Experiments with the Fixed-Parameter Approach for Two-Layer Planarization

Matthew Suderman and Sue Whitesides*

School of Computer Science, McGill University
3480 University Street, Room 318
Montreal, Quebec, H3A 2A7 CANADA
{msuder,sue}@cs.mcgill.ca

Abstract. We present computational results of an implementation based on the fixed parameter tractability (FPT) approach for biplanarizing graphs. These results show that the implementation can efficiently minimum biplanarizing sets containing up to about 18 edges, thus making it comparable to previous integer linear programming approaches. We show how our implementation slightly improves the theoretical running time to $O(6^{\mathsf{bpr}(G)} + |G|)$. Finally, we explain how our experimental work predicts how performance on sparse graphs may be improved.

1 Introduction

A layered drawing of a graph G is a 2-dimensional drawing of G in which each vertex is placed on one of several parallel lines called *layers*, and each edge is drawn as a straight line between its end-vertices. In this paper, we also require that the end-vertices of each edge lie on different layers. Layered graph drawings [16] have applications in visualization [1], DNA mapping [17], and VLSI layout [11]. In this paper, we consider 2-layer drawings which are of fundamental importance in the "Sugiyama" approach to multi-layer graph drawing [14].

One of the most studied criteria for obtaining "good" drawings of graphs is to minimize the number of edge crossings in the drawing. For 2-layer drawings, the minimum number of crossings is called the *bipartite crossing number* of a graph. Unfortunately, the 2-Layer Crossing Minimization problem, of deciding whether or not the bipartite crossing number of a given graph is at most a given constant $k \geq 0$, is NP-Hard [6]. Furthermore, in practice, exact solutions are practical for up to only 15 vertices per layer, and heuristics are extremely inaccurate for sparse graphs [8].

Some recent experimental evidence suggests that, reducing the number of edges creating crossings produces "better" drawings in some cases than minimizing the number of crossings [13]. In other words, this motivates a strategy of removing a minimum number of edges so that the resulting graph can be drawn without crossings (and then possibly re-inserting the removed edges). A graph is

* Research supported by an NSERC operating grant and an FQRNT scholarship.

G. Liotta (Ed.): GD 2003, LNCS 2912, pp. 345–356, 2004.

biplanar if it admits a planar 2-layer drawing. A set of edges whose removal from a graph G makes it biplanar is called a *biplanarizing set* for G. The *biplanarizing number* of a graph G, denoted by $\mathsf{bpr}(G)$, is the size of the minimum biplanarizing set for G. Thus, the 2-Layer Planarization problem is: given a graph G and an integer $k \geq 0$, determine whether or not $\mathsf{bpr}(G) \leq k$. This problem was first studied by Tomii *et al.* [15], who showed that it is NP-Complete. Therefore the optimization problem of finding the biplanarizing number of a graph is NP-Hard. Interestingly, Mutzel [12, 13] reports much better results for integer linear programming solutions that find the biplanarizing number of a graph than those that find its bipartite crossing number. Thus, [12, 13] provide two compelling reasons to study 2-Layer Planarization and its corresponding optimization problem.

Biplanarity has been studied from the parametric complexity view. A problem with input size n and parameter k is said to be *Fixed-Parameter Tractable*, or in the class FPT, if it can be solved in $O(f(k) \cdot n^{\alpha})$ time, for some function f and constant α (see Downey and Fellows [2]). Dujmović *et al.* [3] describe such an algorithm for solving the 2-Layer Planarization problem that runs in time $O(k \cdot 6^k + |G|)$.

In this paper, we describe an implementation based on the algorithm of [3]. Our implementation finds a biplanarizing set of size $\mathsf{bpr}(G)$. We also present experimental evidence showing that the FPT approach to biplanarization is of more than theoretical interest. In particulare, our results show that our implementation can be used in practice to efficiently find minimum biplanarizing sets containing up to about 18 edges. Furthermore, we show that its running time is roughly comparable to the well-studied integer linear programming approach. We show that our implementation runs in time $O(6^k + |G|)$, a slight improvement on the theoretical bound of [3].

Finally, we predict, on the basis of our experimental results, that a further variation of our implementation, described in Section 5, will be able to efficiently planarize sparse graphs with biplanarization numbers much larger than 18.

The rest of the paper is organized as follows. The next section defines several terms and presents previous work. Section 3 describes our implementation and its running time. Section 4 presents computational results for our implementation and compares them to those of Mutzel in [12, 13]. Finally, Section 5 describes a further variation of our implementation for sparse graphs.

2 Preliminaries

In this paper, each graph $G = (V, E)$ is simple and undirected, but not necessarily connected. A *leaf* is a vertex with exactly one neighbor, and 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 with respect to 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

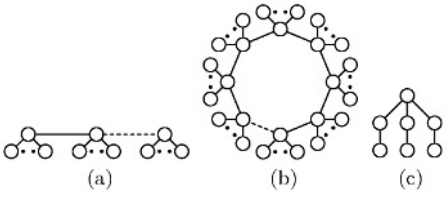

Fig. 1. (a) Caterpillar, (b) Wreath, (c) 2-Claw

consists of a vertex called the *root* that has three neighbors, and each neighbor is additionally adjacent to a leaf.

Lemma 1 ([5, 7, 15]). *Let G be a graph. The following are equivalent:*

1. *G is biplanar;*
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.*

Let $P = v_1 \dots v_p$ be a simple path of length at most 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 $v_2, \dots, v_p$ 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 $v_2, \dots, v_{p-1}$ 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 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 has a neighbor outside of the wreath, and v is on the wreath cycle[1]. The wreath subgraph is said to be *connected* to the rest of the graph at v, its *connection point*. A *pendant triangle* is a pendant wreath composed of three edges, all cycle edges. A *middle edge* of a pendant triangle is any edge whose end-vertices have degree equal to 2.

3 Algorithm Implementation

We begin by recalling the following lemma whose proof we include in order to describe our implementation.

[1] Note: a wreath component may be regarded as a pendant wreath by thinking of one of its leaves as outside the wreath subgraph.

Lemma 2 ([3]). *If there exists a vertex v in a graph G such that $deg'_G(v) \geq 3$, then v belongs to a 2-claw or a 3- or 4-cycle in G.*

Proof. Let w_1, w_2, w_3 be three distinct non-leaf neighbors of v, and let x_1, x_2, x_3 be neighbors of w_1, w_2, w_3, respectively, that are distinct from v. If $x_i = w_j$ for some i and j, then vw_jw_i is a 3-cycle. On the other hand, if $x_i \neq w_j$ for each i and j but $x_i = x_j$ for some $i \neq j$, then $vw_ix_iw_j$ is a 4-cycle. If neither of these is true, then vertices $v, w_1, w_2, w_3, x_1, x_2, x_3$ form a 2-claw rooted at v. □

We call a 2-claw or a 3- or 4-cycle in a graph a *forbidden structure*.

One approach for producing FPT algorithms is the *method of bounded search trees* [2]. The basic idea is to exhaustively search for a solution to the problem in a tree whose size is bounded by a function of the problem parameter. For intuition, we give a basic bounded search tree algorithm for solving the 2-Layer Planarization problem. Both our implementation and the algorithm of [3] elaborate on this basic idea.

We construct the search tree recursively, beginning at the root. To each node, we associate a subgraph H of G; for the root node, we have $H = G$. For each non-leaf node, we also associate a forbidden structure S in H. By Lemma 1, at least one edge in S is in every biplanarizing set of H; consequently, the current node has $|S|$ children, one corresponding to each edge in S. The subgraph associated with each child is obtained by removing an edge in S from H. A node is a leaf if its subgraph H is obtained from G by removing more than k edges, or does not contain any forbidden structures. In the second case, we have, by Lemma 2, that every vertex v in H has $\deg'_H(v) \leq 2$; in other words, each connected component in H is either a caterpillar or a wreath. By Lemma 1, any minimum biplanarizing set H contains exactly one cycle edge from each wreath in H. Thus, a leaf node represents a *yes*-instance to the problem if its subgraph H does not contain any forbidden structures, and the sum of the edges removed from H to obtain G together with the number of wreaths in H is at most k. The corresponding biplanarizing set is simply the edges removed from G to obtain H together with one cycle edge from each wreath in H.

The resulting tree has at most $O(6^k)$ nodes because, first of all, each node has at most 6 children, and secondly, each non-root node corresponds to an edge removal so the height of the tree is at most k. Constructing this tree naively requires $O(|G|)$ time at each node; therefore, we have an $O(6^k \cdot |G|)$ time algorithm for solving the 2-Layer Planarization problem.

Although this is enough to prove that the 2-Layer Planarization problem is Fixed-Parameter Tractable, the running time can be further improved. In fact, an $O(k \cdot 6^k + |G|)$ time algorithm is given in [3], roughly by reducing the graph to a "kernel" of size $O(k)$ so that at most $O(k)$ time is needed at each node.

In looking for a convenient implementation, we discovered that we could further reduce the running time to $O(6^k + |G|)$. We obtained this reduction by finding a way to determine, at each search tree node, whether or not its subgraph H contains a forbidden structure, and, if so, to exhibit one such structure, all in constant time. In addition, instead of handling component wreaths only at leaf nodes, we handle them as soon they are created by an edge-removal.

It is possible to find a forbidden structure in constant time by maintaining the list F of vertices with $\deg' \geq 3$, and, for each vertex v, the list $f(v)$ of edges incident on v that correspond to the non-leaf neighbors of v. We construct a forbidden structure in constant time as in the proof of Lemma 2. We first select the first vertex v in F, then the first three edges (v, w_1), (v, w_2) and (v, w_3) in $f(v)$, and, for each w_i, an incident edge other than (v, w_i). If these six edges induce a cycle, then we have found either a 3- or 4-cycle; otherwise, we have a 2-claw rooted at v. If F is empty, then, we are at a leaf node in the tree.

The following lemmas show that we can update these lists after each edge removal in constant time. If (v, w) is an edge, then $\mathrm{nl}(v, w)$ denotes the other neighbor $w' \neq w$ of v if $\deg(v) = 2$; otherwise, $\mathrm{nl}(v, w) = v$.

Lemma 3. *Let $e = (v_0, v_1)$ be an edge in a graph G. Let F be the set of vertices in G with $\deg' \geq 3$, and let F' the set of vertices in $G \setminus e$ with $\deg' \geq 3$. Then:*

$$F' \subseteq F \subseteq F' \cup \{nl(v_0, v_1), nl(v_1, v_0)\}.$$

Proof. After removing edge e, only vertices whose $\deg'$ decreases to 2 are removed from F. In other words, these vertices either lose a neighbor or one of their neighbors becomes a leaf. Thus, the value of $\deg'$ may change only for v_0, v_1, $\mathrm{nl}(v_0, v_1)$ and $\mathrm{nl}(v_1, v_0)$ when e is removed. If we have $v_0 \neq \mathrm{nl}(v_0, v_1)$, then $\deg(v_0) = 2$ so $v_0 \notin F$ or F'. Symmetric comments apply to v_1. □

The proof of the next lemma is similar so it is omitted.

Lemma 4. *Let $e = (v_0, v_1)$ be an edge in a graph G, and let f be the mapping from each vertex in G to the set of incident edges corresponding to its non-leaf neighbors. Similarly, let f' be the mapping from each vertex in G to the set of incident edges corresponding to its non-leaf neighbors in $G \setminus e$.*

Then, $f'(w) = f(w)$ for each vertex w in $E(G) \setminus \{v_0, nl(v_0, v_1), v_1, nl(v_1, v_0)\}$, and, otherwise:

- $f(v_0) \subseteq f'(v_0) \cup \{(v_0, v_1)\}$,
- $f(nl(v_0, v_1)) \subseteq f'(nl(v_0, v_1)) \cup \{(v_0, nl(v_0, v_1))\}$,
- $f(v_1) \subseteq f'(v_1) \cup \{(v_0, v_1)\}$, *and*
- $f(nl(v_1, v_0)) \subseteq f'(nl(v_1, v_0)) \cup \{(v_1, nl(v_1, v_0))\}$.

As mentioned earlier, we handle component wreaths at each node in the tree rather than leaving them for the leaf nodes. Furthermore, we detect and planarize each wreath in constant time. As a result, we cannot expect to detect a component wreath by traversing each of its member vertices. Instead, we rely on pointers called *cheaters*. Cheaters link the first and last vertices on the spine of every internal caterpillar. For convenience, we will think of a pendant wreath as an internal caterpillar with one connection point.

Suppose that the subgraph H of a node contains no component wreaths but that, for some edge $e = (v_0, v_1)$ in H, $H \setminus e$ contains at least one. By Lemma 3, $v'_0 = \mathrm{nl}(v_0, v_1)$ or $v'_1 = \mathrm{nl}(v_1, v_0)$ belongs to the wreath, and, for each v'_i in the wreath, we have $\deg'_{H \setminus e}(v'_i) = 2$ and $\deg'_H(v'_i) > 2$. All vertices in a component

wreath have $\deg' \leq 2$, so, of all vertices in the wreath subgraph in H, only v'_0 and v'_1 have $\deg'_H > 3$. Thus, the wreath subgraph in H is composed of zero or more internal caterpillars and possibly vertices v'_0 and v'_1 acting as connection points for these internal caterpillars. Thus, removing edge e creates a component wreath if and only if some v'_i is the single connection point for an internal caterpillar in H and $\deg'_{H\setminus e}(v'_i) = 2$, or both v'_0 and v'_1 are the connection points for two internal caterpillars and $\deg'_{H\setminus e}(v'_0) = \deg'_{H\setminus e}(v'_1) = 2$. In both cases, we planarize the wreath by removing any edge in $f(v'_i)$.

Having handled component wreaths using cheaters, we now show how to efficiently update cheaters whenever an edge is removed. If a new internal caterpillar is created by an edge removal, then the edge removal decreases the $\deg'$ of a vertex v down to two. 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. If v is a connection point for only one internal caterpillar P, then the new internal caterpillar is the concatentation 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.

Thus, we have shown how to explore the bounded search tree in $O(6^k + |G|)$ time. The resulting algorithm Bounded Search Tree for solving 2-Layer Planarization is given below. We assume that set F and the map f are correctly initialized for the graph G in $O(|G|)$ time before the algorithm is executed.

Algorithm Bounded Search Tree (graph G; vertex-set F; map f; integer k)

1. **if** $F = \emptyset$ **then return** YES;
2. **else if** $k = 0$ **then return** NO;
3. **else**
   a) $S \leftarrow$ a 2-claw, 3-cycle or 4-cycle in G using F and f;
   b) **for each** edge $(x, y) \in S$ **do**
      i. Remove (x, y) from G and planarize any resulting component wreaths while updating F, f and the cheaters. Let P be the set of removed edges;
      ii. **if** Bounded Search Tree$(k - |P|)$ **then return** YES;
      iii. **else**
         Add edges in P back to G and undo the resulting changes to F, f and the cheaters;
4. **return** NO;

We have following result:

Lemma 5. *Given a graph G and integer k, algorithm* Bounded Search Tree *determines if* bpr $(G) \leq k$ *in $O(6^k + |G|)$ time.*

We use the algorithm Bounded Search Tree to find the minimum biplanarizing set for a graph by repeatedly executing Bounded Search Tree, intially with some lower bound value for k, and then increasing k until the algorithm returns YES.

Thus, we can find the minimum biplanarizing set for a graph in $O(6^0 + 6^1 + \ldots + 6^{\mathsf{bpr}(G)} + |G|) = O(6^{\mathsf{bpr}(G)} + |G|)$ time. Variable k is initially set to either $\Phi(G)/2$ or $|E| - |V| + 1$. As defined in [3], the potential function $\Phi(G)$ denotes the following sum:

$$\sum_{v \in V(G)} \max\{\deg'(v) - 2, 0\}.$$

Lemma 6 ([3]). *For every graph G, $\mathsf{bpr}(G) \geq \frac{\Phi(G)}{2}$.*

The value of $\mathsf{bpr}(G)$ equals $|E| - |V| + 1$ whenever G has a spanning caterpillar. This is best possible for $\mathsf{bpr}(G)$ for any graph G. Thus, we obtain:

Theorem 7. *Given a graph G, there exists an algorithm that finds a minimum biplanarizing set for G in $O(6^{\mathsf{bpr}(G)} + |G|)$ time.*

In our implementation, we include three other improvements to the algorithm Bounded Search Tree described above. We obtain the first improvement by slightly generalizing the method for planarizing component wreaths at each search tree node, which we described above, to planarize all pendant wreaths as well. The generalization can be applied in constant time after each edge removal.

The second improvement is based on the observation that, if a vertex is the root of more than one 2-claw, then planarizing these 2-claws one-at-a-time could result in exploring unnecessary branches of the search tree. For example, if a vertex v has exactly four non-leaf neighbors, and each neighbor has degree equal to two, then v is the root of $\binom{4}{3}$ 2-claws. The original algorithm would typically choose one of these 2-claws, branch on it, and then, on each branch, branch on the 2-claw remaining at v. This results in trying to planarize the 2-claws rooted at v in 36 different ways. Some of these branches are redundant because there are exactly 24 different ways to planarize these 2-claws with two edge removals. Avoiding these redundant branches could lead to a substantially shorter running time when many vertices in the graph are the roots of more than one 2-claw. In fact, we end up with a branching factor of $\sqrt(24) < 5$ rather than 6.

The redundant branches are due to deleting the same set of edges but in a different order. For example, if the children of node N correspond to the edges $e_1, \ldots, e_6$ of a 2-claw, then the search subtree corresponding to removing e_1 may explore the possibility of also removing edge e_2, and, similarly, the search subtree corresponding to removing e_2 may explore the possibility of also removing edge e_1. Entirely exploring both subtrees would be redundant because, if the graph remaining at node N could have been planarized by removing both e_1 and e_2, then that solution node appears in both subtrees. Thus, it would be more efficient to completely explore the subtree corresponding to e_1, and then explore only parts of the subtree corresponding to e_2 that do not involve removing e_1. We avoid these redundant parts by marking e_1 as *tried* after we have explored its subtree and failed to find a solution. If we fail to find a solution at a descendant of node N, then, just before backtracking from N, we remove the mark on e_1. In general, then, we mark an edge as *tried* as soon as we have explored its subtree

and then remove that mark whenever we backtrack away from its parent node. The extra overhead is clearly constant per node so the resulting algorithm runs in $O(6^{\mathsf{bpr}(G)} + |G|)$ time.

The third improvement is based on the observation that we can avoid exploring subtrees that correspond to removing a *candidate* edge from the graph. As defined in [4], an edge is called a *candidate* for removal if it is the middle edge of an internal 3-path or triangle, or it does not belong to an internal 3-path or triangle and one end-vertex has $\deg' > 2$ and the other has $\deg > 1$. 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 8 ([4]). *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 the algorithm still runs in $O(6^{\mathsf{bpr}(G)} + |G|)$ time.

4 Computational Results

We implemented the algorithm described in the previous section using the Java programming language. 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 1004.542 MHz Pentium III computer with 901156 Kb 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 (Java programs are generally 2-3 times slower than similar C++ programs); therefore, we also recorded the sizes of the search trees explored. These values depend only on the input graph and the algorithm. We include the running times in our results only to give a rough idea of how long the implementation takes to planarize a graph.

We applied our implementation to the bipartite graphs from the Stanford Graphbase [10] that were used in the experiments of Mutzel [12,13]. The results of our experiments are shown alongside the results of Mutzel [13] in Table 1. Each row in the table corresponds to the average values from applying the algorithm to 100 different graphs. From Mutzel's ILP experiments, we included both the running time and the average guarantee of the solution value (Gar), i.e. $\frac{UpBound-Sol}{UpBound} \times 100$ 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 determined by the linear programming relaxation when the time limit of 300 seconds expired. It turns out that for the graphs investigated, the initial value for k, $\max(\frac{\phi(G)}{2}, |E|-|V|+1)$, is quite close to $\mathsf{bpr}(G)$. For denser graphs we would expect this value to be closer to $\mathsf{bpr}(G)$ than for sparser graphs because of the higher probability of have a spanning caterpillar. However, on

Table 1. Results for bipartite graphs with $|V_i|$ vertices per bipartition and $|E|$ edges.

		Mutzel ILP [13]		FPT			
$\|V_i\|$	$\|E\|$	Gar	Time ($\leq$ 300s)	Time ($\leq$ 600s)	Steps	bpr	Success
20	20	0.00	0	0	5	1	100
20	25	0.00	0	0	8	1.5	100
20	30	0.00	0	0	25	3	100
20	35	0.00	1	0	90	4.9	100
20	40	0.00	6	0	595	7.7	100
20	45	0.03	26	0	1,829	10.7	100
20	50	0.67	100	2	53,416	14.3	100
20	55	0.53	81	41	1,767,872	18.2	96
20	60	0.37	56	-	-	-	-
20	65	0.32	54	-	-	-	-
20	70	0.13	26	-	-	-	-
20	75	0.13	22	-	-	-	-
20	80	0.03	12	-	-	-	-
20	85	0.10	20	-	-	-	-
20	90	0.02	8	-	-	-	-
20	95	0.00	4	-	-	-	-
20	100	0.00	4	-	-	-	-
20	40	0.00	6	0	495	7.4	100
30	60	0.13	49	1	10,559	11.3	100
40	80	0.55	150	9	243,760	15.6	100
50	100	1.45	253	43	1,281,694	19.4	97

average the biplanarizing number was at most one more than our initial value for k. Thus, even our intial lower bound which could be calculated in $O(|G|)$ time had a good average guarantee of the solution value.

It would not be very meaningful to directly compare our running times to those of Mutzel because of environment differences. More specifically, they were originally reported in 1996 in [12] so the computers used were much slower than the ones we used, and the implementation language was C++ using ABACUS [9].

It is, however, meaningful to compare the shapes of the $|E|$ versus running time graphs. In the first 17 rows of Table 1, we see that the FPT implementation is quite efficient up to $|E| = 55$, finding exact solutions to all input graphs. After $|E| = 55$, the FPT implementation was able to obtain exact solutions to only a few input graphs for maximum time 10 minutes per graph. The ILP 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 FPT approaches tend to be efficient on sparse graphs, ILP approaches tend to be efficient on dense graphs. This is because FPT algorithms have running times like $O(f(k) \cdot n^{\alpha})$; therefore, they will be efficient when $\mathsf{bpr}(G)$ is small, that is, the graph is sparse. ILP algorithms using branch-and-cut, on the other hand, find an optimal solution by repeatedly finding approximate solutions

that close in on an optimal solution. For planarization, this means that an ILP algorithm begins with a biplanarizing set of size between $\mathsf{bpr}(G)$ and $|E|-|V|+1$, and then find increasingly smaller biplanarizing sets until one of size $\mathsf{bpr}(G)$ is found. For dense graphs, the probability that $\mathsf{bpr}(G) = |E| - |V| + 1$ is high, so the ILP algorithm begins with a solution close to the optimal.

Designing and implementing FPT solutions is still quite new as compared to integer linear programming, especially for graph drawing algorithms. Consequently, further work on FPT algorithms is almost sure to yield improvements. In the next section, we describe some future possibilities.

5 Future Work

One way that we might improve the performance of our implementation on larger sparse graphs is to integrate divide-and-conquer into the algorithm. For example, planarizing two subgraphs each with $\mathsf{bpr}= k$ as a single graph could use up to $O(6^{2k} + |G|) = O(36^k + |G|)$ time. If, on the other hand, it is possible to planarize them separately, then we would use $O(2 \cdot 6^k + |G|)$ time. Clearly, the second option is preferrable. In sparse graphs, we would expect this to be often possible.

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 the internal caterpillars that connect this subgraph to other p-components. Notice 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. The proof is not trivial but omitted.

Lemma 9. *If H_i, $1 \leq i \leq p$, are the p-components of a graph G, and M_i are their minimum canonical biplanarizing sets, then $\bigcup M_i$ is a minimum canonical biplanarizing set for G and $M_i \cap M_j = \emptyset$ for each $i \neq j$.*

Lemma 9 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 are able to do better than this by planarizing in such a way as to break a larger p-component into smaller p-components, and then to planarize each of them. One strategy for breaking up a p-component is to branch on forbidden structures containing cut vertices, which we would expect to find in sparse graphs.

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 what parameter 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. 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 searching 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 $O(\mathsf{bpr}(C))$ time. To compute sub-p-components, we simply apply a modification of the algorithm fory finding biconnected components in a graph. We apply the algorithm to the at most $O(\mathsf{bpr}(C))$ vertices having three or more non-leaf neighbors, skipping over internal caterpillars during graph traversal using the cheater pointers described in the previous section.

Straight-forward but tedious analysis shows that the running time of this variation of the algorithm runs in $O(6^k + |G|)$ time. It remains to be seen how well this approach will work in practice. We expect that the running time of the algorithm will differ polynomially with respect the size of sparse graphs with uniform density.

6 Conclusion

We have described the implementation and computational results of an algorithm inspired by parameterized complexity. We have shown that for computing the minimum biplanarizing sets, this algorithm has both practical as well as theoretical value. Furthermore, we have presented experimental evidence showing that it is roughly comparable with the well-studied field of linear programming for finding practical solutions to NP-hard problems. Finally, we have described one possible way to dramatically improve on the experimental results presented in this paper.

In the future, we would plan to obtain computational results from the algorithm based on p-components. We plan to perform further experiments with graphs from sources other than the Stanford Graphbase, such as from DNA-mapping applications.

One of the limitations of our implementation is that it obtains only exact solutions. We plan to investigate using FPT algorithms for finding approximation solutions.

References

1. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice-Hall (1999)

2. Downey, R.G., Fellows, M.R.: Parametrized Complexity. Springer-Verlag (1999)
3. Dujmović, V., Fellows, M.R., Hallett, M.T., Kitching, M., Liotta, G., McCartin, C., Nishimura, N., Ragde, P., Rosamond, F.A., Suderman, M., Whitesides, S., Wood, D.R.: A fixed-parameter approach to two-layer planarization. In Mutzel, P., Jünger, M., Leipert, S., eds.: Graph Drawing, 9th International Symposium (GD 2001). Volume 2265 of Lecture Notes in Computer Science., Springer-Verlag (2001) 1–15
4. Dujmović, V., Fellows, M.R., Hallett, M.T., Kitching, M., Liotta, G., McCartin, C., Nishimura, N., Ragde, P., Rosamond, F.A., Suderman, M., Whitesides, S., Wood, D.R.: A fixed-parameter approach to two-layer planarization, manuscript (2003)
5. Eades, P., McKay, B., Wormald, N.: On an edge crossing problem. In: Proceedings of the 9th Australian Computer Science Conference, Australian National University (1986) 327–334
6. Garey, M.R., Johnson, D.S.: Crossing number is NP-complete. SIAM Journal of Algebraic Discrete Methods **4** (1983) 312–316
7. Harary, F., Schwenk, A.: A new crossing number for bipartite graphs. Utilitas Mathematica **1** (1972) 203–209
8. Jünger, M., Mutzel, P.: 2-layer straightline crossing minimization: performance of exact and heuristic algorithms. Journal of Graph Algorithms and Applications **1** (1997) 1–25
9. Jünger, M., Thienel, S.: The ABACUS-system for branch and cut and price algorithms in integer programming and combinatorial optimization. In: Software–Practice and Experience. Volume 30. (2000) 1324–1352
10. Knuth, D.: The Stanford GraphBase: A Platform for Combinatorial Computing. ACM Press, Addison-Wesley Publishing Company (1993)
11. Lengauer, T.: Combinatorial Algorithms for Integrated Circuit Layout. Wiley (1990)
12. Mutzel, P.: An alternative method to crossing minimization on hierarchical graphs. In North, S.C., ed.: Graph Drawing, Symposium on Graph Drawing (GD '96). Volume 1190 of Lecture Notes in Computer Science., Springer-Verlag (1996) 318–333
13. Mutzel, P.: An alternative method to crossing minimization on hierarchical graphs. SIAM Journal of Optimization **11** (2001) 1065–1080
14. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics **11** (1981) 109–125
15. Tomii, N., Kambayashi, Y., Yajima, S.: On planarization algorithms of 2-level graphs. Papers of tech. group on elect. comp., IECEJ **EC77-38** (1977) 1–12
16. Warfield, J.N.: Crossing theory and hierarchy mapping. IEEE Transactions on Systems, Man, and Cybernetics **7** (1977) 502–523
17. Waterman, M.S., Griggs, J.R.: Interval graphs and maps of DNA. Bulletin of Mathematical Biology **48** (1986) 189–195