar. The following lemma shows that each p-component can be planarized separately.

Lemma 6.16 *If $H_1, H_2, \ldots, H_p$ are the p-components of a graph G, and $M_1, M_2, \ldots, M_p$ are their minimum canonical biplanarizing sets, respectively, then $\bigcup M_i$ is a minimum canonical biplanarizing set for G and $M_i \cap M_j = \emptyset$ for each $i \neq j$.*

Proof: We first show that $\bigcup M_i$ is a biplanarizing set for G, so consider removing all edges in $\bigcup M_i$ from G. The resulting graph G' contains no component wreaths because

each wreath cycle belongs entirely to a single p-component. In addition, each vertex has $\deg' \leq 2$ because each vertex with $\deg' \geq 3$ in G belongs to a p-component. Thus, by Lemma 2.1, subgraph G' is biplanar. Furthermore, any edge that is candidate in H_i is also a candidate in G; therefore, $\bigcup M_i$ is also a canonical biplanarizing set for G.

Next we show that $M_i \cap M_j = \emptyset$ for $i \neq j$. We recall that H_i and H_j share at most one internal caterpillar or path in G. Only two edges on this caterpillar are candidate edges, and, of these two, one is incident on a leaf in H_i and the other is incident on a leaf in H_j. Therefore, the candidate edges of H_i are disjoint from those in H_j so $M_i \cap M_j = \emptyset$.

Finally, we show that $\bigcup M_i$ is a minimum biplanarizing set G. Let M' be a minimum canonical biplanarizing set for G. Let M'_i be the candidate edges in H_i that belong to M. Since M'_i is a biplanarizing set for H_i, we have $|M'_i| \geq |M_i|$. Thus, $|M'| \geq |M_1| + \ldots + |M_p|$. □

Lemma 6.16 suggests a divide-and-conquer variation of the algorithm: divide the graph into p-components, and then planarize each p-component individually. In fact, we could actually do better than this by biplanarizing in such a way as to break larger p-components into smaller p-components so that we can planarize each of them separately. One possible strategy for breaking up a p-component is to branch on forbidden structures containing cut vertices.

A slight complication of this variation of the algorithm is that, when planarizing a p-component, we are using a bounded search tree so we have bounded the number of edge removals by some parameter k. Thus, if we break the p-component C into smaller child p-components, then we must somehow divide the parameter k for C into smaller parameters for each child p-component. The problem is that we do not know which parameter value to assign each child without knowing the size of its minimum biplanarizing set. We solve this problem by initializing the parameter for each child to some lower bound on the child. We re-apply the algorithm to the child, increasing its parameter until we find its minimum biplanarizing set. If the sum of the parameters for the children ever becomes greater than the parameter for their parent, then we realize that the way we divided the parent p-component into smaller p-components will not yield a biplanarizing set matching the parent's parameter. In response, we immediately backtrack to the point in the search tree where we disconnected the parent p-component and continue traversing the search tree from there.

The extra work of computing the p-components and determining if the current p-component C has been broken into smaller p-components can all be done in $\mathcal{O}(\mathsf{bpn}(C))$ time. To compute sub-p-components, we simply apply a modification of the algorithm for finding biconnected components in a graph. We apply the algorithm to the at most $\mathcal{O}(\mathsf{bpn}(C))$

vertices having three or more non-leaf neighbors, skipping over internal caterpillars during graph traversal using the cheaters, which we described earlier.

Despite the extra work at each node, the resulting algorithm still runs in $\mathcal{O}(3.562^k + |G|)$ time. In fact, we will show that we can apply this divide-and-conquer approach to any bounded search tree algorithm with running time $\mathcal{O}(\alpha^k + |G|)$ for some constant $\alpha \geq 3$ to obtain a divide-and-conquer algorithm with the same asymptotic running time. Let $T(k)$ be the running time of our algorithm when applied to a graph with bpn $= k$ after preprocessing has been completed. It has the following upper bound:

$$T(k) \leq T'(0) + T'(1) + \ldots T'(k) \text{ for } k \geq 0$$

where $T'(k)$ is the time it takes to determine if a graph has a biplanarizing set of size k. Let c be a constant of sufficiently large size such that we use at most ck time to compute and check for new p-components after each edge removal. Then, $T'(k)$ can be described as follows:

$$T'(k) \leq \alpha \cdot \begin{cases} T(j_1) + T(j_2) \ldots + T(j_i) + ck \\ \qquad \text{if the edge removal disconnects the p-component into} \\ \qquad \text{children with bpn's equal to } j_1, j_2, \ldots, j_i \text{ such that} \\ \qquad j_1 + j_2 + \ldots + j_i \leq k-1 \text{ and each } j_i \geq 1; \\ T'(k-1) + ck \text{ otherwise.} \end{cases}$$

We prove that $T'(k)$ is in $\mathcal{O}(\alpha^k)$ by induction on k by showing that $T'(k) \leq c'\alpha^k$ for a constant c' large enough that $T'(k) \leq c'$ for $k = 0..1$. For $k \geq 2$, then, we have for the first case:

$$\begin{aligned} T'(k) &\leq \alpha T(1) + \alpha T(k-2) + \alpha ck \\ &\leq \alpha[T'(0) + T'(1)] + \alpha[T'(0) + T'(1) + \ldots + T'(k-2)] + \alpha ck \\ &\leq \alpha 2c' + \alpha[2c' + c'\alpha^2 + c'\alpha^3 + \ldots + c'\alpha^{k-2}] + \alpha ck \\ &= \alpha c'(\alpha^2 + \alpha^3 + \ldots + \alpha^{k-2}) + \alpha 4c' + \alpha ck \\ &= \alpha c'(\alpha^{k-1} - \alpha^2)/(\alpha - 1) + \alpha 4c' + \alpha ck \\ &= c'\alpha^k - c'\alpha^k(\alpha - 2)/(\alpha - 1) - c'\alpha^3/(\alpha - 1) + \alpha 4c' + \alpha ck \\ &\leq c'\alpha^k \text{ for } c \leq c'. \end{aligned}$$

For the second case of $T'(k)$, we similarly have:

$$\begin{aligned} T'(k) &\leq \alpha T'(k-1) + ck \\ &\leq c'\alpha^{k-1} + ck \\ &\leq c'\alpha^k - c'(\alpha-1)\cdot\alpha^{k-1} + ck \\ &\leq c'\alpha^k \text{ for } c \leq c'. \end{aligned}$$

Thus, $T(k) \leq c'\alpha^0 + c'\alpha^1 + c'\alpha^2 + \ldots + c'\alpha^k \leq c'\alpha^{k+1}$, so $T(k)$ is $\mathcal{O}(\alpha^k)$. Together with the preprocessing step, we have an algorithm that runs in $\mathcal{O}(\alpha^k + |G|)$ time. For algorithms with $1 < \alpha < 3$, we cannot quite obtain the results mentioned above. Instead, we simply observe that we spend $\mathcal{O}(k)$ time at each search tree node so the resulting divide-and-conquer algorithm has a running time of $\left(k \cdot \alpha^k + |G|\right)$.

Though this algorithm does not immediately help us to obtain theoretical running time improvements, as we will see in Chapter 8, it does demonstrate dramatically improved experimental performance.

6.5 Conclusions and Future Directions

In this chapter, we have derived fixed-parameter tractable algorithms for both the 1-LAYER PLANARIZATION and 2-LAYER PLANARIZATION problems. Both algorithms significantly improve on the best running times of previous algorithms by combining simple heuristics for biplanarizing sparse and dense subgraphs. We believe, however, that there is still room for improvement because our algorithms do not make use of everything that is known about biplanarization using bounded search trees. For example, the running time improvements of Fernau [39] require sophisticated techniques for analyzing recurrences that describe the size of the bounded search trees. In this chapter, we consider only very simple recurrences and use elementary analysis techniques.

Another possibility for improvement could come from our divide-and-conquer approach. As we show in Chapter 8, this approach results in the most successful implementation discussed in that chapter. We would like to explain its success mathematically. One possibility would be to prove that, given a p-component C and constants $0 < \alpha, \beta < 1$, it is possible biplanarize C in the search tree in such a way that, after removing $\alpha\mathsf{bpn}(C)$ edges from C, we will have broken C into two smaller p-components with biplanarizing sets of size at most $\beta(1-\alpha)\mathsf{bpn}(C)$ each. If $T(k)$ describes the size of the resulting search tree, we

obtain the following upper bound on $T(k)$ (assuming $T(1) = 1$):

$$\begin{aligned} T(k) &\leq 2 \cdot 3.562^{\alpha k} T(\beta(1-\alpha)k) \\ &\leq 2^j \cdot 3.562^{\alpha k(\sum_{i=0}^{j-1}(1-\alpha)^i\beta^i)} T(\beta(1-\alpha)^j k) \\ &\leq 2^{-\log(k)/\log((1-\alpha)\beta)} 3.562^{k\alpha/(1-(1-\alpha)\beta)} \end{aligned}$$

For example, if we have $\alpha = \frac{1}{2}$ and $\beta = \frac{3}{4}$, then the tree has size at most $k^{0.71}3.562^{4k/5} \approx k^{0.71}2.77^k$.

A final possibility for improvement is based on *candidate* and *ambivalent* edges. As defined in [25], an edge $e = (u, v)$ is called a *candidate* for removal if it is the middle edge of an internal 3-path, or it does not belong to an internal 3-path and $\deg'(u) > 2$ and $\deg(v) > 1$. We use $\mathcal{K}$ to denote the set of candidate edges, so a *canonical biplanarizing set* is a biplanarizing set that is a subset of $\mathcal{K}$. The following lemma shows that there is always a minimum biplanarizing set that is canonical.

Lemma 6.17 (Dujmović *et al.* [25]) *If T is a biplanarizing set for a graph G, then there exists a canonical biplanarizing set T^* of G such that $|T^*| \leq |T|$.*

One can easily test whether or not an edge is a candidate for removal in constant time, so it is possible to modify any biplanarizing algorithm to produce only canonical biplanarizing sets without affecting its worst-case performance.

Similarly, we say that an edge e is *ambivalent* with respect to an end-vertex v if:

- $\deg'(v) \geq 3$;
- e is a candidate edge;
- e is a bridge edge (see Chapter 2);
- e belongs to an internal path of length at least four or to an internal caterpillar that is not an internal path.

Unlike non-candidate edges, we cannot simply ignore all ambivalent edges. However, as the following lemma shows, we can ignore some and thus prune some of the branches in our bounded search tree:

Lemma 6.18 *Let e be an ambivalent edge with respect its end-vertex v in a graph G, and let T be a minimum canonical biplanarizing set for G containing e. Then, there exists another edge e' incident on v but not in T such that $(T - e) + e'$ is a minimum canonical biplanarizing set for G.*

Proof: Let $G' = G \setminus (T - e)$. Because e is ambivalent, at most one vertex in G' has three non-leaf neighbors and all other vertices have at most two non-leaf neighbors. Because T is minimum, either v or a neighbor of v has three non-leaf neighbors in G' while all other vertices have at most two non-leaf neighbors. In either case, we can biplanarize G' by removing any non-leaf edge $e' \neq e$ incident on v. □

For example, then, if an edge is ambivalent with respect to the root of a 2-claw, then we normally do not need to consider branching on its removal from the 2-claw. The only exception is if each edge in the 2-claw is either a non-candidate or an ambivalent edge. In that case, we arbitrarily choose one of the ambivalent edges incident on the 2-claw root and remove it.

While candidate edges and canonical biplanarizing sets have already been considered, ambivalent edges are new, and we would like to know if there is a way to use them to further decrease the size of our bounded search trees.

Chapter 7

One-Sided Crossing Minimization Algorithms and Applications

To Vida Dujmović—*you remind me that life is painted with brilliant colors.*

In this chapter, we describe algorithms for solving the 1-SIDED CROSSING MINIMIZATION problem:

> **Given:** A bipartite graph $G = (A, B; E)$, an integer $k \geq 0$, and a linear ordering π_A of A.
> **Question:** Is $\mathsf{bcr}(G, \pi_A) \leq k$?

The algorithm we describe was discovered by Dujmović, Fernau and Kaufmann [26] and currently has the best asymptotic running time of $\mathcal{O}(1.4656^k + k \cdot |G|^2)$. It is based on the bounded search tree algorithms for crossing minimization described in Section 1.3.5.1. Though we are not the authors of this algorithm, we include it in this thesis because, in Chapter 8, we describe our implementation of and experiments with this algorithm. Hence, our description of the algorithm here is more detailed than usual (e.g. than in [26]) in order that our resulting implementation of it is fairly obvious.

We recall that, in Section 6.4 of the previous chapter, we described a divide-and-conquer version of our $\mathcal{FPT}$ algorithm for biplanarizing graphs. Similarly, in this chapter, we show apply this idea to crossing minimization to obtain a divide-and-conquer version of the algorithm of [26]. In Chapter 8, we will present experimental results using an implementation of this version as well.

Finally, at the end of this chapter, we describe two applications closely related to the 1-SIDED CROSSING MINIMIZATION problem. We show, for the first time, that both applications are $\mathcal{NP}$-complete and explain how to modify the algorithms presented in this chapter to obtain efficient solutions for these applications. These modified algorithms show that, unlike most exact solutions to hard problems, the bounded search tree algorithms described in this chapter are quite flexible.

7.1 Preliminaries

Let $G = (A, B; E)$ be a bipartite graph. Given a linear ordering π_A of A and a linear ordering π_B of B, consider any 2-layer drawing of G in which the vertices of A are drawn on one layer in the ordering given by π_A, and the vertices of B are drawn on the other layer in the ordering given by π_B. We compute the number of edge crossings in this drawing as follows:

$$\sum_{u <_{\pi_B} v,\ u,v \in B} c_{uv}$$

where c_{uv} denotes the number of edge crossings in the drawing between edges incident on u and v. If we were to reverse the order of u and v in the drawing, then above calculation for the number of edge crossings in the new drawing would contain the value of c_{vu} instead of c_{uv} because v is before u. We observe that the values of c_{uv} and c_{vu} are the same regardless of the ordering defined by π_B. We call c_{uv} and c_{vu} the *crossing numbers* of u and v with respect to π_A.

We will define the algorithm in terms of crossing numbers, so the first step of the algorithm is to actually compute them for each pair of vertices. It is not too difficult to obtain a brute-force algorithm for this step that runs in $\mathcal{O}(|G|^3)$ time. However, as described in [26], crossing numbers can all be computed more efficiently as follows:

1. Sort the adjacency lists of the vertices in B according to π_A.
2. For each vertex v in B:
   (a) For each vertex u in A, compute the number $r(u)$ of neighbors of v that are before u in π_A.
   (b) For each vertex v' in $B - v$, we compute $c_{vv'}$ as follows:

$$c_{vv'} = \sum_{(u',v') \in E} r(u')$$

Sorting adjacency lists is the key to obtaining efficiency, and it can be completed in $\mathcal{O}(|E|)$ time using a bucket sort. With sorted adjacency lists, we are able to compute $r(u)$ for each vertex u in A in $\mathcal{O}(|A|)$ time provided that we iterate through the vertices of A in the order of π_A. Finally, we compute $c_{vv'}$ for each vertex v' in B in $\mathcal{O}(|B| + |E|)$ time. This can be improved to $\mathcal{O}(k|B|)$ by observing that we will not be using crossing numbers greater than k where k is crossing bound of the 1-SIDED CROSSING MINIMIZATION problem instance. Therefore, if we iterate through each neighbor u' of v' in A in their sorted order and stop as soon as we realize that $c_{vv'}$ is larger than k or that $r(u') = 0$, then we spend $\mathcal{O}(k)$ computing

$c_{vv'}$. In cases where we do not fully compute $c_{vv'}$, we just set its value to $k+1$. Thus, in total we use $\mathcal{O}(|E|+|B|(|A|+\min(k|B|,|B|+|E|))) = \mathcal{O}(\min(|A||B|+k|B|^2, |B|^2+|B||E|))$ time. To simplify, we'll say we use $\mathcal{O}(k|G|^2)$ time to compute crossing numbers.

7.2 Basic Bounded Search Tree Algorithm

As described in Section 1.3.5.1, the key observation in the design of bounded search tree algorithms for one-sided crossing minimization has to do with natural vertex orderings. A pair of vertices u and v in B are said to have a *natural ordering* in which u is before v whenever $c_{uv} = 0$, and a pair with a natural ordering are said to be *suited*. Dujmović and Whitesides [28] prove that optimal solutions to 1-SIDED CROSSING MINIMIZATION place suited pairs in their natural orderings:

Lemma 7.1 (Dujmović and Whitesides [28]) *Let $G = (A, B; E)$ be a bipartite graph and π_A a linear ordering of A. Let Γ be a 2-layer drawing of G in which the vertices of A are drawn on one layer in the order given by π_A. If Γ contains at most $\mathsf{bcr}(G, \pi_A)$ edge crossings, then all suited pairs of vertices in B appear in their natural ordering in Γ.*

7.2.1 Ordering Suited Pairs

With this result, the next step in the algorithm, after computing crossing numbers, is to fix the order of all suited pairs of vertices in B in their natural orderings. At this step, we must be careful when ordering suited pairs with two natural orderings so that the resulting ordering, denoted π_1, is a valid partial ordering on B. One way to ensure a partial ordering is to break these "ties" according to some arbitrary linear ordering on the vertices in B. With Lemma 7.1 and this "tie-breaking" method, we guarantee that π_1 is a partial order on B.

The running time for this step in the algorithm is $\mathcal{O}(|B|^2)$.

7.2.2 Computing and Updating the Lower Bound

In the next step, we compute a lower bound on the number of crossings in any optimal solution. A solution assigns an order to each pair of vertices in B, so any optimal solution will contain at least the following number of edge crossings:

$$\sum_{u,v \in B,\ u \neq v} \min(c_{uv}, c_{vu}).$$

As the algorithm progresses, this lower bound may need to be updated if a pair, say u and v, are ordered so that they contribute $\max(c_{uv}, c_{vu})$ crossings rather than $\min(c_{uv}, c_{vu})$ crossings. When that happens, we do not need to recalculate the lower bound L. Instead, we simply add $\max(c_{uv}, c_{vu}) - \min(c_{uv}, c_{vu})$ to the previous lower bound.

Initially computing the lower bound can be accomplished in $\mathcal{O}(|B|^2)$ time. Thereafter, we use constant time per update.

7.2.3 Kernelizing

We kernelize our problem input by first comparing our lower bound L to k. Clearly, if L is larger than k, then our problem input is a no-instance to the problem so we return this negative answer; otherwise, we continue with $L \leq k$. Because we have already ordered each suited pair of vertices, the remaining pairs u and v are such that $\min(c_{uv}, c_{vu}) \geq 1$. Since $L \leq k$, we have at most k unordered pairs of vertices in B.

Next, we look for pairs of vertices u and v for which c_{uv} is greater than k. Because the drawing cannot have more than k edge crossings, we are forced to fix their order so that v is before u.

After fixing these pairs given the initial partial ordering π_1 from the previous step, we must obtain transitive closure so that the resulting ordering π_2 is a partial order. To do this, we first compute the list of unordered pairs of vertices U which, as mentioned earlier, contains at most k pairs. Then, each time we assign an order to a pair u and v, say u before v, we compute the transitive closure for the current ordering π by iterating through U exactly one time. For each pair of vertices u' and v' in U, we apply two tests:

1. $u' \leq_\pi u$ and $v \leq_\pi v'$; and
2. $v' \leq_\pi u$ and $v \leq_\pi u'$.

If the first test holds, then we have by transitivity that $u' < v'$. Conversely, if the second test holds, then we have that $v' < u'$. If either test holds, then we remove the pair u' and v' from U and update π accordingly; otherwise, if neither test holds, then we just leave the pair in U and π remains the same.

After all such pairs have been assigned an order, the resulting order π_2 is a partial order on B and U is the set of pairs that are not ordered in π_2.

This step uses $\mathcal{O}(k^2)$ time because we assign an order to at most k pairs and spend $\mathcal{O}(k)$ time obtaining transitive closure after assigning an order to a pair.

7.2.4 Traversing the Bounded Search Tree

At this point we are ready to construct and traverse our bounded search tree. We describe this procedure in a little more detail than we did in Section 1.3.5.1.

To each node N in the search tree, we associate a partial ordering π_N on the vertices in B, a list U_N of pairs of vertices in B that are not ordered π_N, and a lower bound L_N on the number of crossings in any solution that agrees with π_N. At the root node, U_N is the final set of unordered pairs remaining after the kernelization step and $\pi_N = \pi_2$.

If the lower bound L_N at a node is not greater than k, then we create children for the node by selecting a pair of vertices u and v from U_N and creating two children, one for each way to assign an order to u and v. The vertex ordering and set of unordered vertex pairs at each child is obtained by adding the corresponding ordering of u and v to π_N, removing the pair u and v from U_N, and then obtaining transitive closure for the resulting ordering using the same method that we used in the kernelizing step. The lower bound for each child is obtained by updating L_N as described in the kernelizing step.

The size $T(k)$ of the resulting search tree is bounded as follows:

$$T(k) \leq 2T(k-1).$$

For small values of k, $T(k)$ is bounded by a constant so the size of the tree is $\mathcal{O}(2^k)$. Since the size of U_N is at most k_N, the time spent at each search tree node N is bounded by $\mathcal{O}(k_N)$. Thus, the overall running time of this algorithm, including the initialization time, is $\mathcal{O}(2^k + k \cdot |G|^2)$.

7.3 Improving to $\mathcal{O}(k^2 \cdot 1.618^k + k \cdot |G|^2)$ Time

As hinted at in Section 1.3.5.1, we can improve the running time of the previous algorithm by observing that we need never consider branching on unordered pairs u and v for which $c_{uv} = c_{vu} = 1$. We simply branch on pairs with $c_{uv} + c_{uv} > 2$ until there are none left. At a node N where each unordered pair has $c_{uv} + c_{vu} \leq 2$, each pair is also unsuited so $c_{uv} = c_{vu} = 1$. As a result, every solution containing π_N has L_N edge crossings, our lower bound; therefore, we just arbitrarily assign an order to each remaining pair. The size $T(k)$ of the resulting search tree is then bounded as follows:

$$T(k) \leq T(k-1) + T(k-2).$$

An easy proof by induction on k shows that $T(k) \leq \alpha^k$ for $\alpha = \frac{1+\sqrt{5}}{2} \approx 1.618$.

7.4 Improving to $\mathcal{O}(1.4656^k + k \cdot |G|^2)$ Time

We can further improve our running time by observing that not only can we avoid branching on cases where $c_{uv} + c_{vu} \leq 2$, but also on many cases where $c_{uv} + c_{vu} = 3$.

Lemma 7.2 (Dujmović, Fernau and Kaufmann [26]) *Let $G = (A, B; E)$ be a bipartite graph and π_A a linear ordering of A. Let Γ be a 2-layer drawing of G in which the vertices of A are drawn on one layer in the order given by π_A, and let u and v be a pair of vertices in B such that $\deg(u) = \deg(v) = 2$, $c_{uv} = 1$ and $c_{vu} = 2$. If Γ contains at most $\mathsf{bcr}(G, \pi_A)$ crossings, then u is left of v in Γ.*

Therefore, immediately after assigning orderings to suited pairs in the previous algorithm, we insert a new step in which we assign the order implied by Lemma 7.2 to each applicable pair of unordered vertices.

Then, in the bounded search tree, we first only consider unordered pairs u and v for which $c_{uv} + c_{vu} \geq 4$. If none of these are left at a node N, then we immediately apply two new kernelizing steps. In the first, we arbitrarily assign an order to pairs of vertices that have the same set of neighbors. We note that this step could have been applied at the beginning of the algorithm. In the second kernelizing step, we find pairs of unordered vertices that have been assigned an order with respect to every other vertex. These pairs are said to be *independent* because any order we assign to them cannot transitively trigger an ordering of other unordered pairs. Consequently, we assign an ordering to the pair that minimizes the number of resulting edge crossings. In particular, if u and v are an independent pair, then we order them u before v if $c_{uv} < c_{vu}$, and, otherwise, we order them v before u.

Lemma 7.3 (Dujmović, Fernau and Kaufmann [26]) *Let N be a search tree node with two children C_1 and C_2. Let π_N be the partial order at node N such that each pair of unordered vertices u and v has $c_{uv} + c_{vu} \leq 3$, u and v do not constitute an independent pair, and u and v do not share the same set of neighbors. If an unordered pair of vertices x and y are such that $c_{xy} = 1$ and $c_{yx} = 2$, then $(k_N - k_{C_1}) + (k_N - k_{C_2}) \geq 4$.*

As in the previous algorithm, at nodes where there are no unordered pairs with $c_{uv} + c_{vu} = 3$, each pair has $c_{uv} = c_{vu} = 1$ so we arbitrarily assign an order to all the remaining unordered pairs. According to Lemma 7.3, then, the size $T(k)$ of our search tree tree is bounded as follows:

$$T(k) \leq \max(T(k-1) + T(k-3), 2T(k-2))$$

An easy proof by induction on k shows that $T(k) \leq \alpha^k$ for $\alpha \approx 1.4656$. We call the resulting algorithm OSCM.

Theorem 7.4 (Dujmović, Fernau and Kaufmann [26]) *Given any bipartite graph $G = (A, B; E)$, an integer $k \geq 0$, and a linear ordering π_A on A, algorithm* OSCM *determines in $\mathcal{O}(1.4656^k + k \cdot |G|)$ time whether or not* bcr$(G, \pi_A) \leq k$.

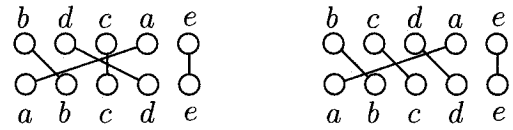
7.5 A Divide-And-Conquer Heuristic

In the previous section describing algorithm OSCM, we defined independent pairs of vertices in terms of a partial order π of the vertices in B. Such pairs are ordered in π with respect to all other vertices in B, but the pair itself is not ordered. As a result, we can assign an order to this pair without any consideration of the other vertices in B. In other words, we find a globally optimal solution by finding a locally optimal solution.

Generalizing, we consider a maximal subset $B' \subseteq B$ such that each pair of vertices consisting of one vertex in B' and one outside B is ordered in π. Then, any globally optimal solution contains an optimal solution for B'. In graph-theoretic terms, let G_π be the graph with $V(G_\pi) = B$ such that there is an edge $(u, v) \in E(G_\pi)$ whenever u and v are not ordered in π. Then, B' corresponds to a connected component in G_π, so we incorporate the divide-and-conquer approach described in Section 6.4 based on components of G_π rather than on p-components. Using the same running time analysis, we find that the resulting algorithm runs in $\mathcal{O}(k \cdot 1.4656^k + k \cdot |G|^2)$. We note that the term $k \cdot 1.4656^k$ cannot be replaced with 1.4656^k because $1.4656 < 3$. We call this algorithm OSCM-C and refer to it by this name in Chapter 8.

7.6 Two Applications of Crossing Minimization

As we mentioned in the introduction to this chapter, we describe two applications for one-sided crossing minimization, prove that the resulting problems are $\mathcal{NP}$-complete and show how to modify algorithm OSCM to obtain solutions to these problems. Both applications involve computing a "distance" between the linear orderings π_1 and π_2 of two subsets X_1 and X_2, respectively, of a common set of elements X. One natural way to measure distance is in terms of a 2-layer graph drawing: draw the elements of X_1 as vertices on one layer so that their order corresponds to π_1, draw the elements of X_2 as vertices on the other layer so that their order corresponds to π_2, and then draw an edge (line segment) between vertices that correspond to the same element in X. The "distance" between π_1 and π_2, then, is the number of edge crossings in the drawing (see, e.g. Figure 7.1). We can think of this distance measure as the number of pairs in X whose order in π_1 is different from their order in π_2.



$\text{diff}(bdcae, abcde) = 4$ $\text{diff}(bcdae, abcde) = 3$

Figure 7.1: The distance of two lists from the list $abcde$.

We will use $\text{diff}(\pi_1, \pi_2)$ to denote this distance between π_1 and π_2. We recall from Section 1.2 (see e.g. [32]) that $\text{diff}(\pi_1, \pi_2)$ can be computed in polynomial time.

7.6.1 Application for Phylogenetic Trees

The first application comes from the study of phylogenetic trees. Phylogenetic trees are graphs used to model the relationships between various species or other entities under the assumption that each pair of entities has a common ancestor. The Tree of Life is an example of a phylogenetic tree showing the evolutionary relationships between organisms. In this tree, each leaf corresponds to a currently living organism. The remaining vertices correspond to hypothetical organisms thought to be ancestors of the living organisms.

Normally, there are several different methods for computing a phylogenetic tree from a given set of entities and each produces a different tree. In addition, no specific tree captures entity relationships "better" than the other trees, so biologists are forced to compare two or more phylogenetic trees. The Tree of Life is no exception; its structure, including the relationships of the ancestors common to most living organisms, is highly controversial.

As described by Dwyer and Shreiber [29], one way to compare two different phylogenetic trees is to determine the minimum "distance" between the leaf vertex orderings admitted by each respective tree. A tree *admits* a given *leaf ordering* if there exists a corresponding set of orderings on the children of each vertex. More formally, let u and v be leaves in the tree with lowest common ancestor a, and let u_a be the child of a that is the ancestor of u and v_a be the child of a that is the ancestor of v. Then, u is before v in the admitted leaf ordering if and only if u_a is before v_a in the corresponding child ordering of a. Figure 7.2 shows each of the leaf orderings admitted by a small binary tree.

As described in the introduction to this section, we define the distance between two leaf orderings π_1 and π_2 as the value of $\text{diff}(\pi_1, \pi_2)$ (note: we compute diff as if π_1 and π_2 are not leaf orderings but actually the corresponding orderings of the entities such as genes that the leaves represent). Hence, the distance between two trees T_1 and T_2 is the distance between the pair of admitted leaf orderings of T_1 and T_2 that have the minimum distance;

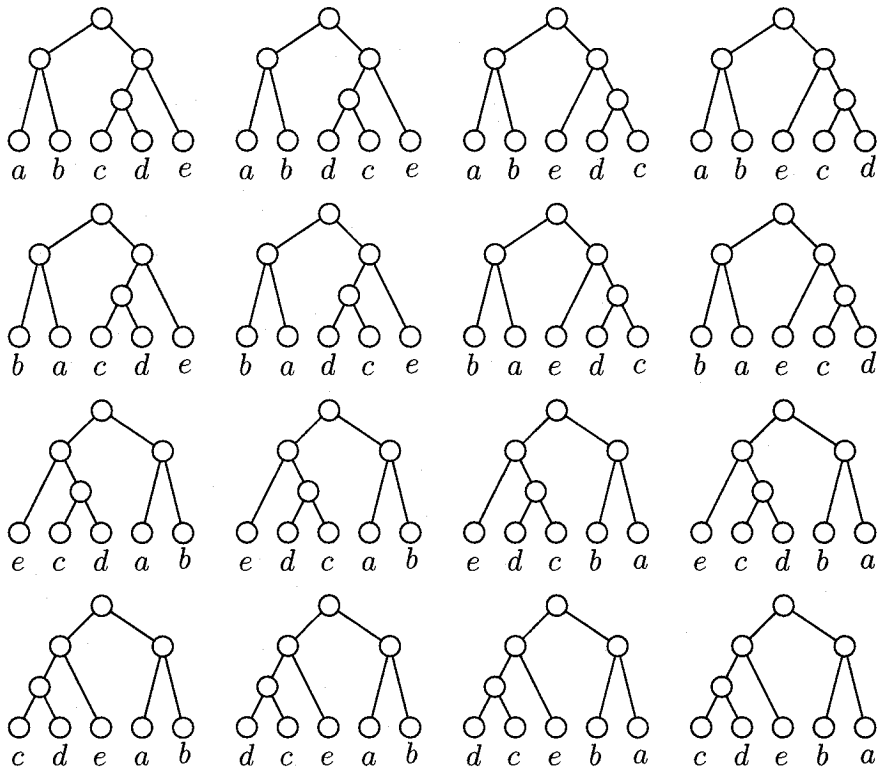


Figure 7.2: The leaf orderings admitted by a small binary tree.

that is, the distance between T_1 and T_2 is equal to:

$$\min\{\text{diff}(\pi_1, \pi_2) \mid \pi_1 \text{ is an admitted leaf ordering of } T_1, \pi_2 \text{ is an admitted leaf ordering of } T_2\}$$

We call the resulting decision problem 2-SIDED TREE COMPARISON:

> **Given:** An integer $k \geq 0$ and two trees T_1 and T_2 whose leaves are subsets of a common vertex set L.
> **Question:** Does T_1 admit a leaf ordering π_1 and T_2 admit a leaf ordering π_2 such that $\text{diff}(\pi_1, \pi_2) \leq k$?

We could also consider a "one-sided" version of this problem where the problem input is a tree and an ordering of the leaves. Solutions to this problem would be useful in situations where candidate phylogenetic trees are compared to some canonical leaf ordering, or several iterations of the solution are used to approximate a solution to the two-sided problem. We call this problem 1-SIDED TREE COMPARISON:

Given: An integer $k \geq 0$, a tree T, and a linear ordering π of the leaves of T.
Question: Does T admit a leaf ordering π' such that $\text{diff}(\pi, \pi') \leq k$?

Though these problems may seem easier than related crossing minimization problems, they are both $\mathcal{NP}$-complete. We describe a simple hardness proof based on ideas described by Prieto [81]:

Theorem 7.5 *Problems* 1-SIDED TREE COMPARISON *and* 2-SIDED TREE COMPARISON *are $\mathcal{NP}$-complete.*

Proof: Both problems clearly belong to $\mathcal{NP}$, so we immediately describe reductions from 1-SIDED CROSSING MINIMIZATION and 2-SIDED CROSSING MINIMIZATION, respectively.

Let bipartite graph $G = (A, B; E)$, linear ordering π_A of A and integer $k \geq 0$ be input to 1-SIDED CROSSING MINIMIZATION. Muñoz *et al.* [73] show that the problem remains $\mathcal{NP}$-complete even when the vertices in A are leaves so we will assume that A contains only leaves. Let T be a rooted tree composed of a root with one child for each vertex in B, and each of these children has one child for each incident edge in G. Let A' be the leaves of T. We observe that there is a 1-1 correspondence between A' and the edges of G and therefore the vertices of A because the vertices of A are leaves in G.

Given a 2-layer drawing Γ of G in which the vertices of A lie on one layer in the order given by π_A, we obtain an ordering π of the edges in G in the order that they intersect the layer containing the vertices of B. By definition, if we let π' be the ordering of A' corresponding to π, then, by definition, π' is admitted by T and Γ contains exactly $\text{diff}(\pi', \pi_A)$ edge crossings.

On the other hand, given a leaf ordering π' admitted by T, let π be a corresponding ordering of the edges of G. Since π' is admitted by T, edges incident on the same vertex in B are consecutive in π; therefore, there is a 2-layer drawing of G in which the vertices of A lie on one layer in the order given by π_A and the edges intersect a parallel line an arbitrarily small distance from the other layer in the order of π. This drawing has exactly $\text{diff}(\pi', \pi_A)$ edge crossings.

To show that 2-SIDED TREE COMPARISON, the reduction is identical and from 2-SIDED CROSSING MINIMIZATION. Here, we construct a tree for A using the same construction method as we use to construct the tree for B. □

We note that, for the case where the input trees are binary trees, trees in which each vertex has at most two children, Dwyer and Schreiber [29], give polynomial-time solutions to the 1-SIDED TREE COMPARISON problem.

For the general problem, however, we do not expect to find a polynomial-time solution to either of these problems, so we will be satisfied with finding efficient $\mathcal{FPT}$ algorithms instead. Indeed, it is not difficult to modify algorithm OSCM to solve the 1-SIDED TREE COMPARISON problem. In the modified versions of the algorithms, as soon as we assign an order to a pair of vertices u and v, we fix the order of each pair of vertices u' and v' where u' belongs to a subtree containing u but not v, and v' belongs to a subtree containing v but not u. We observe that none of these pairs will have previously been assigned an order so the resulting algorithm has the same running time as algorithm OSCM. Thus, we have an algorithm, which we call OSTC, for solving 1-SIDED TREE COMPARISON based on algorithm OSCM.

Theorem 7.6 *Given any integer $k \geq 0$, tree T, and linear ordering π on the leaves of T, algorithm* OSTC *determines in $\mathcal{O}(1.4656^k + k \cdot |G|^2)$ time whether or not there exists an admitted leaf ordering π' of T such that* $\text{diff}(\pi, \pi') \leq k$.

Unfortunately, we do not yet know of any efficient solutions for the 2-SIDED CROSSING MINIMIZATION problem so we cannot yet use a similar approach to obtain an efficient algorithm to the 2-SIDED TREE COMPARISON.

7.6.2 Application for Synthesizing Lists

High-throughput genomics and proteomic strategies, such as microarray studies, are often used to compare the importance of genes to some biological process. The result of these comparisons is usually a list of the genes whose order corresponds to the relative importance of each gene. As can be expected, different strategies give rise to similar though slightly different lists; therefore, it is necessary to combine the lists into a single list that somehow represents what is common to each of the strategies. One possible candidate is a list whose "distance" from the given lists is minimum. In other words, given two lists π_1 and π_2, we choose the list π such that $\text{diff}(\pi_1, \pi) + \text{diff}(\pi_2, \pi)$ is minimum possible. In Figure 7.3, we illustrate with the example where $\pi_1 = bcdae$ and $\pi_2 = bdcae$ and give the value of $\text{diff}(\pi_1, \pi) + \text{diff}(\pi_2, \pi)$ for two different values of π. We call the resulting decision problem LIST SYNTHESIS:

> **Given:** A set of elements X, $p \geq 1$ subsets $X_1, X_2, \ldots, X_p$ of X, a linear ordering π_i of each list X_i for $1 \leq i \leq p$, and an integer $k \geq 0$.
> **Question:** Is there linear ordering π of X such that the sum $\sum_{1 \leq i \leq p} \text{diff}(\pi, \pi_i) \leq k$?

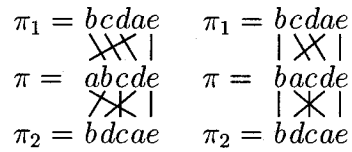


Figure 7.3: For $\pi = abcde$, we have $\text{diff}(\pi_1, \pi)+\text{diff}(\pi_2, \pi) = 3+4 = 7$ and, for $\pi = bacde$, we have $\text{diff}(\pi_1, \pi) + \text{diff}(\pi_2, \pi) = 2 + 3 = 5$.

As in our crossing minimization problem, we use c_{xy} to denote, for each pair of elements $x, y \in X$, the number of cases where $x >_{\pi_i} y$ for $1 \leq i \leq p$. We observe that this corresponds to the crossing minimization viewpoint where we have $p + 1$ layers and we place the elements of each X_i on their own layer in order given by π_i and then place the elements of X on layer $p + 1$. Adding an edge between each element on an X_i layer to the corresponding element on layer $p + 1$, our problem turns into a crossing minimization problem where we attempt to order the elements on layer $p + 1$ so that the number of edge crossings is minimized. In other words, we can view c_{xy} as the number of crossings created by edges incident on x and y on layer $p+1$ when x is before y on the layer. We also observe that we can reformulate the sum that we are attempting to minimize as follows:

$$\sum_{x,y \in X \mid x <_\pi y} c_{xy} = \sum_{1 \leq i \leq p} \text{diff}(\pi, \pi_i).$$

In addition, we can reuse the lower bound that we used for crossing minimization:

$$\sum_{x,y \in X} \min\{c_{xy}, c_{yx}\} \leq \sum_{1 \leq i \leq p} \text{diff}(\pi, \pi_i).$$

We prove $\mathcal{NP}$-completeness of this problem using the basic ideas used by Muñoz *et al.* [73] to show that 1-SIDED CROSSING MINIMIZATION is $\mathcal{NP}$-complete even when the vertices on the ordered layer are leaves and the vertices on the other layer have degree equal to four. The reduction is from the $\mathcal{NP}$-complete FEEDBACK ARC SET problem (see, e.g. [41]):

> **Given:** Given a directed graph $G = (V, E)$ and an integer $k \geq 0$.
> **Question:** Is there a set A, called a *feedback arc set*, of at most k edges in G such that $G - A$ contains no directed cycles?

Given the directed graph in Figure 7.4, sets $\{e_4, e_7\}$ and $\{e_5, e_6\}$ are feedback arc sets. Thus, the graph is a yes-instance for the FEEDBACK ARC SET for $k = 2$.

Theorem 7.7 LIST SYNTHESIS *is $\mathcal{NP}$-complete.*

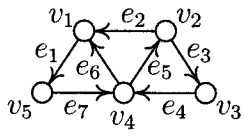


Figure 7.4: A directed graph.

Proof: It is easy to verify that the problem belongs to $\mathcal{NP}$, so we need only show that FEEDBACK ARC SET reduces to LIST SYNTHESIS. Let $G = (V, E)$ be a directed graph; we assume that each vertex in G belongs to a directed cycle. This assumption is without loss of generality because if a vertex v does not belong to any directed cycle, then a set of edges is a feedback arc set for G if and only if it is a feedback arc set for $G - v$. From G, we construct four sets $A_1 = A_2 = A_3 = A_4 = V \cup E$. Linear ordering π_1 is composed of consecutive subsequences of the form $v, e_1, e_2, \ldots, e_p$ where $\{e_1, e_2, \ldots, e_p\}$ is the set of edges directed out from vertex v. We obtain the linear ordering π_2 from π_1 by reversing π_1 and then replacing each subsequence $e_p, e_{p-1}, \ldots, e_1, v$ with $v, e_p, e_{p-1}, \ldots, e_1$. Linear ordering π_3 is similar to π_1 except that it consists of consecutive subsequences of the form $f_1, f_2, \ldots, f_q, v$ where $\{f_1, f_2, \ldots, f_q\}$ is the set of edges directed *into* v. Finally, we obtain π_4 from π_3 by reversing π_3 and then replacing each subsequence $v, f_q, f_{q-1}, \ldots, f_1$ with $f_q, f_{q-1}, \ldots, f_1, v$. For example, from the graph shown in Figure 7.4, we obtain the following linear orderings:

$$\begin{aligned}
\pi_1 &= v_1 e_1\ v_2 e_2 e_3\ v_3 e_4\ v_4 e_5 e_6\ v_5 e_7 \\
\pi_2 &= v_5 e_7\ v_4 e_6 e_5\ v_3 e_4\ v_2 e_3 e_2\ v_1 e_1 \\
\pi_3 &= e_2 e_6 v_1\ e_5 v_2\ e_3 v_3\ e_4 e_7 v_4\ e_1 v_5 \\
\pi_4 &= e_1 v_5\ e_7 e_4 v_4\ e_3 v_3\ e_5 v_2\ e_6 e_2 v_1
\end{aligned}$$

We will show that G has a feedback arc set of size k if and only if there is a linear ordering π of $V \cup E$ such that $\sum_{1 \le i \le 4} \text{diff}(\pi, \pi_i) \le L + 2k$, where L is the lower bound defined above.

To establish this result, we depend on the following claim:

Claim. Let $x, y \in V \cup E$. Then:

1. y is an edge directed into vertex x or x is an edge directed out from vertex y, and $c_{xy} = 1$ and $c_{yx} = 3$; or
2. y is an edge directed out from vertex x or x is an edge directed into vertex y, and $c_{yx} = 1$ and $c_{xy} = 3$; or

3. $c_{xy} = c_{yx} = 2$.

To prove this claim, we first consider the case where y is an edge adjacent to x. By definition, then, $y = (x, w)$ or $y = (w, x)$ for some vertex w. If $y = (x, w)$, then $x <_{\pi_1} y$ and $x <_{\pi_2} y$ because y is directed out from x. In π_3 and π_4, on the other hand, x and y do not belong to the defined subsequences and π_4 is obtained by reversing π_3, so $x <_{\pi_3} y$ and $x >_{\pi_4} y$, or $x >_{\pi_3} y$ and $x <_{\pi_4} y$. Thus, $c_{xy} = 1$ and $c_{yx} = 3$.

Otherwise, we have $y = (w, x)$. In this case, $x >_{\pi_3} y$ and $x >_{\pi_4} y$ because y is directed into x. In π_1 and π_2, on the other hand, x and y do not belong to the defined subsequences and π_2 is obtained by reversing π_1, so $x <_{\pi_1} y$ and $x >_{\pi_2} y$, or $x >_{\pi_1} y$ and $x <_{\pi_2} y$. Thus, $c_{xy} = 3$ and $c_{yx} = 1$.

The cases where x is an edge adjacent to y are symmetric so $c_{xy} = 1$ and $c_{yx} = 3$ if x is directed out from y, and $c_{xy} = 3$ and $c_{yx} = 1$ if x is directed into y.

The only remaining cases to handle occur when x and y are both vertices, both edges, or one is a vertex and the other is a non-adjacent edge. In these cases, we have $x <_{\pi_1} y$ and $x >_{\pi_2} y$, or $x >_{\pi_1} y$ and $x <_{\pi_2} y$, and $x <_{\pi_3} y$ and $x >_{\pi_4} y$, or $x >_{\pi_3} y$ and $x <_{\pi_4} y$. Thus, we have $c_{xy} = c_{yx} = 2$.

Having established the claim, it is now a simple matter to prove the correctness of our reduction.

First, suppose that A is a feedback arc set of size k for G. By definition, then, $G - A$ is acyclic so let π' be a topological sort of $G - A$ so that, if (u, v) is an edge in $G - A$, then $u >_{\pi'} v$. Then, let π be a linear ordering of $V \cup E$ obtained from π' by inserting each edge (u, v) in $G - A$ immediately before u, and inserting each edge (u, v) in A immediately after u. By our claim above, for each pair $x, y \in V \cup E$ such that $x <_\pi y$ and $c_{xy} > c_{yx}$, we have that $c_{xy} = c_{yx} + 2$ and either x is an edge directed into vertex y or else y is an edge directed out of vertex x. By the construction of π, then, x or y is an edge in A. Thus, we have shown that $\sum_{1 \leq i \leq 4} \text{diff}(\pi, \pi_i) = L + 2k$.

Now suppose that we have a linear ordering π of $V \cup E$ such that $\sum_{1 \leq i \leq 4} \text{diff}(\pi, \pi_i) = L + 2k$. By our claim above, if $x <_\pi y$ and $c_{xy} > c_{yx}$, then $c_{xy} = c_{yx} + 2$ and x or y is an edge (u, v) in G such that $u <_\pi y$. If A is the set of all such edges, then $|A| = k$ and π is a topological sort of $G - A$. Therefore, A is a feedback arc set for G. □

We note that our proof shows that the problem is $\mathcal{NP}$-complete for $p \geq 4$ subsets. For $p \leq 2$ sets, the problem can be easily solved in polynomial time. Hence, it remains open whether or not the problem is $\mathcal{NP}$-complete for $p = 3$ subsets. The current results for the 1-SIDED CROSSING MINIMIZATION problem are analogous if we consider the maximum degree of the vertices on the layer whose order is not fixed. More specifically, Muñoz

et al. [73] show that the 1-SIDED CROSSING MINIMIZATION problem is $\mathcal{NP}$-complete when the vertices on the fixed layer are leaves and the vertices on the other unfixed layer have maximum degree four. They also show that the problem is polynomial when, instead, the vertices on the unfixed layer have maximum degree two. When the maximum degree is three, the complexity question remains open. Because the hardness proofs are so similar for the two problems, we believe that answering the open problem for either will quickly lead to a similar answer for the other problem.

It is not difficult to modify algorithm OSCM to obtain an $\mathcal{FPT}$ algorithm for LIST SYNTHESIS. We do this by showing that Lemma 7.1 about suited pairs and natural orderings is true for the current problem. We say that two elements x and y in X have a *natural ordering* in which x is before y whenever $c_{xy} = 0$. A pair with a natural ordering are said to be *suited.*

Lemma 7.8 *Let X be a set of elements, $X_1, X_2, \ldots, X_p$ be $p \geq 1$ subsets of X, and π_i be a linear ordering each set X_i for $1 \leq i \leq p$. If π is a linear ordering of X such that the sum $\sum_{1 \leq i \leq p} \text{diff}(\pi_i, \pi)$ is minimized, then all suited pairs of elements in X appear in their natural ordering in π.*

Proof: By way of contradiction, suppose that x and y are suited but $x <_\pi y$ is not their natural ordering. Thus, for each $1 \leq i \leq p$, we have $x >_{\pi_i} y$. We obtain π_x from π by removing x and inserting just after y, and we obtain π_y from π by removing y and inserting just before x. In both π_x and π_y, then, x and y are in their natural ordering. We will show that $S_x = \sum_{1 \leq i \leq p} \text{diff}(\pi_i, \pi_x)$ or $S_y = \sum_{1 \leq i \leq p} \text{diff}(\pi_i, \pi_y)$ is strictly less than $S = \sum_{1 \leq i \leq p} \text{diff}(\pi_i, \pi)$. For each $1 \leq i \leq p$, we let $\alpha_i = |\{z \in X_i | x <_\pi z <_\pi y, z <_{\pi_i} y\}|$, $\beta_i = |\{z \in X_i | x <_\pi z <_\pi y, y <_{\pi_i} z <_{\pi_i} x\}|$, and $\gamma_i = |\{z \in X_i | x <_\pi z <_\pi y, x <_{\pi_i} z\}|$. Thus, $S_x = S - p + \sum_{1 \leq i \leq p} -\alpha_i - \beta_i + \gamma_i$ and $S_y = S - p + \sum_{1 \leq i \leq p} \alpha_i - \beta - \gamma_i$. Adding them together, we obtain $S_x + S_y = 2S - 2p + \sum_{1 \leq i \leq p} -2\beta_i < 2S$ so $S_x < S$ or $S_y < S$. However, this contradicts our assumption that S is minimum. □

Now, for $p = 2$ lists, we recall that LIST SYNTHESIS can be solved in polynomial time. In particular, when $p = 2$, we have $c_{xy} + c_{yx} = 2$, and if x and y are not suited, then $c_{xy} = c_{yx} = 1$. Thus, by Lemma 7.8, we solve the problem by first putting each suited pair in their natural order and then arbitrarily ordering the remaining pairs with $c_{xy} = c_{yx} = 1$.

For $p = 3$, on the other hand, we do not know whether or not the problem is $\mathcal{NP}$-hard, and, for $p \geq 4$, the problem is $\mathcal{NP}$-hard so we solve these cases using a bounded search tree algorithm. Algorithm initialization includes computing the crossing numbers of each pair of elements in X, ordering suited pairs, and then kernelization, just like our algorithms for 1-SIDED CROSSING MINIMIZATION. However, since $c_{xy} \leq p$ for each pair

of elements $x, y \in X$, initialization can be completed in $\mathcal{O}(p|X|^2)$ time. After initialization, we construct and traverse a bounded search tree by selecting a pair of unordered elements $x, y \in X$ at each non-leaf node and creating one child corresponding to setting $x < y$ and another child corresponding to setting $y > x$. Since x and y are not suited and $c_{xy} + c_{yx} = p$, the size $T(k)$ of the resulting search tree is bounded as follows for large values of k:

$$T(k) \leq \max_{1 \leq i \leq p-1} \{T(k-i) + T(k-p+i)\}.$$

We observe that $T(k) \leq \alpha^k$ for any $\alpha \geq 1$ that is a solution for each of the following polynomial inequalities: $\alpha^{p-i} - \alpha^{p-2i} - 1 \geq 0$ for all $2 \leq 2i \leq p$. In fact, this follows as long as $\alpha^{p-1} - \alpha^{p-2} - 1 \geq 0$ because $\alpha \geq 1$; therefore, the search tree has size $T(k) \leq \alpha^k$ for the positive root of the polynomial $\alpha^{p-1} - \alpha^{p-2} - 1 = 0$.

We call the algorithm Synthesizer:

Theorem 7.9 *Let X be a set of elements, and, for some $p \geq 1$, let $X_1, X_2, \ldots, X_p$ be subsets of X, each with a linear ordering π_i. For any integer $k \geq 0$, algorithm* Synthesizer *determines in $\mathcal{O}(\alpha^k + p \cdot |X|^2)$ time whether or not there exists a linear ordering π of X such that $\sum_{1 \leq i \leq p} \text{diff}(\pi, \pi_i) \leq k$, where α is the positive solution to $\alpha^{p-1} - \alpha^{p-2} - 1 = 0$.*

Thus, for $p \geq 3$ lists, the algorithm has running times similar to the following:

p	running time
3	$\mathcal{O}(1.618^k + \|X\|^2)$
4	$\mathcal{O}(1.466^k + \|X\|^2)$
5	$\mathcal{O}(1.381^k + \|X\|^2)$
6	$\mathcal{O}(1.325^k + \|X\|^2)$
7	$\mathcal{O}(1.286^k + \|X\|^2)$
8	$\mathcal{O}(1.256^k + \|X\|^2)$
100	$\mathcal{O}(1.035^k + \|X\|^2)$

As one would expect from the math, as p increases the running time in terms of k and $|X|$ decreases. However, this seems a little counter-intuitive because one would expect the difficulty of the problem to increases as p increases. To capture this notion, it might perhaps be better to parameterize in terms of $l = \frac{k}{p}$ rather than just k to capture the fact that crossings are coming from p different lists. In that case, our running times look like the following:

p	running time
3	$\mathcal{O}(4.236^l + \|X\|^2)$
4	$\mathcal{O}(4.619^l + \|X\|^2)$
5	$\mathcal{O}(5.024^l + \|X\|^2)$
6	$\mathcal{O}(5.412^l + \|X\|^2)$
7	$\mathcal{O}(5.817^l + \|X\|^2)$
8	$\mathcal{O}(6.194^l + \|X\|^2)$
100	$\mathcal{O}(30.74^l + \|X\|^2)$

7.7 Conclusions and Future Directions

In this chapter, we have described the most recent $\mathcal{FPT}$ algorithm for solving the 1-SIDED CROSSING MINIMIZATION problem. In further studies, we would like to see if its running time could be reduced by investigating cases where $c_{uv} + c_{vu} = 4$.

The other algorithm, called OSCM-C, shows how one way to incorporate divide-and-conquer into the bounded search tree algorithms with very little time penalty in order to obtain a heuristic speed-up. As we discussed in Chapter 6, we would like to determine whether or not this divide-and-conquer approach can be used to obtain theoretical running-time improvements.

Finally, we have described two related applications of 1-SIDED CROSSING MINIMIZATION, proved that their corresponding problems are $\mathcal{NP}$-complete and showed how to easily modify algorithm OSCM so that it solves these new problems. These modified algorithms are interesting because they show that bounded search tree algorithms can be quite flexible, an unusual property for exact solutions to hard problems.

Chapter 8

Experiments with $\mathcal{FPT}$ Algorithms

To Mike and Fran—*may your infectious love for adventure and one another never cool.*

The algorithms described in Chapters 6 and 7 for solving crossing minimization and planarization problems are among the first $\mathcal{FPT}$ algorithms used to draw graphs. Though, in theory, fixed parameter tractability sounds like a promising approach for solving certain $\mathcal{NP}$-hard problems, it is not at all clear that these solutions have efficient implementations that are competitive with other approaches.

In this chapter, we describe our implementations of two biplanarizing algorithms from Chapter 6, and two crossing minimization algorithms from Chapter 7. The first biplanarizing algorithm has a running time of $\mathcal{O}(6^k + |G|)$ and the second algorithm is a divide-and-conquer variation of this algorithm that has the same running time. We recall that, in Chapter 6, we actually describe a faster algorithm that runs in $\mathcal{O}(3.562^k + |G|)$ time. Unfortunately, we did not discover the branching rules described in Section 6.2 until recently. On the other hand, we did know how to construct and traverse the tree by spending constant time at each search tree node; therefore, the algorithm we use in the experiments in this chapter runs in $\mathcal{O}(6^k + |G|)$ time. In addition to this, our algorithm also pruned the search tree by considering only candidate edges for removal (see Section 6.5). We call this algorithm TLP-6. In Section 6.4, we show how to obtain a divide-and-conquer version of our $\mathcal{O}(3.562^k + |G|)$ time algorithm; however, our description could be used instead to obtain a divide-and-conquer version of TLP-6 with the same running time. Some of the experiments of this chapter are with such an algorithm, which we call TLP-C. Our crossing minimization experiments are with implementations of algorithms OSCM and OSCM-C as described in Chapter 7.

We show that our implementations are both competitive with existing approaches, and that there is much room for improving their performance. We are particularly interested in comparing our approach to the integer linear programming (ILP) approach of Jünger and Mutzel [57, 74] because, previously, theirs was the only practical approach for obtaining exact solutions to these problems.

8.1 Implementation Details

We implemented all of our algorithms in the Java programming language. We initially considered using C++ because it appears to be the most popular language for implementing algorithms. Indeed, there are some high-quality algorithms libraries available like the Library of Efficient Data Structures and Algorithms (LEDA) [70]. This is the library used to implement the branch-and-cut algorithms for 1-SIDED CROSSING MINIMIZATION and 2-LAYER PLANARIZATION described in [57, 74].

In spite of this, we eventually chose not to use C++. First of all, we decided not to use any of the algorithms libraries because none of them provided quite the right set of data structures and algorithms for our purposes so it wasn't clear that using any of the available libraries would have saved us much in the way of implementation time. Secondly, we decided to use Java instead of C++ because Java programs are easier to write, maintain and distribute. Java programs are easier to write because Java has a more principled design than C++; as a result, it is much easier to learn and tends to result in programs that are easier to understand. Java programs are easier to maintain because Java enforces many of the most important programming practices recommended by software engineering experts (see e.g. [65]). Finally, distribution of programs written in Java to users is more convenient because, whereas C++ programs are distributed as system-dependent machine code, Java programs are distributed in a so-called "byte code" form which is system-independent. As a result, just about any Java program can be executed by any computer with a Java byte-code interpreter. In fact, animations of our planarization algorithms are available as Java Applets on the World Wide Web. In Appendix C, we illustrate in detail one difference between Java and C++ which was an important factor in our decision to choose Java over C++.

We compiled the program using the byte-code compiler from the Java SDK version 1.4.1 from Sun Microsystems. We ran the experiments using their byte-code interpreter on a 1 GHz Pentium III computer with 1 Gb RAM running Debian Linux version 2.4.18. Actual running times depend on many factors such as the speed and architecture of the computer, other processes running in the background, the quality of the implementation, and the choice of implementation language; therefore, we also recorded the number of high-level steps taken by the algorithm to solve the problem. These values depend only on the input graph and the algorithm. We included the running times in our results only to give a rough idea of how long the implementation takes to planarize a graph.

8.2 Experiment Data

All of the pseudo-random graphs used in our experiments are from Stanford GraphBase [62]. We chose to use this graph generator in order to test our algorithms on the same graphs used by Jünger and Mutzel [57, 74]. Detailed steps for generating these graphs are described in Appendix B.

Stanford GraphBase generates pseudo-random graphs given a desired number of vertices and edges. The system first creates a graph on the desired number of vertices without any edges, and then adds the desired number of edges, each edge added by randomly selecting its end-vertices. With this generation algorithm, each labelled graph on the given number of vertices and edges has an equal chance of being generated. Unfortunately, the trouble with this type of graph generation is that the resulting graphs tend to have very little structure, whereas, the graphs generated by practical applications like DNA mapping, tend to have a lot of structure. In spite of this, we chose to use the Stanford GraphBase in order to compare our results with results from other experiments that also used Stanford GraphBase.

For future research, we would like to obtain graphs from DNA mapping and phylogenetics in order to better evaluate our the performance of our implementations in practice.

8.3 Two-Layer Planarization Experiments

As previously mentioned, we applied our two-layer planarization implementations to the bipartite graphs from the Stanford GraphBase [62] that were used in the experiments of Mutzel [74, 75]. See Appendix B for detailed instructions for how to generate the set of graphs used in these experiments.

Our experimental results are shown alongside those of Mutzel [75] in Table 8.1. Each row in the table contains average values from applying each algorithm to 100 different graphs. The column labelled "T" gives the average running time, and "Steps" gives the average number of search tree nodes traversed. The column labelled "%" denotes the number of graphs G for which the algorithm was able to compute $\mathsf{bpn}(G)$ in under 600 seconds. We note that Mutzel terminated the search for $\mathsf{bpn}(G)$ after 300 seconds. We chose to wait 600 seconds because, unlike branch-and-cut algorithms, ours do not maintain an improving upper bound as they run; consequently, it is more important for us to allow our algorithms to completely finish their computations in order to obtain any data with which to analyze the algorithms. Because branch-and-cut algorithms maintain an upper bound, Mutzel's experimental results include not only a running time but also an average guarantee of solution

value (Gar). More specifically, for each graph planarized, the value $\frac{UpBound-Sol}{UpBound} \times 100$ is computed, where Sol denotes the number of edges in a biplanar subgraph of G having the most edges among the biplanar subgraphs found, and UpBound denotes the value of the maintained upper bound determined by the linear programming relaxation when the time limit of 300 seconds expired. The average guarantee of solution value (Gar) in the table of results, denotes the average of these values over a set of 100 graphs.

It turns out that for the graphs investigated, our lower bound for $\mathsf{bpn}(G)$ is quite close to $\mathsf{bpn}(G)$ itself. The difference between them is shown in the column labelled *Diff* in Table 8.1. For denser graphs we would expect this value to be smaller than for sparser graphs because of the higher probability of having a spanning caterpillar. On average, this value is around 0.6, so even our lower bound which is calculated in $O(|G|)$ time is close to optimal.

8.3.1 ILP versus FPT

It would not be very meaningful to directly compare our running times to those of Mutzel because of environment differences. More specifically, Mutzel's results were originally reported in 1996 [74] so the computers used were slower than the ones we used. On the other hand, their implementation language was C++ and used the branch-and-cut library ABACUS [59], whereas our implementation language was Java and we did not use any other libraries.

It is, however, meaningful to compare the shapes of the $|E|$ versus running time (T) graphs. In the first 17 rows of Table 8.1, we see that the $\mathcal{FPT}$ implementation is quite efficient up to $|E| = 55$, finding exact solutions to all input graphs. After $|E| = 55$, our implementation of TLP-6 is able to obtain exact solutions to only a few input graphs for a maximum time of 10 minutes per graph. The branch-and-cut implementation, on the other hand, demonstrates poorest performance at $|E| = 50$. However, after $|E| = 50$, it improves as $|E|$ approaches 100.

Thus, we see that these two different approaches may be complimentary. It appears that, whereas $\mathcal{FPT}$ approach tend to be efficient on sparse graphs, the ILP approach tends to be efficient on dense graphs. A possible reason for this is that the ILP approach finds an optimal solution by repeatedly finding approximate solutions that close in on the optimal solution. For planarization, this means that an branch-and-cut algorithm begins with a biplanarizing set of size between $\mathsf{bpn}(G)$ and $|E| - |V| + 1$, and then finds increasingly smaller biplanarizing sets until one of size $\mathsf{bpn}(G)$ is found. For dense graphs, the probability that $\mathsf{bpn}(G) = |E| - |V| + 1$ is high, so the branch-and-cut algorithm begins with a solution very close to the optimal. Our $\mathcal{FPT}$ algorithm, on the other hand, finds $\mathsf{bpn}(G)$

by beginning with a lower bound for $\mathsf{bpn}(G)$ and then working upward toward the optimal. As a result, the $\mathcal{FPT}$ algorithm is more efficient when $\mathsf{bpn}(G)$ is close to the lower bound. In addition, the $\mathcal{FPT}$ algorithm traverses search trees whose sizes are bounded by $\mathsf{bpn}(G)$; therefore, smaller values of $\mathsf{bpn}(G)$ and, consequently, sparser graphs are preferred.

The divide-and-conquer version of the $\mathcal{FPT}$ algorithm, TLP-C, is the clear winner in all experiments except possibly the cases where $|V_i| = 80, 90, 100$ and $|E| = 160, 180, 200$, respectively. In these cases, we are not entirely sure which approach is the winner because the ILP approach was not able to obtain optimal solutions for these values of $|V_i|$ and $|E|$, and the divergence of its solutions from the optimal increases for larger values of $|V_i|$ and $|E|$.

Though we hypothesized that this version of the $\mathcal{FPT}$ algorithm would perform better than the original $\mathcal{FPT}$ algorithm, we had no idea that the improvement would be so dramatic. The biggest surprise is its improvement with respect to dense graphs. One possible explanation is that all graphs become fairly sparse at sufficient depth in the search tree so the algorithm is able to "divide-and-conquer" these graphs very efficiently.

8.3.2 Sparse Graph Experiments

In Chapter 6, we also hypothesized that if we applied TLP-C to sparse graphs of equal density but different sizes, then the running time should increase polynomially as graph size increased. This hypothesis arises from the assumption that a large sparse graph is composed of several smaller sparse subgraphs of similar density that are very "loosely" connected together.

Results from experiments designed to test this hypothesis are illustrated in Figures 8.1 and 8.2. Detailed results are given in Appendix D. We consider three graph densities ($|E|/|V|$): 0.6, 0.8 and 1.0. Figure 8.2 shows a closer view of the results for density $|E|/|V| = 0.6$.

Looking at the results, it is rather difficult to determine whether or not our hypothesis is correct. Of all three densities, $|E|/|V| = 0.6$ is the only one that might support our hypothesis. In other words, we might say that, for TLP-C, a sparse graph has density $|E|/|V| \leq 0.6$. To confirm this, it would be necessary to run further experiments with much larger graphs. This would give us a longer curve to look at.

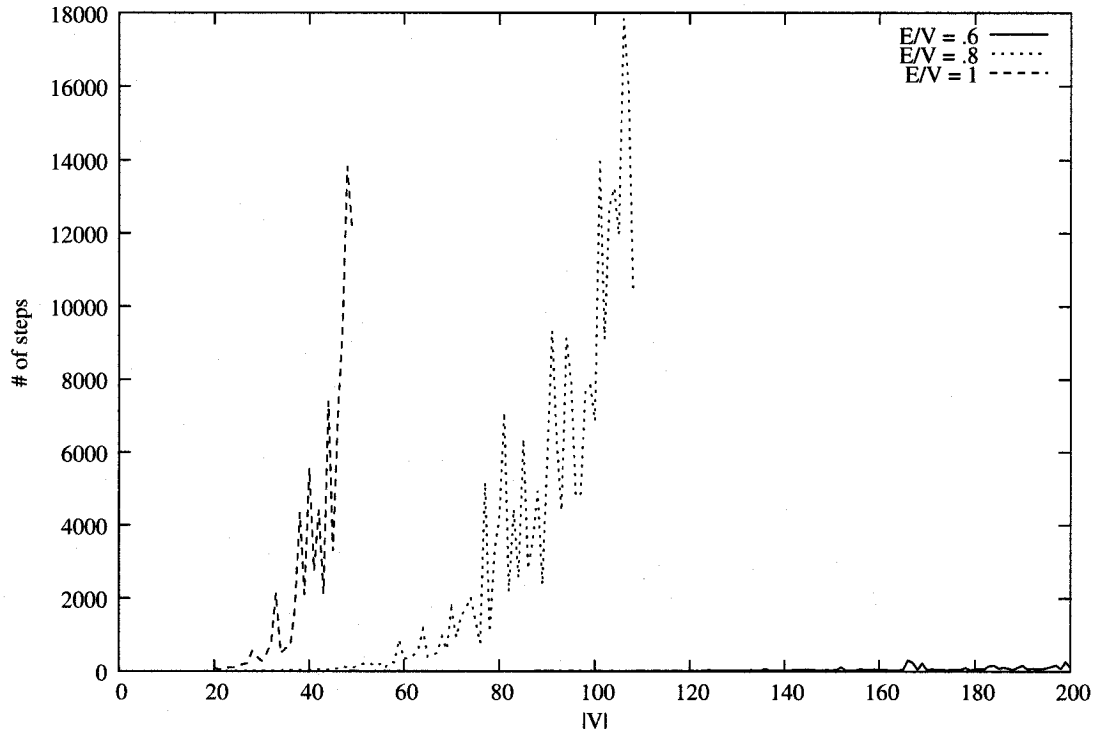


Figure 8.1: Number of Steps vs. $|V_i|$ when $|E|/|V| = 0.6, 0.8, 1$ from experiments with TLP-C.

8.4 One-Sided Crossing Minimization Experiments

After our results with planarization problems, we were surprised that our one-sided crossing minimization experimental results did not compare so favourably with those of Jünger and Mutzel [57, 58]. The results of our experiments are shown alongside their results in Table 8.2. Each row in the table corresponds to the average values of applying the algorithm to 100 different graphs generated by Stanford GraphBase [62]. For detailed instructions on how to generate these graphs or to repeat these experiments, see Appendix B. In Table 8.2, the column labelled "T" gives the average running time, and "Steps" gives the average number of search tree nodes traversed. The column labelled "%" denotes the number of graphs G for which the algorithm was able to compute $\mathsf{bcr}(G, \pi)$ in under 600 seconds. We note that Mutzel terminated the search for $\mathsf{bcr}(G, \pi)$ after 300 seconds.

8.4.1 Unexpected Results

Surprisingly, our crossing minimization results are almost the reverse of what we obtained for planarization. First of all, our $\mathcal{FPT}$ approach performs fairly well on dense graphs but

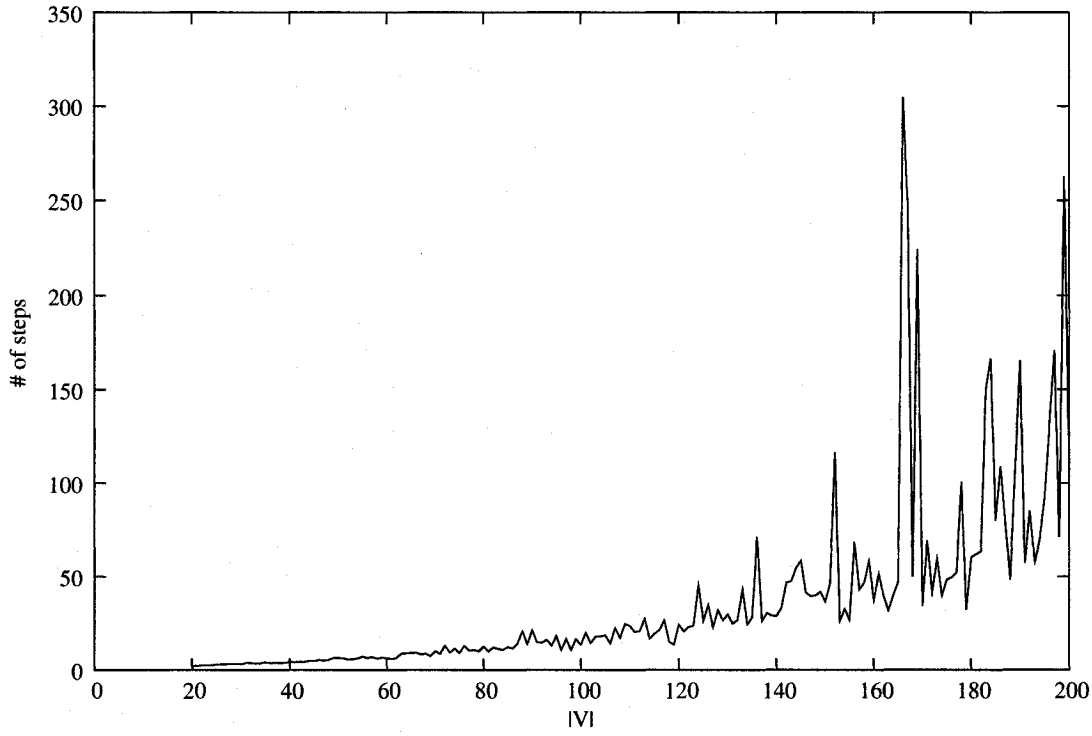


Figure 8.2: Number of Steps vs. $|V_i|$ when $|E|/|V| = 0.6$ from experiments with TLP-C.

dismally on sparse graphs. In addition, whereas TLP-C was significantly better than TLP-6 for planarization, the reverse is true for crossing minimization, OSCM performs slightly better than OSCM-C in almost all cases. This is surprising because our $\mathcal{FPT}$ algorithms for planarization traverse search trees of size up to $6^{\mathsf{bpn}(G)}$, whereas our $\mathcal{FPT}$ algorithms for crossing minimization traverse search trees of size up to $1.47^{\mathsf{bcr}(G,\pi)}$. It would seem, then, that if either of them were to be successful, then it would be the crossing minimization algorithms.

Fortunately, however, we may be able to explain our results. For sparse graphs, our experimental results, as expected, resemble an exponential function like $1.47^{\mathsf{bcr}(G,\pi)}$. However, in addition to this, the average value of $\mathsf{bcr}(G,\pi)$ for each value of $|V_i|$ appears to be in the order of $\Theta(|V_i|^2)$. This implies that the value of $\mathsf{bcr}(G,\pi)$ is tied to the size of the graph. Consequently, the running time of our $\mathcal{FPT}$ algorithm is exponential in the size of the graph G. It is easy, then, to imagine why the algorithm is not effective when applied to the set of sparse graphs. In the planarization results of Table 8.1, on the other hand, the value of $\mathsf{bpn}(G)$ is always much smaller than the number of vertices in the graph G. In other words, the sparse graphs in the crossing minimization experiments are actually not sparse in terms of their crossing numbers since even the sparse graphs have very large cross-

ing numbers. We hypothesize, then, that experiments on graphs with small crossing number would better show the strength of our implementations in comparison to other approaches.

With dense graphs, the running times of our $\mathcal{FPT}$ algorithms do not resemble a function anything like $O(1.47^{\mathsf{bcr}(G,\pi)})$. In fact, as the the average $\mathsf{bcr}(G,\pi)$ value increases, the average time to solve each problem decreases. This suggests that the ability of the algorithm to prune the bounded search tree increases drastically as edge density increases. Recall from Chapter 7 that the algorithm attempts to prune the search tree using kernelizing rules to automatically order certain pairs of vertices. One of these rules states that two vertices with the same set of neighbors can be arbitrarily ordered. The possibility of applying this rule increases as edge density nears its maximum value. The maximum density we consider occurs when $|V_i| = 20$ and $|E| = 360$. This is very close to the maximum possible density at $|V_i| = 20$ and $|E| = 400$. However, the application of this rule does not explain the overall trend which begins at $|V_i| = 20$ and $|E| = 120$, when each of the 20 vertices in the unordered layer has an average of 6 neighbors in the ordered layer containing a total of 20 vertices. In this case, it is very unlikely that two vertices in a random graph have the same set of neighbors. It is also unlikely that many vertices have less than four neighbors in these graphs so we can be reasonably sure that it is our dynamically maintained lower bound that is responsible for search tree pruning (see Section 7.2.2). At each search tree node, we decide whether to investigate its subtree by comparing a lower bound L against our parameter k. If L is greater than k, then we conclude that the subtree contains no solutions so we ignore it; otherwise, $L \leq k$ and we explore the subtree.

The experiments of Jünger and Mutzel [57] agree with this conclusion because they show that the *median heuristic* is one of the most effective heuristics for one-sided crossing minimization. As described in Section 1.3.2, this heuristic is called an "averaging heuristic" because it computes an "average position" for each vertex based on the positions of its neighbors, and then orders the vertices according to their "average position" values. More specifically, in this heuristic, the "average position" of a vertex is the same as the position of its neighbor with the median position over all its neighbors. Very roughly, then, if vertex u appears to the left of v in the final ordering, then there is a neighbor u' of u that appears to the left of a neighbor v' of v, and half of the neighbors of u appear before u' and half of the neighbors of v appear before v'. Thus, if we consider u and v alone, they contribute fewer edge crossings to the total solution if u is to the left of v. This coincides with exactly how our lower bound rule would prune the search tree, if applicable. This is because lower bounds corresponding to the branch in which u is left of v is strictly smaller than the branch in which u is right of v. Consequently, if applicable, the lower bound rule would prune the branch corresponding to u right of v and keep the other branch.

Table 8.1: Two-layer planarization results for bipartite graphs with $|V_i|$ vertices per bipartition class and $|E|$ edges. We note that each ILP experiment was terminated after 300 seconds, and each $\mathcal{FPT}$ experiment after 600 seconds.

				ILP		TLP-6			TLP-C		
$\|V_i\|$	$\|E\|$	bpn	Diff	Gar	T	T	Steps	%	T	Steps	%
20	20	0.72	0.06	0.00	0	0	5	100	0	2	100
20	25	1.47	0.12	0.00	0	0	8	100	0	2	100
20	30	3.00	0.24	0.00	0	0	25	100	0	5	100
20	35	4.91	0.41	0.00	1	0	90	100	0	14	100
20	40	7.65	0.34	0.00	6	0	595	100	0	76	100
20	45	10.66	0.15	0.03	26	0	1,829	100	0	85	100
20	50	14.29	0.19	0.67	100	2	53,416	100	4	4,694	100
20	55	18.27	0.35	0.53	81	41	1,767,872	96	1	946	100
20	60	22.53	0.30	0.37	56	-	-	-	5	6,232	100
20	65	27.21	0.49	0.32	54	-	-	-	3	3,645	97
20	70	31.75	0.46	0.13	26	-	-	-	7	8,263	99
20	75	36.54	0.32	0.13	22	-	-	-	2	2,249	100
20	80	41.41	0.28	0.03	12	-	-	-	2	2,060	99
20	85	46.27	0.22	0.10	20	-	-	-	5	5,366	100
20	90	51.23	0.21	0.02	8	-	-	-	6	6,503	99
20	95	56.20	0.17	0.00	4	-	-	-	8	8,276	99
20	100	61.10	0.09	0.00	4	-	-	-	4	5,243	98
20	40	7.43	0.32	0.00	6	0	495	100	0	95	100
30	60	11.30	0.46	0.13	49	1	10,559	100	0	356	100
40	80	15.63	0.49	0.55	150	9	243,760	100	3	3,002	100
50	100	19.40	0.56	1.45	253	43	1,281,694	97	14	11,876	99
60	120	23.55	0.66	1.86	279	-	-	-	64	48,240	96
70	140	27.51	0.59	2.35	294	-	-	-	129	91,339	88
80	160	30.64	0.49	2.90	300	-	-	-	-	-	-
90	180	-	-	3.48	301	-	-	-	-	-	-
100	200	-	-	4.67	300	-	-	-	-	-	-

Table 8.2: One-sided crossing minimization results for bipartite graphs with $|V_i|$ vertices per bipartition class and $|E|$ edges. We note that the each $\mathcal{FPT}$ experiment was terminated after 600 seconds.

				ILP	OSCM			OSCM-C		
$\|V_i\|$	$\|E\|$	bcr	Diff	T	Steps	T	%	Steps	T	%
20	40	179.36	0.45	0.02	548.12	0.14	100	657.06	0.51	100
20	80	970.50	1.99	0.03	7494.57	2.00	100	7,234.65	8.71	100
20	120	2,452.13	2.26	0.03	3394.35	0.96	100	3,280.13	4.23	100
20	160	4,655.86	2.12	0.04	3064.95	0.87	100	3,045.71	3.88	100
20	200	7,531.25	1.51	0.05	1966.16	0.60	100	2,038.21	2.75	100
20	240	11,353.04	1.34	0.07	1951.70	0.54	100	1,889.22	2.25	100
20	280	15,863.25	0.55	0.09	704.52	0.23	100	767.75	1.01	100
20	320	21,294.51	0.32	0.11	533.70	0.17	100	509.20	0.66	100
20	360	27,736.15	0.06	0.14	226.52	0.10	100	224.97	0.36	100
10	20	37.34	0.00	0.00	19.63	0.01	100	25.52	0.03	100
20	40	178.57	0.40	0.01	325.80	0.09	100	275.67	0.48	100
30	60	430.21	1.45	0.11	71,750.26	15.72	97	22,768.08	31.95	97
40	80	765.70	4.20	0.30	156,031.21	62.68	80	54,468.60	109.77	75
50	100	1252.20	4.90	0.68	302,475.43	137.86	35	61,245.62	196.07	29
60	120	1687.60	4.50	1.09	303,426.92	173.60	13	108,789.30	372.66	10
70	140	2479.00	14.00	4.46	204,463.00	522.82	2	282,390.00	804.99	1
80	160	3172.10	18.20	6.42	-	-	-	-	-	-
90	180	4132.80	28.80	25.13	-	-	-	-	-	-
100	200	5162.70	35.30	435.51	-	-	-	-	-	-

8.5 Conclusions and Further Experiments

In this chapter, we described our implementations of $\mathcal{FPT}$ algorithms for 2-layer graph drawing, and showed that they are competitive with existing branch-and-cut algorithms. Sine these are only the first experiments using fixed-parameter tractability to solve graph drawing problems, many more algorithms remain to be discovered, programming solutions with improved performance to be implemented, and, of course, many more experiments to be performed. In particular, in Chapter 6, we described planarization algorithms with improved running times that may have implementations with improved performance. The only way to know for sure, is to implement them and perform experiments.

In this chapter, all of our experiments involved pseudo-random graphs. To better study the performance of our approach and that of others, we would like to obtain graphs from applications like DNA mapping and phylogenetics. Related to this, we noted that the so-called "sparse" graphs used in our crossing minimization experiments were not really sparse in terms of the number of crossings in their drawings or in terms of applications like DNA mapping. We would like to know how our approach performs on these graphs, and whether or not it resembles the performance of our approach to planarization. We would also like to know how other approaches perform in these extremely sparse graphs.

Chapter 9

Conclusions and Future Research

To Jesus—*your thoughts are like discovering a theorem.*

In this thesis, we have investigated problems related to layered graph drawing. The purpose of this thesis was to obtain efficient algorithms for some of these problems that can be used to obtain drawings in practical applications. Indeed, we have found linear-time algorithms for obtaining proper 3-layer planar drawings and planar layered drawings of trees.

For drawings in which edges are permitted to bend, we had initially believed that the related recognition and drawing problems would be polynomial. Surprisingly, however, we have shown that they are in fact $\mathcal{NP}$-hard. In spite of this, we were able to obtain a linear-time algorithm for drawing 2-outerplanar graphs using our characterization for two-layer drawings. Our characterization may yet lead to efficient algorithms for other classes of graphs.

In the second part of the thesis, we studied two-layer drawings that may contain edge crossings. Here we were able to modify an existing $\mathcal{FPT}$ algorithm for the 1-SIDED CROSSING MINIMIZATION problem to obtain the first $\mathcal{FPT}$ algorithms for two related $\mathcal{NP}$-complete problems. For the 1-LAYER PLANARIZATION and 2-LAYER PLANARIZATION problems, we discovered new $\mathcal{FPT}$ algorithms with running times that are much better than previous algorithms. In the final chapter, we showed that parameterized complexity not only has a theoretical contribution but also a practical contribution to make to graph drawing.

We now conclude this thesis by listing the open problems and future directions raised by the investigations of this thesis.

Chapter 3 Proper 3-layer drawings.

- Is it possible to extend our results for proper 3-layer planarity to proper 4-layer planarity?
- Can our approach to proper 3-layer planarity be modified for non-proper layered drawings?

- Can the graph reduction approach [3,69,85] for recognizing graphs with treewidth equal to 3 and 4 be used for 3 and 4-layer planarity?

Chapter 4 Tree drawings.

- Is there an efficient way to compute the minimum number of layers required to draw any given tree (under the unconstrained, proper, short and upright drawing models)?
- Can the approach described in this chapter be used to draw other classes of graphs, e.g. outerplanar graphs?

Chapter 5 One-bend drawings.

- What is the complexity of recognizing graphs that have planar drawings with one-bend-per-edge on two non-parallel layers?
  Is there a concise characterization of the graphs that admit such drawings?
- What is the complexity of recognizing graphs that have planar drawings with one-bend-per-edge on a single convex poly-line composed of a constant number of line segments?
  Is there a concise characterization of the graphs that admit such drawings?
- Can our characterization of 2-layer, 1-bend planarity be generalized to characterize 3-layer, 1-bend planarity?
- Can we use our characterization of 2-layer, 1-bend planarity to obtain efficient algorithms for obtaining 2-layer, 1-bend planar drawings of any well-known classes of graphs? Recall that we show that this is the case for 2-outerplanar graphs. Are there other classes?

Chapter 6 Biplanarization algorithms.

- Are there asymptotically faster algorithms for planarization?
- ... by incorporating ambivalent edges and other heuristics?
- ... by incorporating the divide-and-conquer approach? Recall that the divide-and-conquer approach did not change the asymptotic running time but did improve experimental results quite dramatically as described in Chapter 8.
- Do the improved planarization algorithms described in this thesis suggest better approximation algorithms?
  Currently, there is a 2-approximation for 2-LAYER PLANARIZATION and a 3-approximation for 1-LAYER PLANARIZATION [25].

- Are there efficient $\mathcal{FPT}$ algorithms for k-layer planarization? Healy, Kuusik and Leipert [49] derive the minimal subgraphs that are not proper k-layer planar (after the vertices have been assigned to layers). From this result, it would be possible, for example, to derive minimal subgraphs that cannot belong to 3-layer planar graphs. If these subgraphs contain at most d edges each, then we have a trivial $\mathcal{O}(d^k \cdot |G|)$-time algorithm for the 3-layer planarization problem.

Chapter 7 One-sided crossing minimization and applications

- Are there asymptotically faster algorithms for 1-SIDED CROSSING MINIMIZATION?
- ... by incorporating the divide-and-conquer approach?
- Is there an efficient algorithm for 2-SIDED CROSSING MINIMIZATION?
  Note: we believe that we have discovered one algorithm which combines ideas from crossing minimization and planarization. Until we have formally described and proved the correctness of the algorithm, the problem remains open. The algorithm appears to have a running time of approximately $\mathcal{O}(12^k + |G|)$ but we have improvements in mind that could reduce the running time to $\mathcal{O}(8^k + |G|)$.
- Is the k-LIST SYNTHESIS problem $\mathcal{NP}$-hard for $p = 3$ lists?

Chapter 8 Experiments with $\mathcal{FPT}$ algorithms.

- Further experiments with data from the 1-SIDED CROSSING MINIMIZATION applications for comparing phylogenetic trees and synthesizing lists.
- Investigate ways of combining the advantages of our $\mathcal{FPT}$ implementations with branch-and-bound solutions.
  e.g. it appears that the $\mathcal{FPT}$ implementations are efficient because they are so problem-specific and can make effective use of a parameter. It would be interesting to study different ways of incorporating parameters into branch-and-bound solutions.
  [Suggested by Peter Eades (private communication, 2005).]
- Implement and repeat the experiments with the most recent planarization algorithms described in the thesis.
- Parallelize our implementations and performing experiments.
  Parallel bounded search tree algorithms have already been implemented for the VERTEX COVER problem [2].

- Implement and experiment with approximation approaches based on bounded search tree algorithms:

  e.g. Consider a search tree bounded by parameter $k = \alpha$ and another tree bounded by parameter $k = \alpha + 1$. If the first tree contains solution nodes, then the second tree contains at least as many solutions nodes, and possibly more. For example, for 2-LAYER PLANARIZATION, if S is a biplanarizing set of size α, then $S \cup \{e\}$ for any edge $e \notin S$ is also a biplanarizing set, but of size $\alpha + 1$. Thus, if the first tree has Δ solutions, then the second tree has at least $(|E| - \alpha)\Delta$ solution nodes, where $|E|$ is the number edges in the graph. Generalizing, then, the tree bounded by $k = \alpha + \beta$, for some $0 \leq \beta \leq |E| - \alpha$, has at least $(|E| - \alpha)(|E| - \alpha - 1) \cdots (|E| - \alpha - \beta + 1)\Delta$ which is approximately $|E|^{\beta}\Delta$ for small enough β. Of course, bounding the search tree by $k = \alpha + \beta$ rather than $k = \alpha$ results in a larger bounded search tree, however, searching the larger tree may actually be faster because it contains so many more solution nodes (as soon as the search algorithm finds a solution node, it stops). In fact, as β increases, the number of solutions increases exponentially, and much faster than the bounded search size increases. For example, one of our search tree algorithms constructs a tree of size 3.562^k so, as β increases, the tree size increases by a factor of 3.562 whereas the number of solutions increases by a factor of $|E|$.

  According to our experimental results shown in Table 8.1, the biplanarizing number of most graphs is less than or equal to our lower bound plus 1 or 2. Therefore, for most graphs, we could turn our algorithms into approximation algorithms by simply applying them with the parameter k set to the initial lower bound plus 2.

  [Suggested by Mike Hallett (private communication, 2005).]

- Experiment with heuristic approaches based on exact algorithmic solutions:

  e.g. Cook and Seymour [16] describe a heuristic approach to solving the travelling salesman problem by first applying several heuristic algorithms that each compute a tour, then creating a restricted graph consisting of the edges from each of these tours, and then applying an exact solution for solving the travelling salesman problem on the restricted graph. By definition, the resulting algorithm will do at least as well as the best heuristic algorithm, and often better.

  In terms of 2-LAYER PLANARIZATION problem, we would simply obtain a collection of heuristic and approximation algorithms, apply them to an input graph, mark each edge removed by one of these algorithms as "removable," and

then apply any one of the biplanarizing algorithms described in Chapter 6 to the same graph except that the algorithm may only consider removing edges marked "removable." For example, if the algorithm is branching on removing a 2-claw that contains only two edges marked "removable", then the algorithm considers only two branches rather than six branches.

[Suggested by Mike Fellows (private communication, 2004).]

Appendix A

Problems with the Proper 3-Layer Planarity Testing Algorithm of Fößmeier and Kaufmann

Fößmeier and Kaufmann [40] claim to have a linear-time algorithm for proper 3-layer planarity testing; however, the description of their algorithm contains ambiguities and does not appear to be correct.

Below is their proper 3-layer planarity testing algorithm as described in [40]. In the algorithm description, a vertex is *small* if it is leaf; on the other hand, if a vertex has at least two neighbors, then it is said to be *large*. We have also inserted our own boxed comments to clarify or point out what we believe to be errors in the algorithm.

test($G = (V_1, V_0; E)$, *borders*)

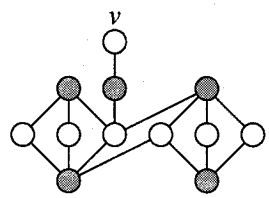
The input to the algorithm is a bipartite graph $G = (V_1, V_0; E)$ and a set of vertices *borders*. The algorithm returns true if and only if G has a proper 3-layer planar drawing in which the vertices of V_1 lie on the middle layer and, if *borders* contains at least one vertex, one vertex in *borders* has the smallest x-coordinate in the drawing, and, if *borders* contains two vertices, then the other vertex has the largest x-coordinate in the drawing. We observe, then, that if we want to know if G is proper 3-layer planar, then we set $borders = \emptyset$. If $\|borders\| > 2$, then the algorithm returns false.

(a) (* Simple checks *)

(1) **if** $|V_1| \leq 1$ **then** return true;

(2) **if** there is a small vertex $v \in V_1$ **then**
if $v \in borders$ **then** insert v's neighbor into *borders* **fi**;
call test($G \setminus \{v\}$, $borders \setminus \{v\}$);

Error: Consider the following graph $G = (V_1, V_0; E)$ where the vertices of V_0 are darkened:



The algorithm would first remove v from the graph as described in (a2). The remaining graph $G - v$ is clearly proper 3-layer planar so the recursive call to the algorithm should return true. However, by Theorem 3.1, this graph is not proper 3-layer planar because the main biconnected component is not safe (see Definition 3.4).

(3) **if** there is a vertex $v \in V_1$ with > 4 large neighbors **then** return false;

(* the non-trivial cases *)

(b) there are vertices in V_1 with four large neighbors; let v be such a vertex **if** v is a separator vertex **and** all (at most four) connected components of $G \setminus \{v\}$ are drawable in the two-sides-constrained model (where v gets a new border) **and** at most two of the components are no-caterpillar components **then** return true **else** return false;

A *separator vertex* is a cut-vertex, i.e. a vertex whose removal from a graph disconnects the graph. The *two-sides-constrained model* is exactly the proper 3-layer planar drawing model. We assume that the phrase "where v gets a new border" means that, in each recursive call, v is added to the set of *borders*. From the description of the algorithm, it sounds like the recursive calls are testing the drawability of components of $G - v$. In fact, according to the proof of correctness, the recursive calls are actually being applied to $G(V(H) + v)$ for each component H of $G - v$. More specifically, when the algorithm says that a component H is or is not a caterpillar, it should actually say that $G(V(H) + v)$ is or is not a caterpillar.

(c) there are some vertices in V_1 with three large neighbors; let v be such a vertex;

(1) **if** v is a separator vertex **then** we make the same test as in case (b)

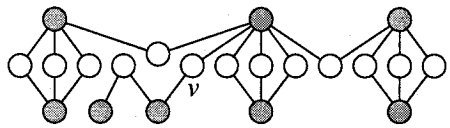
(2) **else** we need a more special case analysis involving separator edges which is omitted;

A *separator edge* is an edge (u, v) such that removing u and v disconnects the graph. The special case analysis mentioned here does not appear anywhere in the literature. Unfortunately, by definition, the separator edge referred to belongs to a cycle. Based on the fact that the bulk of the complexity in Theorem 3.1 relates to cycles, we would be very surprised if there is a straight-forward way to handle this case.

(d) (*all vertices in V_1 have at most 2 large neighbors *)

(1) **if** there is a separator vertex $v \in V_1$
then call *test* for the two connected components of $G \setminus \{v\}$ after inserting v into *borders* for these procedure calls;

Error: The graph drawn below is proper 3-layer planar but, if $v \in V_1$ is the separator vertex in case (d1), then one of the recursive calls would return false because one of the components of $G - v$ plus v does not have a proper 3-layer planar drawing in which v has the smallest or largest x-coordinate in the drawing.



We note that each vertex of V_1 in this counter-example has at most two non-leaf neighbors and none of them is a leaf; therefore, cases (a)-(c) do not apply.
It seems that that this error could be corrected by a more careful analysis of the components of $G - v$. Most likely, v should be added to *borders* in the recursive calls only for components of $G - v$ that are not caterpillars.

(2) **if** there is a separator vertex $w \in V_0$ **then**
if there are at most two no-caterpillar components among the connected components of $G \setminus \{w\}$ **then** call *test* for the no-caterpillar components after inserting w into *borders* for these calls **else** return false;

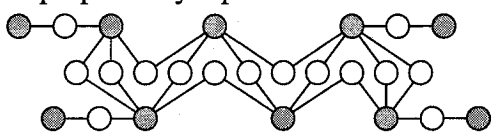
In case (b) above, we showed that the authors were actually not referring to components H of $G - w$ but rather subgraphs $G(V(H) + w)$. Here, again, the recursive calls apply to $G(V(H) + w)$.

(3) (* no separator vertices at all, vertices in V_1 have degree 2 *)
let L be the graph obtained by replacing all $v \in V_1$ by edges be-

tween its neighbors. **if** L is a ladder graph (an outerplanar graph with completely nested shortcut edges) **then** return true **else** return false;

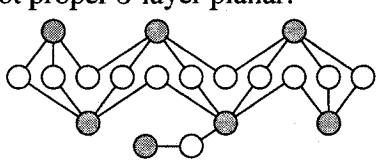
We assume that the authors mean that, not only must L be a ladder graph, but it must have a drawing in which one vertex of $border$, if $|border| > 0$, has the smallest x-coordinate in the drawing and the other vertex, if $|border| > 1$, has the largest x-coordinate in the drawing.

If this is the case, then the algorithm would reject the following graph even though it is proper 3-layer planar:



Since the graph contains four separator vertices in V_0, the algorithm would return false because any attempt to recursively draw the main biconnected component in the graph would require the parameter *borders* to contain all four separator vertices. This is due to case (d2) which is not described here but is similar to (d1), except that it applies to vertices in V_0 rather than V_1. However, as mentioned above, the algorithm would return false because *borders* may contain at most two vertices.

If this is not the case (i.e. the algorithm ignores *borders* in case (d3)), then the algorithm would return true for the following graph even though it is not proper 3-layer planar:



This is one of the problems that we address in our characterization by defining safe biconnected components (see Definition 3.4).

Appendix B

Detailed Instructions for Repeating Experiments

B.1 Two-Layer Planarization Experiments

The graphs for the experiments corresponding to the first 17 rows of Table 8.1 can be reproduced using the Stanford GraphBase [62]. We first generated 1700 random integers beginning with seed 5841. From each integer we generated a bipartite graph with $|V_i|$ vertices in each bipartition set and $|E|$ edges. The graphs used in the experiments corresponding to the last 9 rows can be generated by first generating 900 random integers beginning with seed 4741, and then, from each integer, generating a bipartite graph with $|V_i|$ vertices per bipartition set and $|E|$ edges.

Below is the corresponding code written in C:

1. "Dense graphs" corresponding to the first 17 rows of Table 8.1.

```
1.  unsigned long v = 20;
2.  int i;
3.  Graph *G;
4.  unsigned long seeds[1700];
5.  gb_init_seed(5841);
6.  for (i = 0; i < 1700; i++) {
7.    seeds[i] = gb_next_rand();
8.  }
9.  for (unsigned long e = 20, i = 0; e ≤ 100; e += 5, i++) {
10.   for (int j = 0; j < 100; j++, i++) {
11.     G = random_bigraph(v, v, e, 0, 0, 0, 0, 0, seeds[i]);
        // output G
12.   }
```

13. }

2. "Sparse graphs" corresponding to the last 9 rows of Table 8.1.

```
1. Graph *G;
2. int i;
3. unsigned long seeds[900];
4. gb_init_seed(4741);
5. for (i = 0; i < 900; i++) {
6.    seeds[i] = gb_next_rand();
7. }
8. for (unsigned long v = 20, i = 0; v ≤ 100; v += 10, i++) {
9.    unsigned long e = v+v;
10.   for (int j = 0; j < 100; j++, i++) {
11.      G = random_bigraph(v, v, e, 0, 0, 0, 0, 0, seeds[i]);
         // output G
12.   }
13. }
```

3. "Sparse graphs" used to compute Tables D.1, D.2 and D.3 are generated with similar code, initializing the Stanford GraphBase random number generator with seed 4741.

The following files need to be included in the programs above:

```
#include "gb_flip.h"
#include "gb_graph.h"
#include "gb_rand.h"
```

The reason why we put all the random graph seeds into an array (*seeds*[]) is because the random graph generator uses the random number generator. As a result, had we generated each random graph seed immediately before generating its corresponding graph, we would have generated a different sequence of random graphs.

In the code mentioned above, we include a comment that would be replaced by code for saving or otherwise communicating the contents of the graph to the planarization program. To see how this might be done, we include a piece of C code that prints the graph to the terminal (standard output), one edge per line and each edge printed as a pair of comma-separated vertices.

```
1. Vertex *vertex = G→vertices;
2. Arc *arc;
```

```
3. for (; vertex < G→vertices + G→n; vertex++) {
4.   for (arc = vertex→arcs; arc; arc = arc→next) {
5.     if (vertex < arc→tip)
6.       printf("%s, %s\n", vertex→name, arc→tip→name);
7.   }
8. }
```

B.2 One-Sided Crossing Minimization Experiments

Unfortunately, though we were able to obtain the same graphs as those used by Jünger and Mutzel [57], we were not able to determine how they selected an ordering for the vertices on the ordered layer. Here we describe how we selected the ordering. Suppose we create a given graph with the following function call to Stanford GraphBase:

G = random_bigraph(v, v, e, 0, 0, 0, 0, 0, $seeds[i]$);

The vertices in the bipartition class of size v can be referenced using $G \rightarrow vertices[i]$ where i is in the range $0 \ldots v-1$. The other bipartition, of size v, can also be referenced using $G \rightarrow vertices[i]$ but with i in the range $v \ldots v+v-1$. In our experiments, we chose the vertices in the first bipartition class to be unordered and the vertices in the second bipartition class to be ordered as they appear in the array $G \rightarrow vertices$. In other words, if our algorithm found a solution to the problem for graph G, then the layer containing the vertices of $G \rightarrow vertices[0..v-1]$ would be ordered so as to minimize the number of crossings when the remaining vertices of $G \rightarrow vertices[v..v+v-1]$ on the opposite layer are ordered as they appear in the array.

Below we describe the C code for generating the graphs used in the experiments:

1. "Dense graphs" corresponding to the first 9 rows of Table 8.2.
   This is the same as the "Dense graphs" code for the planarization experiments with the exception that, as can be seen from Table 8.2, 1700 must be replaced with 900, e=20 with e=40, and $e \leq 100$ with $e \leq 360$.

2. "Sparse graphs" corresponding to the last 10 rows of Table 8.1.
   This is the same as the "Sparse graphs" code for the planarization experiments with the exception that, as can be seen from Table 8.2, 900 must be replaced with 1000, and v=20 with v=10. In addition, we initialize the random number generator with 5841 instead of 4741.

Appendix C

Rationale for Choice of Java as Implementation Language

We illustrate one specific difference between Java and C++ encountered when implementing data structures in each language that store and operate on generic data objects. We highlight the difference because all recently designed algorithms libraries including ours contain dozens of such data structures.

In Java, it is quite straight-forward to implement such data structures because every data object belongs to the class `Object`. This is not, however, the case in C++ because there is no universal class like `Object` in the language.

In C++, there are basically two ways to implement generic data structures. One is to define a "generic" object class and require that all objects stored by the data structures belong to this class. Unfortunately, if a program uses this data structure and other third-party software libraries, such as LEDA or a library for displaying graphics, then, since the third-party software library knows nothing about this special "generic" object class, it will be impossible to store objects from such a library in the data structure. There are ways to work around this problem but they tend to create overly-complicated programs.

A second option is to use template classes. Template classes are a generalization of regular classes whose definitions include one or more class name placeholders. To create objects belonging to a template class, a program provides a specific object class for each placeholder. For example, if a list class is defined as a template class where the objects to be stored in the list are represented by a class name placeholder, then we create a list of say, integers, by replacing the placeholder with the integer class name. Template classes are an excellent idea in theory, but they are notoriously difficult to support in practice; as a result, most C++ compilers do not fully support them. In addition to this, when most C++ compilers encounter a statement creating an object from a template class, the compiler looks up the template, creates an entirely new class by replacing the class placeholders with actual class names and then compiling this new class. This obviously has the potential of

```
typedef void* VoidPtr;

class List
{
 public:

 ...

 Node& insert
  (Node& node, VoidPtr data);

 VoidPtr get(const Node& node);

 ...

}
```

```
template <class DATA>
class GenericList
{
 public:

 ...

 Node& insert
  (Node& node,
   const DATA& data) {
  return
   list.insert(node, &data);
 }

 DATA& get
  (const Node& node) {
  return
   *(DATA*)list.get(node);
 }

 ...

private:
 List list;
}
```

Figure C.1: Illustration of how to implement a generic list data structure in C++.

unnecessarily creating enormously large programs.

One way to work around these problems with templates, a solution used to implement LEDA, is to implement two classes for each data structure, a light-weight template class which acts as a "false front" for a second regular class whose data objects are "void" pointers. In C++, a "void" pointer can point to any machine address, and therefore any piece of data regardless of its class. Consider the list data structure as an example. The list template class, call it `GenericList`, defines operations on a sequence of data objects belonging to a class represented by a placeholder, while the regular list class, call it `List`, operates on a sequence of "void" pointers. A closer look at these definitions reveals that `List` defines a typical linked list that one might see in any introductory programming course, whereas `GenericList` contains little more than the data class placeholder and calls to a list of type `List`. See Figure C.1 for more details. The definition of `insert` in the template class first obtains a "void" pointer (`VoidPtr`) to the data object to insert and then inserts

it into the regular list object. Conversely, the definition of `get` in the template class first obtains the "void" pointer to the desired data object and then uses this pointer to return the data object it points to.

Clearly, this solution is much more complicated, prone to error, and time-consuming than a similar Java implementation.

Appendix D

Sparse Experimental Results

Here we present detailed results from the sparse graph experiments applied to the divide-and-conquer $\mathcal{FPT}$ algorithm described in Section 8.3.2 of Chapter 8. We considered three different graph densities, 0.6, 0.8 and 1.0. Each row in the three tables below give the average results of applying our algorithm to 100 bipartite graphs with $|V_i|$ vertices per bipartition set and $|E|$ edges. Column bpn contains the average biplanarization set size, *Diff* the average difference between the lower bound and bpn, *Steps* the average number of search tree nodes traversed to compute bpn, and % the number of graphs for which the algorithm found bpn in under 15 minutes.

If the algorithm was unable to solve at least 95 of the 100 input graphs, then we immediately aborted the experiment for the current edge density.

All of the graphs were generated using Stanford GraphBase. See Appendix B for details on how to generate exactly these graphs.

Table D.1: Sparse experiments ($|E|/|V| = 0.6$)

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
20	24	1.2100	0.1100	2.1000	100
21	25	1.2800	0.0900	2.1700	100
22	26	1.4700	0.1300	2.4400	100
23	27	1.5100	0.1800	2.4900	100
24	28	1.4800	0.1700	2.5100	100
25	30	1.7600	0.2200	2.8800	100
26	31	1.9200	0.2200	3.0400	100
27	32	1.9700	0.2500	3.1300	100
28	33	1.9500	0.2400	3.1700	100
29	34	2.0700	0.2200	3.2900	100
30	36	2.2500	0.3700	3.1200	100

Continued on next page

Sparse experiments ($|E|/|V| = 0.6$) *continued...*

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
31	37	2.1400	0.3500	3.8700	100
32	38	2.4200	0.3900	3.8200	100
33	39	2.2700	0.2900	3.4000	100
34	40	2.3400	0.2700	3.5900	100
35	42	2.7500	0.4900	4.1500	100
36	43	2.5600	0.3600	3.7500	100
37	44	2.8200	0.4300	3.8300	100
38	45	2.8200	0.4900	3.9000	100
39	46	2.6800	0.4600	3.8800	100
40	48	3.0200	0.5100	4.4800	100
41	49	2.8900	0.5100	4.0600	100
42	50	3.1100	0.5800	4.3900	100
43	51	2.9500	0.5100	4.3800	100
44	52	3.2300	0.4900	4.8900	100
45	54	3.3000	0.5600	4.8400	100
46	55	3.8800	0.8000	5.4000	100
47	56	3.5900	0.6000	5.0400	100
48	57	3.6900	0.6600	5.4300	100
49	58	3.8000	0.5900	6.5300	100
50	60	4.4400	0.9100	6.4300	100
51	61	4.0800	0.6700	6.3000	100
52	62	4.0400	0.8400	5.6100	100
53	63	4.1800	0.8800	5.8900	100
54	64	3.9600	0.7400	6.3200	100
55	66	4.3900	0.7700	7.1700	100
56	67	4.4000	0.9000	6.4900	100
57	68	4.5300	0.9300	7.0100	100
58	69	4.3400	0.8200	6.1300	100
59	70	4.4700	0.8600	6.7000	100
60	72	4.3800	0.7900	6.3700	100
61	73	4.6000	0.8300	5.9500	100
62	74	4.9100	0.9300	6.3900	100
63	75	4.7100	0.9500	8.8100	100

Continued on next page

Sparse experiments ($|E|/|V| = 0.6$) *continued...*

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
64	76	5.0500	1.0000	8.9400	100
65	78	5.1900	0.9800	9.0300	100
66	79	5.1400	1.0100	9.3300	100
67	80	5.3200	1.0700	8.2400	100
68	81	5.5000	1.0800	8.7800	100
69	82	5.2500	1.0400	7.4500	100
70	84	5.4700	1.1000	10.1100	100
71	85	5.5500	1.0900	8.5500	100
72	86	5.9400	1.1800	13.0000	100
73	87	6.2100	1.1500	9.2500	100
74	88	5.8600	1.1300	11.3800	100
75	90	5.8300	1.1700	8.8900	100
76	91	6.4200	1.1100	12.8400	100
77	92	6.3800	1.1900	10.3400	100
78	93	6.0900	1.2100	10.6700	100
79	94	6.1800	1.2200	9.9200	100
80	96	6.5400	1.2600	12.5000	100
81	97	6.8900	1.3300	9.8300	100
82	98	6.6000	1.2200	12.1200	100
83	99	6.4400	1.2500	11.2300	100
84	100	6.8700	1.4100	10.7000	100
85	102	7.2000	1.4800	12.2700	100
86	103	7.1300	1.4400	11.4400	100
87	104	6.7900	1.3400	14.2000	100
88	105	6.8500	1.3500	20.8200	100
89	106	6.8200	1.2800	13.8100	100
90	108	7.3200	1.4400	21.1300	100
91	109	7.2400	1.6000	14.9300	100
92	110	7.3500	1.4000	14.5400	100
93	111	7.5900	1.5900	16.2300	100
94	112	7.5900	1.6000	13.0100	100
95	114	7.8100	1.4700	18.2200	100
96	115	7.3400	1.5200	10.5400	100

Continued on next page

Sparse experiments ($|E|/|V| = 0.6$) *continued...*

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
97	116	8.0700	1.7500	16.5800	100
98	117	7.5100	1.5900	10.4600	100
99	118	8.0500	1.6700	16.5600	100
100	120	7.9100	1.6200	13.4800	100
101	121	8.3200	1.6800	19.7900	100
102	122	8.2200	1.7100	14.3700	100
103	123	8.4400	1.7100	17.7600	100
104	124	8.2500	1.6600	17.8900	100
105	126	8.7500	1.6900	18.4900	100
106	127	8.5900	1.7200	14.1600	100
107	128	8.4600	1.8300	22.3200	100
108	129	8.9800	1.8800	17.2200	100
109	130	8.9300	1.8900	24.6400	100
110	132	9.0200	1.8800	23.5900	100
111	133	8.9200	1.7600	20.2100	100
112	134	8.8900	1.8800	20.7800	100
113	135	9.2700	1.8200	27.5200	100
114	136	9.1500	1.9400	16.7000	100
115	138	9.8500	2.0100	19.1500	100
116	139	9.6200	2.0100	21.2200	100
117	140	9.6800	2.0400	26.5400	100
118	141	9.4700	1.9700	15.0300	100
119	142	9.3500	1.9900	13.5400	100
120	144	9.9800	2.0100	24.1600	100
121	145	9.3700	2.0500	20.5500	100
122	146	10.2100	2.1700	22.9900	100
123	147	10.2100	2.1100	23.5400	100
124	148	10.0200	2.0200	45.3100	100
125	150	10.4700	2.1300	26.2100	100
126	151	10.2900	2.1300	34.7000	100
127	152	10.6500	2.2500	22.7900	100
128	153	10.1700	2.0400	31.8200	100
129	154	10.2800	2.1800	26.5300	100

Continued on next page

Sparse experiments ($|E|/|V| = 0.6$) *continued...*

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
130	156	10.4100	2.2900	29.6600	100
131	157	11.0600	2.1300	24.7800	100
132	158	10.3700	2.1800	26.5900	100
133	159	10.9000	2.2700	42.9600	100
134	160	10.9500	2.3100	24.3900	100
135	162	10.9300	2.3400	27.9600	100
136	163	11.5800	2.5200	71.1500	100
137	164	11.2300	2.4300	26.1900	100
138	165	10.7800	2.0600	30.2100	100
139	166	11.2300	2.2200	28.9700	100
140	168	12.1100	2.5500	28.7200	100
141	169	11.1200	2.4800	33.0800	100
142	170	11.3000	2.3000	46.7700	100
143	171	11.3900	2.3500	47.3000	100
144	172	11.5100	2.6600	54.6200	100
145	174	12.0500	2.5900	58.4000	100
146	175	12.0000	2.6700	41.4100	100
147	176	12.0600	2.5300	39.3100	100
148	177	11.9700	2.5000	39.7200	100
149	178	12.3400	2.4500	41.7200	100
150	180	12.7600	2.8500	36.5400	100
151	181	12.5700	2.6600	46.1300	100
152	182	12.6100	2.6600	116.3800	100
153	183	12.1400	2.6600	26.1000	100
154	184	12.4000	2.7000	32.5400	100
155	186	12.1700	2.6500	26.6300	100
156	187	13.0000	2.7600	68.5100	100
157	188	12.8100	2.6800	42.9400	100
158	189	12.4900	2.5400	46.5600	100
159	190	13.0300	2.8900	58.3500	100
160	192	13.1400	2.8100	37.2900	100
161	193	13.5200	2.7500	51.4100	100
162	194	13.5500	2.9200	39.4600	100

Continued on next page

Sparse experiments ($|E|/|V| = 0.6$) *continued...*

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
163	195	12.9000	2.8100	31.6200	100
164	196	13.2500	2.8100	39.7100	100
165	198	13.0000	2.9100	46.9300	100
166	199	13.7100	2.9700	304.6900	100
167	200	13.4100	2.8100	248.2300	100
168	201	13.7000	2.8600	49.8600	100
169	202	14.2500	2.8400	224.2500	100
170	204	13.5300	2.9800	34.0200	100
171	205	13.5000	3.0000	69.0800	100
172	206	14.0600	3.1000	41.2800	100
173	207	14.0000	2.9100	59.6200	100
174	208	14.1900	3.0900	39.6000	100
175	210	14.3000	3.1200	48.2600	100
176	211	14.7000	3.1600	49.4600	100
177	212	14.5600	3.1000	52.1800	100
178	213	14.2400	2.9900	100.5000	100
179	214	14.5900	3.1100	32.0200	100
180	216	14.7900	3.1700	60.2300	100
181	217	14.8600	3.1900	61.9800	100
182	218	14.8100	3.2800	63.4100	100
183	219	15.4600	3.3000	150.1100	100
184	220	15.4200	3.0600	166.2500	100
185	222	15.4500	3.1200	79.9700	100
186	223	15.3600	3.2200	108.9200	100
187	224	15.5700	3.2800	79.3300	100
188	225	15.6300	3.1300	48.3100	100
189	226	15.6200	3.5000	108.0100	100
190	228	15.9200	3.4700	165.2300	100
191	229	15.5400	3.0300	57.2800	100
192	230	15.3800	3.2500	85.1600	100
193	231	16.1400	3.5100	57.8400	100
194	232	16.0400	3.3800	69.3300	100
195	234	16.6000	3.6000	91.4200	100

Continued on next page

Sparse experiments ($|E|/|V| = 0.6$) *continued...*

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
196	235	16.6300	3.5300	130.9900	100
197	236	16.6700	3.5000	170.6700	100
198	237	16.5600	3.6300	71.3400	100
199	238	16.1500	3.4300	262.5300	100
200	240	16.7200	3.7800	114.9000	100

Table D.2: Sparse experiments ($|E|/|V| = 0.8$)

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
20	32	3.6100	0.3800	7.5900	100
21	33	3.8100	0.4200	6.4400	100
22	35	4.2800	0.2600	9.2400	100
23	36	4.2800	0.3800	11.3700	100
24	38	4.6100	0.5700	9.6400	100
25	40	4.8600	0.4500	10.9800	100
26	41	4.9900	0.4800	13.8400	100
27	43	5.2500	0.5000	13.0200	100
28	44	5.4000	0.6100	16.5700	100
29	46	5.6900	0.5600	16.5500	100
30	48	6.0000	0.5800	13.6300	100
31	49	5.8600	0.5800	19.5100	100
32	51	6.3400	0.5900	22.5900	100
33	52	6.3200	0.6300	21.9800	100
34	54	6.5800	0.5700	20.9200	100
35	56	7.1600	0.6900	28.4600	100
36	57	6.8700	0.6900	26.3900	100
37	59	7.6900	0.8900	37.5200	100
38	60	7.3900	0.7200	35.2000	100
39	62	7.4500	0.8700	24.2000	100
40	64	7.9800	0.7700	48.0600	100
41	65	7.7500	0.8200	39.5600	100
42	67	8.2700	0.8900	58.7300	100
43	68	8.1800	0.9400	40.8600	100

Continued on next page

Sparse experiments ($|E|/|V| = 0.8$) *continued...*

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
44	70	8.7700	0.8900	46.0400	100
45	72	9.0400	0.8900	56.2900	100
46	73	9.1700	0.9800	84.0900	100
47	75	9.5500	0.9100	126.1800	100
48	76	9.4700	0.8700	113.0500	100
49	78	10.0100	1.0300	92.4700	100
50	80	10.5200	1.1100	136.7300	100
51	81	10.3100	1.1700	198.6600	100
52	83	10.4100	1.0000	200.1900	100
53	84	10.5200	0.9200	220.0600	100
54	86	10.9700	1.0700	140.7300	100
55	88	11.3400	1.0800	203.9600	100
56	89	11.0700	1.2300	84.2700	100
57	91	11.6600	1.1900	259.3400	100
58	92	11.2700	1.1100	244.5600	100
59	94	12.2900	1.2500	869.8700	100
60	96	12.0100	1.2000	344.8200	100
61	97	12.2300	1.3800	358.4000	100
62	99	12.8600	1.4300	443.4300	100
63	100	12.6100	1.2000	572.0500	100
64	102	13.2800	1.3600	1156.2000	100
65	104	13.3000	1.3900	399.9700	100
66	105	12.9900	1.3400	455.8200	100
67	107	13.8800	1.4800	507.3000	100
68	108	13.8600	1.5300	950.1100	100
69	110	13.8600	1.1900	587.9800	100
70	112	14.7900	1.4600	1717.9700	100
71	113	14.2200	1.3900	928.5700	100
72	115	14.8100	1.4800	1507.3200	100
73	116	15.0100	1.3600	1727.4900	100
74	118	15.1200	1.3900	2021.3100	100
75	120	15.3100	1.6100	1392.1700	100
76	121	15.6768	1.3737	780.1717	99

Continued on next page

Sparse experiments ($|E|/|V| = 0.8$) *continued...*

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
77	123	15.8600	1.5100	5176.3100	100
78	124	15.5400	1.7000	1078.1700	100
79	126	16.2000	1.5100	3264.2400	100
80	128	16.7000	1.6400	4236.1600	100
81	129	16.8788	1.6869	7088.6364	99
82	131	16.7100	1.5100	2181.8900	100
83	132	16.8800	1.8900	4438.7900	100
84	134	16.9600	1.7700	2553.0100	100
85	136	17.8600	1.8200	6366.0300	100
86	137	17.4900	1.6500	2793.9600	100
87	139	17.8990	1.7879	3600.2828	99
88	140	17.2245	1.6429	4942.0510	98
89	142	17.8900	1.7600	2365.8300	100
90	144	18.4800	1.7800	5854.8500	100
91	145	18.7172	1.8889	9330.7879	99
92	147	18.5859	1.6263	6254.2727	99
93	148	18.6633	1.8469	4404.5816	98
94	150	19.4900	1.8500	9138.0100	100
95	152	19.4747	1.9091	7949.0303	99
96	153	19.3000	1.9000	4799.7200	100
97	155	19.9495	1.8586	4828.9192	99
98	156	19.8788	2.1212	7684.8081	99
99	158	20.0000	1.8041	7835.2474	97
100	160	20.3434	1.9899	6872.7980	99
101	161	20.4639	2.0206	13975.3093	97
102	163	20.4000	1.9895	9109.4211	95
103	164	20.8687	1.9091	12725.3131	99
104	166	21.2041	2.0918	13246.1837	98
105	168	21.8367	2.0306	11963.5816	98
106	169	22.0102	2.1429	17847.4694	98
107	171	21.8660	2.1031	16022.3093	97
108	172	21.8676	2.4559	10505.5441	68

Table D.3: Sparse experiments ($|E|/|V| = 1.0$)

$\|V_i\|$	$\|E\|$	bpn	Diff	Steps	%
20	40	7.4300	0.3200	94.5800	100
21	42	7.9600	0.3600	56.8200	100
22	44	8.5200	0.1900	59.2800	100
23	46	8.8500	0.3000	103.1100	100
24	48	9.3500	0.3900	102.2900	100
25	50	9.6100	0.3500	153.4900	100
26	52	9.9500	0.4500	194.2800	100
27	54	10.4000	0.3500	219.0700	100
28	56	10.8400	0.4100	551.9200	100
29	58	11.2500	0.4200	396.5000	100
30	60	11.5700	0.4300	281.6800	100
31	62	11.8200	0.3600	480.5300	100
32	64	12.5600	0.4500	788.2300	100
33	66	12.8900	0.4800	2138.3900	100
34	68	13.1400	0.3700	486.5400	100
35	70	13.7200	0.4800	625.9600	100
36	72	13.7700	0.4800	678.3400	100
37	74	14.7400	0.5500	1725.5000	100
38	76	14.8500	0.3800	4335.5300	100
39	78	15.1800	0.5200	2091.4100	100
40	80	15.5900	0.4600	5559.7600	100
41	82	15.8500	0.5700	2736.6200	100
42	84	16.2200	0.4700	4414.7000	100
43	86	16.2900	0.5000	2124.2100	100
44	88	17.3232	0.5152	7407.7374	99
45	90	17.2400	0.5100	3284.0400	100
46	92	17.8200	0.5300	6941.5100	100
47	94	18.4343	0.4444	9415.6970	99
48	96	19.0101	0.4646	13834.9596	99
49	98	19.1622	0.5946	12165.2703	37

Bibliography

[1] Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Chris T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2001. 15

[2] Faisal N. Abu-Khzam, Michael A. Langston, and Pushkar Shanbhag. Scalable parallel algorithms for difficult combinatorial problems: A case study in optimization. In *International Conference on Parallel and Distributed Computing and Systems (PDCS)*. ACTA Press, 2003. 179

[3] S. Arnborg and A. Proskurowski. Characterization and recognition of partial 3-trees. *SIAM Journal of Algebraic and Discrete Methods*, 7(2):305–314, 1986. 56, 178

[4] R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Analysis and Design of Algorithms for Combinatorial Problems*, 25:27–46, 1985. 14

[5] Wilhelm Barth, Michael Jünger, and Petra Mutzel. Simple and efficient bilayer cross counting. In Michael Goodrich and Stephen Kobourov, editors, *Graph Drawing, 10th International Symposium (GD 2002)*, volume 2528 of *Lecture Notes in Computer Science*, pages 130–141. Springer-Verlag, 2002. 6

[6] Guiseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999. 1, 6, 8

[7] F. Bernhart and P.C. Kainen. The book thickness of a graph. *Journal Combinatorial Theory, Series B*, 27(3):320–331, 1979. 80, 91, 101

[8] Therese Biedl. Drawing outer-planar graphs in $o(n \log n)$ area. In Michael Goodrich and Stephen Kobourov, editors, *Graph Drawing, 10th International Symposium (GD 2002)*, volume 2528 of *Lecture Notes in Computer Science*, pages 54–65. Springer-Verlag, 2002. 79

[9] Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In I. Privara and P. Ruzicka, editors, *Proceedings of the 22nd International Symposium of Mathematical Foundations of Computer Science (MFCS 1997)*, volume 1295 of *Lecture Notes in Computer Science*, pages 19–36. Springer-Verlag, 1997. 18

[10] Ibrahim Cahit and Mehmet Ozel. The characterization of all maximal planar graphs, Manuscript. `http://www.emu.edu.tr/~cahit/prprnt.html`, 2003. 90

[11] M. J. Carpano. Automatic display of hierarchized graphs for computer aided decision analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(11):705–715, 1980. 1

[12] Tiziana Catarci. The assignment heuristic for crossing reduction. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(3):515–521, 1995. 9

[13] Hubert Chan. A parameterized algorithm for upward planarity testing (extended abstract). In Susanne Albers and Tomasz Radzik, editors, *12th annual European Symposium on Algorithms (ESA 2004)*, volume 3221 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 2004. 16

[14] James Cheetham, Frank Dehne, Andrew Rau-Chaplin, Ulrike Stege, and Peter J. Taillon. Solving large FPT problems on coarse-grained parallel machines. *Journal of Computer and System Sciences*, 67(4):691–706, 2003. 14, 15

[15] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41(2), 2001. 122

[16] William Cook and Paul Seymour. Tour merging via branch-decomposition. *Journal on Computing*, 15:233–248, 2003. 180

[17] Sabine Cornelsen, Thomas Schank, and Dorothea Wagner. Drawing graphs on two and three lines. In Michael Goodrich and Stephen Kobourov, editors, *Graph Drawing, 10th International Symposium (GD 2002)*, volume 2528 of *Lecture Notes in Computer Science*, pages 31–41. Springer-Verlag, 2002. 4, 28, 116

[18] Sabine Cornelsen, Thomas Schank, and Dorothea Wagner. Drawing graphs on two and three lines. *Journal of Graph Algorithms and Applications*, 8(2):161–177, 2004. 116

[19] Carlos Cotta, Christian Sloper, and Pablo Moscato. Evolutionary search of thresholds for robust feature set selection: Application to the analysis of microarray data.

In Günther R. Raidl, Stefano Cagnoni, Jürgen Branke, David Corne, Rolf Drechsler, Yaochu Jin, Colin G. Johnson, Penousal Machado, Elena Marchiori, Franz Rothlauf, George D. Smith, and Giovanni Squillero, editors, *Applications of Evolutionary Computing, EvoWorkshops 2004*, volume 3005 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2004. 17

[20] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. *Handbook of Theoretical Computer Science*, B, chapter 5, 1990. 15

[21] Rodney G. Downey and Michael R. Fellows. *Parametrized Complexity*. Springer-Verlag, 1999. 15

[22] Stefan Dresbach. A new heuristic layout algorithm for dags. *Operations Research Proceedings*, pages 121–126, 1994. 9

[23] Vida Dujmović, Michael R. Fellows, Michael T. Hallett, M. Kitching, Giuseppe Liotta, Catherine McCartin, Naomi Nishimura, Prabhakar Ragde, Frances A. Rosamond, Matthew Suderman, Sue Whitesides, and David R. Wood. A fixed-parameter approach to two-layer planarization. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing, 9th International Symposium (GD 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2001. 17, 123, 124, 128

[24] Vida Dujmović, Michael R. Fellows, Michael T. Hallett, M. Kitching, Giuseppe Liotta, Catherine McCartin, Naomi Nishimura, Prabhakar Ragde, Frances A. Rosamond, Matthew Suderman, Sue Whitesides, and David R. Wood. On the parameterized complexity of layered graph drawing. In Friedhelm Meyer auf der Heide, editor, *Algorithms, 9th European Symposium (ESA 2001)*, volume 2161 of *Lecture Notes in Computer Science*, pages 488–499. Springer-Verlag, 2001. 4, 18, 56, 58

[25] Vida Dujmović, Michael R. Fellows, Michael T. Hallett, M. Kitching, Giuseppe Liotta, Catherine McCartin, Naomi Nishimura, Prabhakar Ragde, Frances A. Rosamond, Matthew Suderman, Sue Whitesides, and David R. Wood. A fixed-parameter approach to two-layer planarization. *Algorithmica*, to appear. 17, 121, 132, 141, 147, 178

[26] Vida Dujmović, Henning Fernau, and Michael Kaufmann. One-sided crossing minimization revisited. In Giuseppe Liotta, editor, *Graph Drawing, 11th International Symposium (GD 2003)*, volume 2912 of *Lecture Notes in Computer Science*, pages 332–344. Springer-Verlag, 2003. 17, 20, 149, 150, 154, 155

[27] Vida Dujmović, Matthew Suderman, and David R. Wood. Personal communication, 2002. 61, 66

[28] Vida Dujmović and Sue Whitesides. An efficient fixed parameter tractable algorithm for 1-sided crossing minimization. In Michael Goodrich and Stephen Kobourov, editors, *Graph Drawing, 10th International Symposium (GD 2002)*, volume 2528 of *Lecture Notes in Computer Science*, pages 118–129. Springer-Verlag, 2002. 16, 17, 151

[29] Tim Dwyer and Falk Schreiber. Optimal leaf ordering for two and a half dimensional phylogenetic tree visualisation. In Neville Churcher and Clare Churcher, editors, *Australasian Symposium on Information Visualisation, (invis.au'04)*, volume 35 of *Conferences in Research and Practice in Information Technology*, pages 109–115, Christchurch, New Zealand, 2004. ACS. 1, 156, 158

[30] Peter Eades and David Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combinatoria*, 21(A):89–98, 1986. 9

[31] Peter Eades, Brendan McKay, and Nick Wormald. On an edge crossing problem. In *Proceedings of the 9th Australian Computer Science Conference*, pages 327–334. Australian National University, 1986. 24

[32] Peter Eades and Sue Whitesides. Drawing graphs in two layers. *Theoretical Computer Science*, 131(2):361–374, 1994. 6, 156

[33] Peter Eades and Nick Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994. 6, 9, 10

[34] John A. Ellis, Ivan Hal Sudborough, and Jonathan Turner. The vertex separation and search number of a graph. *Information and Computation*, 113(1):50–79, 1994. 59, 60, 76

[35] Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors. *Proceedings of the 13th international conference on World Wide Web, WWW 2004*. ACM, 2004. 1

[36] Michael R. Fellows, Catherine McCartin, Francis A. Rosamond, and Ulrike Stege. Coordinatized kernels and catalytic reductions: An improved fpt algorithm for max leaf spanning tree and other problems. In *Foundations of Software Technology and Theoretical Computer Science 20th Conference (FST TCS 2000)*, volume 1974 of *Lecture Notes in Computer Science*, pages 240–251. Springer, 2000. 15

[37] Stefan Felsner, Giuseppe Liotta, and Stephen K. Wismath. Straight-line drawings on restricted integer grids in two and three dimensions (extended abstract). In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing, 9th International Symposium (GD 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 328–342. Springer-Verlag, 2001. 58, 59, 60

[38] Stefan Felsner, Giuseppe Liotta, and Stephen K. Wismath. Straight-line drawings on restricted integer grids in two and three dimensions. *Journal of Graph Algorithms and Applications*, (special issue):1–33, 2003. 64, 75

[39] Henning Fernau. Two-layer planarization: Improving on parameterized algorithmics. In Peter Vojtás, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *31st Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2005)*, volume 3381 of *Lecture Notes in Computer Science*, pages 137–146. Springer-Verlag, 2005. 17, 20, 121, 122, 146

[40] Ulrich Fößmeier and Michael Kaufmann. Nice drawings for planar bipartite graphs. In Gian Carlo Bongiovanni, Daniel P. Bovet, and Giuseppe Di Battista, editors, *Proceedings of the 3rd Italian Conference on Algorithms and Complexity (CIAC 1997)*, volume 1203 of *Lecture Notes in Computer Science*, pages 122–134. Springer-Verlag, 1997. 4, 28, 56, 182

[41] Michael R. Garey and David S. Johnson. *Computers and Itractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co, 1979. 14, 160

[42] Michael R. Garey and David S. Johnson. Crossing number is NP-complete. *SIAM Journal of Algebraic Discrete Methods*, 4(3):312–316, 1983. 6

[43] Emilio Di Giacomo, Walter Didimo, Giuseppe Liotta, and Matthew Suderman. Hamiltonian-with-handles graphs and the k-spine drawability problem. In János Pach, editor, *Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 262–272. Springer, 2004. viii

[44] Emilio Di Giacomo, Walter Didimo, Giuseppe Liotta, and Stephen K. Wismath. Curve-constrained drawings of planar graphs. *Computational Geometry: Theory and Applications*, 30:1–23, 2005. 80, 81

[45] Frank Harary and Allen Schwenk. A new crossing number for bipartite graphs. *Utilitas Mathematica*, 1:203–209, 1972. 3, 24

[46] Patrick Healy and Ago Kuusik. The vertex-exchange graph: A new concept for multi-level crossing minimisation. In Jan Kratochvíl, editor, *Graph Drawing, 7th International Symposium (GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 205–216. Springer-Verlag, 1999. 7, 13

[47] Patrick Healy and Ago Kuusik. The vertex-exchange graph and its use in multi-level graph layout. Technical Report UL-CSIS-99-1, Department of Computer Science and Information Systems, University of Limerick, April 1999. 7, 13

[48] Patrick Healy and Ago Kuusik. Algorithms for multi-level graph planarity testing and layout. *Theoretical Computer Science*, 320(2-3):331–344, 2004. 4

[49] Patrick Healy, Ago Kuusik, and Sebastian Leipert. A characterization of level planar graphs. *Discrete Mathematics*, 280(1-3):51–63, 2004. 179

[50] Patrick Healy and Karol Lynch. Fixed-parameter tractable algorithms for testing upward planarity. In Peter Vojtás, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *31st Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2005)*, volume 3381 of *Lecture Notes in Computer Science*, pages 199–208. Springer, 2005. 16

[51] Lenwood S. Heath and Arnold L. Rosenberg. Laying out graphs using queues. *SIAM Journal on Computing*, 21(5):927–958, 1992. 4, 27

[52] Johan Høastad. Some optimal inapproximability results. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC 1997)*, pages 1–10. ACM Press, 1997. 14

[53] Johan Høastad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182:105–142, 1999. 14

[54] Michael Jünger, Eva K. Lee, Petra Mutzel, and Thomas Odenthal. A polyhedral approach to the multi-layer crossing minimization problem. In Giuseppe Di Battista, editor, *Graph Drawing, 5th International Symposium (GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 13–24. Springer-Verlag, 1997. 7, 11, 13

[55] Michael Jünger and Sebastian Leipert. Level planar embedding in linear time. *Journal of Graph Algorithms and Applications*, 6(1):67–113, 2002. 4

[56] Michael Jünger, Sebastian Leipert, and Petra Mutzel. Level planarity testing in linear time. In Sue Whitesides, editor, *Graph Drawing, 6th International Symposium (GD*

'98), volume 1547 of *Lecture Notes in Computer Science*, pages 224–237. Springer-Verlag, 1998. 4

[57] Michael Jünger and Petra Mutzel. Exact and heuristic algorithms for 2-layer straightline crossing minimization. In Franz-Josef Brandenburg, editor, *Graph Drawing, Symposium on Graph Drawing (GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 337–348. Springer-Verlag, 1995. 166, 167, 168, 171, 173, 188

[58] Michael Jünger and Petra Mutzel. 2-layer straightline crossing minimization: performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1(1):1–25, 1997. 6, 9, 13, 171

[59] Michael Jünger and S. Thienel. The ABACUS-system for branch and cut and price algorithms in integer programming and combinatorial optimization. In *Software–Practice and Experience*, volume 30, pages 1324–1352, 2000. 169

[60] Michael Kaufmann and Dorothea Wagner. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001. 1, 6

[61] Michael Kaufmann and Roland Wiese. Embedding vertices at points: Few bends suffice for planar graphs. *Journal of Graph Algorithms and Applications*, 6(1):115–129, 2002. 5, 80, 81, 82

[62] Donald Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, Addison-Wesley Publishing Company, 1993. 168, 171, 186

[63] Arie M. C. A. Koster, Hans L. Bodlaender, and Stam P. M. van Hoesel. Treewidth: Computational experiments. In J. Breese and D. Koller, editors, *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence*, pages 32–39. Morgan Kaufmann Publishers, 2001. 19

[64] Arie M. C. A. Koster, Stan P. M. van Hoesel, and Antoon W. J. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40(3):170–180, 2002. 19

[65] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996. 1, 8, 167

[66] Xiao Yu Li and Matthias F. Stallmann. New bounds on the barycenter heuristic for bipartite graph drawing. *Information Processing Letters*, 82:293–298, 2002. 10

[67] Erkki Mäkinen. Experiments on drawing 2-level hierarchical graphs. *International Journal of Computer Mathematics*, 36:175–181, 1990. 9

[68] Rafael Martí and Manuel Laguna. Heuristics and meta-heuristics for 2-layer straight line crossing minimization. *Discrete Applied Mathematics*, 2002. 9

[69] Jirí Matousek and Robin Thomas. Algorithms finding tree-decompositions of graphs. *Journal of Algorithms*, 12(1):1–22, 1991. 56, 178

[70] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 167

[71] B. Mohar. A linear time algorithm for embedding graphs in an arbitrary surface. *SIAM Journal of Discrete Mathematics*, 12(1):6–26, 1999. 15

[72] B. Monien and E. Speckenmeyer. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica*, 22:115–123, 1985. 14

[73] Xavier Muñoz, Walter Unger, and Imrich Vrt'o. One sided crossing minimization is np-hard for sparse graphs. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing, 9th International Symposium (GD 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 115–123. Springer-Verlag, 2001. 6, 158, 160, 163

[74] Petra Mutzel. An alternative method to crossing minimization on hierarchical graphs. In Stephen C. North, editor, *Graph Drawing, Symposium on Graph Drawing (GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 318–333. Springer-Verlag, 1996. 6, 13, 166, 167, 168, 169

[75] Petra Mutzel. An alternative method to crossing minimization on hierarchical graphs. *SIAM Journal of Optimization*, 11(4):1065–1080, 2001. 6, 13, 168

[76] Petra Mutzel. Optimization in leveled graphs. In Panos M. Pardalos and Christodoulos A. Floudas, editors, *Encyclopedia of Optimization*, pages 189–196. Kluwer Academic Publishers, 2001. 1

[77] Petra Mutzel and René Weiskircher. Two-layer planarization in graph drawing. In Kyung-Yong Chwa and Oscar H. Ibarra, editors, *Proceedings of the 9th International Symposium on Algorithms and Computation (ISAAC 1998)*, volume 1533 of *Lecture Notes in Computer Science*, pages 69–78. Springer-Verlag, 1998. 6

[78] Hiroshi Nagamochi. An improved approximation to the one-sided bilayer drawing. In Giuseppe Liotta, editor, *Graph Drawing, 11th International Symposium (GD 2003)*,

volume 2912 of *Lecture Notes in Computer Science*, pages 406–418. Springer-Verlag, 2003. 10

[79] G.L. Nemhauser and L.E. Trotter. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975. 16

[80] Rolf Niedermeier and Peter Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73(3–4):125–129, 2000. 136

[81] Elena Prieto. Personal communication, 2004. 158

[82] F. Roberts, J. Kratochvil, and J. Nesetril, editors. *Parameterized complexity: a framework for systematically confronting computational intractability*, volume 49. Amer. Math. Soc. Providence,RI, 1999. 14

[83] G. Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986. 18

[84] G. Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory Series B*, 63:65–110, 1995. 18

[85] Daniel P. Sanders. On linear recognition of tree-width at most four. *SIAM Journal on Discrete Mathematics*, 9(1):101–117, 1995. 56, 178

[86] M. Sarrafzadeh and C.K. Wong. *An Introduction to VLSI Physical Design*. McGraw Hill, 1996. 1, 6

[87] Petra Scheffler. A linear algorithm for the pathwidth of trees. In Rainer Bodendiek and Rudolf Henn, editors, *Topics in Combinatorics and Graph Theory*, pages 613–620. Physica-Verlag, Heidelberg, 1990. 59, 60

[88] Matthias F. Stallmann, Franc Brglez, and Debabrata Ghosh. Heuristics, experimental subjects, and treatment evaluation in bigraph crossing minimization. *ACM Journal on Experimental Algorithmics*, 2002. 10

[89] Matthew Suderman. Pathwidth and layered drawings of trees. *International Journal of Computational Geometry & Applications*, 14(3):203–225, 2004. viii

[90] Matthew Suderman and Sue Whitesides. Experiments with the fixed-parameter approach for two-layer planarization. In Giuseppe Liotta, editor, *Graph Drawing, 11th International Symposium (GD 2003)*, volume 2912 of *Lecture Notes in Computer Science*, pages 345–356. Springer-Verlag, 2003. viii, 141

[91] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981. 1, 4, 7, 9

[92] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. 51

[93] N. Tomii, Yahiko Kambayashi, and Shuzo Yajima. On planarization algorithms of 2-level graphs. Technical Report EC77-38, Institute of Electronic and Communication Engineers of Japan (IECEJ), 1977. 1, 6, 24

[94] Etsuji Tomita and Tomokazu Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In Cristian Calude, Michael J. Dinneen, and Vincent Vajnovszki, editors, *Discrete Mathematics and Theoretical Computer Science, 4th International Conference, DMTCS 2003*, volume 2731 of *Lecture Notes in Computer Science*, pages 278–289. Springer DMTCS, 2003. 14, 15

[95] Vance E. Waddle and Ashok Malhotra. An $e \log e$ line crossing algorithm for levelled graphs. In Jan Kratochvíl, editor, *Graph Drawing, 7th International Symposium (GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 59–71. Springer-Verlag, 1999. 6

[96] Michael S. Waterman and Jerrold R. Griggs. Interval graphs and maps of DNA. *Bulletin of Mathematical Biology*, 48(2):189–195, 1986. 1, 2, 4

[97] Avi Wigderson. The complexity of the hamiltonian circuit problem for maximal planar graphs. Technical Report 298, Princeton University, EECS Department, 1982. 80, 94

[98] Atsuko Yamaguchi and Akihiro Sugimoto. An approximation algorithm for the two-layered graph drawing problem. In Takao Asano, Hiroshi Imai, D. T. Lee, Shin-Ichi Nakano, and Takeshi Tokuyama, editors, *Computing and Combinatorics, 5th Annual International Conference (COCOON 1999)*, volume 1627 of *Lecture Notes in Computer Science*, pages 81–91. Springer-Verlag, 1999. 10

Index

W