Every Set of Disjoint Line Segments Admits a Binary Tree

Prosenjit Bose,* Michael E. Houle,** and Godfried Toussaint*

*McGill University, School of Computer Science, Montréal, Canada
**University of Newcastle, Dept. of Computer Science, Newcastle, Australia

Abstract

Given a set of n disjoint line segments in the plane, we show that it is always possible to form a tree with the endpoints of the segments such that each line segment is an edge of the tree, the tree has no crossing edges, and the maximum vertex degree of the tree is 3. Furthermore, there exist configurations of line segments where any such tree requires degree 3. We provide an $O(n \log n)$ time algorithm for constructing such a tree, and show that this is optimal.

1 Introduction

Given a set of disjoint line segments, determining whether the set admits certain combinatorial structures has received considerable attention. One of the best-studied such structures has been the simple circuit or polygon through a set of line segments. The question of deciding whether a set of disjoint line segments admits a simple circuit is conjectured to be NP-complete since Rappaport [8] has shown that deciding whether a set of line segments allowed to intersect at their endpoints admits a simple circuit is an NP-complete problem. For certain special cases, however, polynomial-time algorithms have been obtained. Avis and Rappaport [1] gave an $O(n^4)$ time and $O(n^2)$ space algorithm to decide whether a set of disjoint line segments admits a simple monotone circuit. Rappaport, Imai, and Toussaint [9] have shown that the decision problem is $O(n \log n)$ when every line segment in the set has at least one endpoint on their convex hull (such a configuration is known as a *convexly independent* set of line segments). Although not every convexly independent set of line segments admits a simple circuit, Mirzaian [5] has shown that such a set always admits a simple polygon such that the line segments are either part of the boundary of the polygon or form internal diagonals. Mirzaian's result does not hold for arbitrary sets of disjoint line segments, as was shown by Urabe and Watanabe [12], but it is conjectured that the result is true if the line segments are afforded the additional freedom of forming external diagonals of the polygon.

The structures with which this paper is concerned are trees which span a set of disjoint line segments such that each line segment is an edge of the tree and the tree has no crossing edges — such a tree will be referred to as an *encompassing tree*. The problem of determining whether a set of line segments admits an encompassing tree was first studied by Bose and Toussaint [3], who showed that a set of disjoint line segments always admits an encompassing tree. In addition, they showed that the encompassing tree of minimum total edge length has maximum degree 7. Subsequently, Rivera-Campo and Urrutia [10] proved that a disjoint set of line segments always admits an encompassing tree with maximum degree 4.

A natural question to ask is: *Given a set of disjoint line segments, is there always an encompassing tree with maximum degree less than 4?* Figure 1 shows that there exist configurations that do not admit an encompassing tree with maximum degree 2. However, we show that a set of disjoint line segments always admits an encompassing tree with maximum degree 3 (a binary tree), and that such a tree can be computed in optimal $\Theta(n \log n)$ time.

The tree construction relies heavily on a convex subdivision of the plane induced by the set of line segments.

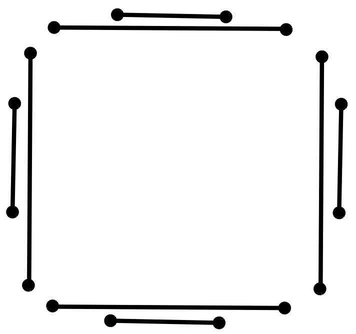


Figure 1: No encompassing tree with maximum degree 2

The construction of the subdivision is discussed in Section 2, and the special structure of the subdivision is examined in Section 3. In Section 4, it is shown how the subdivision may be used to construct an encompassing tree of degree 3 in $O(n \log n)$ time. In Section 5 we present a proof of an $\Omega(n \log n)$ lower bound for the problem. Closing remarks and open problems can be found in Section 6. Some proof are omitted in this extended abstract. For full detailed proofs, we refer the reader to the technical report [4].

2 The Convex Subdivision

To arrive at an algorithm for computing a degree-3 encompassing tree of S, we first construct a convex subdivision derived from the segments of S. Instead of subdividing the entire plane, we will find it convenient to place a bounding box around the set of line segments, and to subdivide the interior of the box into convex regions. In so doing, we ensure that the subdivision has no unbounded regions or edges.

To simplify the description of the algorithm, and to avoid degeneracies, we will assume throughout the paper that

- no segment of S is horizontal (that is, parallel to the x-axis),
- no three endpoints of segments of S are collinear.
- of the lines obtained by extending the segments to infinity in either direction, no three intersect in a common point.

The first of these assumptions is easily realized — if horizontal segments are present, a simple reorientation of the coordinate axes can be performed. The second and third assumptions can be realized using a perturbation scheme; however, we will not address these issues here.

Conceptually, the subdivision is obtained by extending each segment s along the unique line containing it. The extensions take the form of two rays, one oriented "upwards" (increasing in y-coordinate) and the other oriented "downwards" (decreasing in y-coordinate). Each ray is allowed to continue until it intersects an obstacle or another ray, at which point it is possibly truncated.

The rules governing these intersections are as follows:

1. If the intersection is determined by a ray r and an edge b of the bounding box, then r is truncated at that intersection point: it does not continue beyond b.
2. If the intersection is determined by a ray r and a segment s of S, then r is truncated at that intersection point: it does not continue beyond s.
3. If the intersection is determined by two rays r_1 and r_2 of the same orientation, then one ray is allowed to continue, and the other is truncated. Let us assume that r_2 intersects r_1 from the right (as viewed from r_1). If the rays are upward-oriented, then r_2 is truncated; otherwise, r_1 is truncated.
4. If the intersection is determined by an upward-oriented ray r_u and a downward-oriented ray r_d, then r_u is allowed to continue, and r_d is truncated.

See Figure 2 for illustrations of each of these cases.

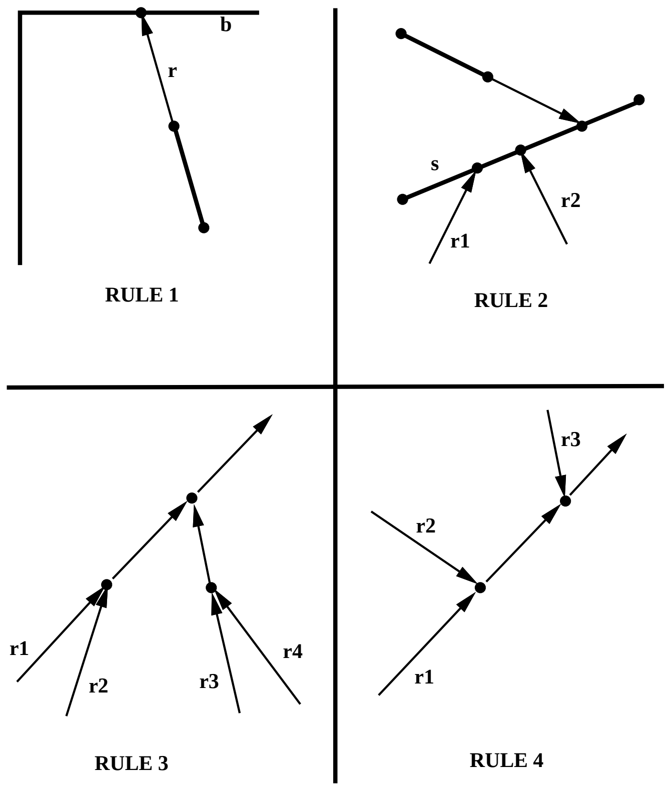


Figure 2: Extension ray intersection rules.

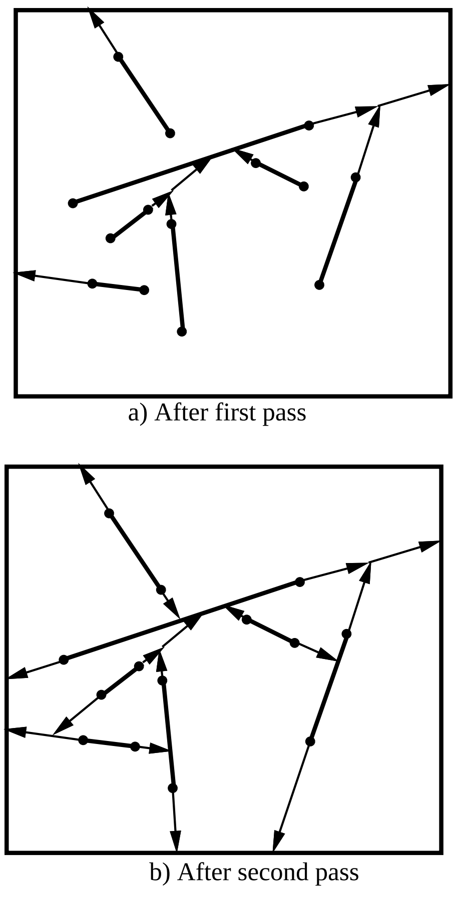
These rules are sufficient to guarantee that the resulting subdivision is convex. A vertex v of the subdivision is either an endpoint of a segment of S, a corner of the bounding box, or the truncation point of some ray, but in each of these cases, every angle incident to v (and interior to the box) is at most π by the construction. Thus every region is a polygon with no interior angle greater than π, and is thereby convex.

To construct the subdivision in an efficient manner, we make use of the well-known sweep-line paradigm. We assume that the reader is generally familiar with this paradigm, and present only a sketch of the construction here. For more information regarding sweep-line techniques, see [7].

The sweep is done in two passes: in the first pass, a horizontal line is swept from bottom to top, searching for intersections involving upward-oriented rays only — downward-oriented rays are ignored. When an intersection is detected, the appropriate rule (1, 2, or 3) is applied.

In the second pass, the downward-oriented rays are introduced. A horizontal line is swept from top to bottom, searching for intersections involving downward-oriented rays. When an intersection is detected, the appropriate rule (1, 2, 3, or 4) is applied. Note that the fourth rule guarantees that the subdivision edges introduced in the first pass are not disturbed, as these edges derive from upward-oriented rays.

An example of the subdivisions obtained after the first and second passes are shown in Figures 3a and 3b respectively.



a) After first pass

b) After second pass

Figure 3: Constructing the convex subdivision

3 Properties of the Convex Subdivision

In this section, we state and prove a number of facts concerning convex subdivisions of the kind described in the previous section. We shall also examine structures to be found within the subdivision which are central to the description of the algorithm presented in the following section, both in its motivation, and in the proof of its correctness.

We assume throughout that $\mathcal{S}$ is the subdivision for a set S of n line segments in the plane.

Lemma 1 *The number of edges, vertices, and regions of $\mathcal{S}$ is $O(n)$.*

The edges of the subdivision can be of one of three types:

- *segment edges*, which derive from segments of S,
- *extension edges*, which derive from extension rays of segments of S, and
- *box edges*, which derive from the sides of the bounding box.

Each extension edge can be thought to have an orientation, namely that of the ray from which the edge is derived. It can be classified as an *upward* extension edge or a *downward* extension edge, depending on the orientation of the ray.

Lemma 2 *Every cycle in $\mathcal{S}$ (other than the cycle forming the bounding box) contains an endpoint of some segment in S.*

Lemma 2 has an immediate implication concerning the structures formed by extension edges. Since a subdivision cannot contain cycles consisting entirely of extension edges, the graph $\mathcal{F}$ formed by the set of all extension edges must be a *forest*; that is, each connected component of the graph is a tree. We will refer to such trees as *extension trees*.

By the orientation of its incident edge, we can distinguish between two types of leaves of extension trees: those whose incident edges are directed away from the leaf, and those whose incident edges are directed towards the leaf. The former kind correspond to endpoints of segments of S; the latter kind can be formed only when an extension ray meets either the side of a segment or the bounding box. While an extension tree can have many leaves of the former kind, it turns out that it can have only one of the latter kind. We shall refer to these latter kinds of nodes as *roots* of their respective trees, reserving the term *leaf* for nodes of the former kind. The following lemma justifies the use of this terminology:

Lemma 3 *If T is an extension tree, then it has exactly one root. Furthermore, the edges along the path from any node to the root are all oriented towards the root.*

Even though the leaves of an extension tree may lie on many different segments of the subdivision, the uniqueness of the root allows us to associate each tree with either a unique segment of S, or the bounding box. Let T_1 and T_2 be extension trees rooted on the same side of a common segment s of S, and let r_1 and r_2 be their respective roots. If no other extension tree rooted on the same side of s has its root between r_1 and r_2, then we say that T_1 and T_2 are *adjacent*. In the same spirit, we say that two trees rooted on the bounding box are adjacent if it is possible to move along the bounding box from one root to the other without encountering the root of any other extension tree. See Figure 4 for examples of adjacent extension trees.

Consider a segment s of S, and the set $\mathcal{T}_s = \{T_1, T_2, \ldots, T_k\}$ of all trees rooted to one particular side of s. Let us assume that the trees of $\mathcal{T}$ are indexed in accordance with the left-to-right ordering of their roots with respect to s, as viewed from the side of s to which the trees attach. Let $(v_1, v_2, \ldots v_m)$ be the sequence of leaves one would obtain if one reported them as they were encountered during a traversal of all the trees of $\mathcal{T}$ in left-to-right order. With respect to this ordering, we say that v_i is the *left neighbour* of v_{i+1}, and v_{i+1} is the *right neighbour* of v_i (see Figure 4).

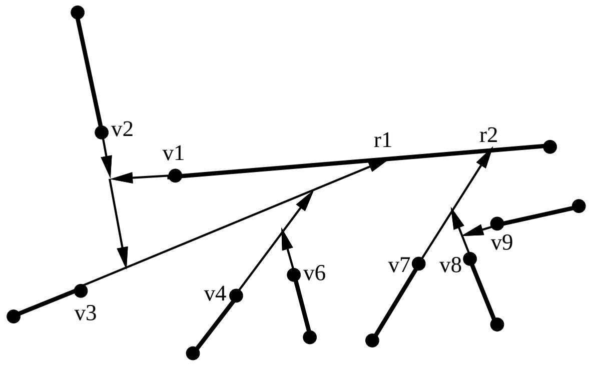


Figure 4: Two adjacent extension trees

Observation 4 *Let v and w be neighbouring leaves with respect to some segment s of S. Then there exists a path from v to w in $\mathcal{S}$ which:*

- *passes only through extension edges of trees rooted at s, or segment edges contained in s, and*
- *which is entirely contained in the boundary of some cell c of $\mathcal{S}$.*

Observation 5 *Let v be a leaf of an extension tree T_v rooted at some segment s of S. Let s_v be the left endpoint of s as viewed from the side to which the extension tree is rooted. If v has no left neighbour, then there exists a path from v to s_l in $\mathcal{S}$ which:*

- *passes only through extension edges of T_v, or segment edges contained in s, and*
- *which is entirely contained in the boundary of some cell c of $\mathcal{S}$.*

By symmetry, Observation 5 holds when v has no right neighbour and s_v is the right endpoint.

We can extend Observation 4 to the case where we consider all trees rooted at the bounding box. The only difference worth noting here is that whereas v_1 has no left neighbour and v_m has no right neighbour in the case outlined above, every leaf of a tree rooted at the bounding box always has both a left and a right neighbour.

4 Constructing an Encompassing Tree of Degree 3

The tree construction algorithm, Encompass, can best be described as incremental: starting from a single segment of S, previously unattached segments are attached to a growing tree G one by one until no unattached segments remain. When the algorithm terminates, G is the encompassing tree for S.

In the next subsection, we discuss some of the invariants and conventions observed by Encompass.

In Section 4.2 we present a key procedure of the overall algorithm, AttachTo — one which given a leaf of an extension tree, attaches to it the segment at which the tree is rooted.

Procedure AttachTo is not in itself sufficient to correctly link up all the segments into an encompassing tree of degree 3. Although the main algorithm greedily relies on AttachTo to attach as many segments as possible to the growing connected component, it sometimes occurs that segments are left unattached even after all opportunities for applying AttachTo have been exhausted. In Section 4.3, we present the procedure StitchUp which takes a cell with both attached and unattached segments in its boundary, and attaches to G those segments which AttachTo could not find.

In Subsection 4.4, we present the main algorithm, as well as its complexity analysis, and a proof of correctness.

4.1 Preliminaries

Algorithm Encompass accepts as its input a set of segments S and returns an encompassing tree G of degree at most 3. Whenever in the course of the execution of the algorithm an edge of G is created between two segment endpoints v and w, we shall say that a *bridge* has been created between v and w.

The Encompass algorithm maintains the following invariants as regards the creation of bridges:

- A bridge is added only between two mutually visible endpoints.

- Each bridge added to the encompassing tree passes through the interior of exactly one cell, from one segment endpoint on its boundary to another segment endpoint on the boundary.
- Each endpoint can have at most two bridges attached to it — one through each of the two cells whose boundaries contain it.
- A bridge is never created between two endpoints so as to introduce a cycle into G.

During the execution of the algorithm, as vertices are visited and bridges created, the segments, segment endpoints, and cells of $\mathcal{S}$ will acquire various labels. Segments can be labeled *attached* or *unattached*, depending on whether the segment has yet been connected to the encompassing tree G. Initially, all segments are *unattached*.

Segment endpoints can be labeled *unvisited*, *pending*, or *examined*. An endpoint is *unvisited* if its segment has not yet been attached. Otherwise, if it is a candidate leaf from which to apply AttachTo, then it carries the label *pending*. An endpoint labeled *examined* is one from which a call to AttachTo is no longer necessary. Initially, all segment endpoints are *unvisited*.

The labels of the cells of $\mathcal{S}$ depend on the labels of the endpoints of segments contained in its boundary. If these labels are all *unvisited*, then the cell is also labeled *unvisited*. If the endpoints are all *examined*, this implies that all endpoints in the boundary of the cell are mutually connected by the encompassing tree, and thus the cell acquires the label *connected*. A cell which is neither *connected* nor *unvisited* (that is, only "partly" connected) is labeled *pending*. Initially, all cells are *unvisited*. When all cells become *connected*, all segments are in G.

In the descriptions to come, the labels of cells are not explicitly mentioned. We assume that every time an endpoint label is modified, the labels of the two cells upon which it borders are updated in accordance with the new label. This can be done simply by maintaining an appropriate counter for each cell.

4.2 Connecting the Leaves of Extension Trees

Under the assumption that no two segments are collinear, Observation 4 implies that subject to other restrictions (such as the invariants outlined in the previous subsection), a bridge can always be created between any two neighbours v and w — unless v and w are opposite endpoints of the same segment of S, in which case no bridge is necessary. If endpoint v has no left neighbour, then by Observation 5 a bridge can be created between v and the left endpoint of the segment to which the extension tree of v is rooted (and similarly if v has no right neighbour). Algorithm Encompass takes advantage of this by means of its procedure AttachTo.

Procedure AttachTo(x, *dir*) accepts a leaf x of an extension tree which is already contained in G, and a direction *dir* ("left" or "right"). If T_x is the extension tree of which x is a leaf, then the behaviour of AttachTo depends on whether T_x is rooted at some segment s_x of S, or at the bounding box. In the former case, AttachTo first checks whether s_x is already *attached* — if so, nothing further is done. If not, then AttachTo traverses the trees attached to s_x towards one of the endpoints of s_x (determined by *dir*), linking the leaves when necessary as it goes along. This process is guaranteed to work, since each of the trees traversed on the way to the targeted endpoint of s_x are all rooted at s_x. Moreover, if one of the leaves in the sequence (call it y) is already part of G, then instead of bridging to it (and introducing an unwanted cycle into G), the procedure simply procedes onward from y without creating a bridge. The facts that y is already connected, and that the extension tree T_y of y is also rooted at s_x, together ensure that s_x will eventually be attached.

Once segment s_x has been attached (say at its endpoint λ_x), AttachTo is called again starting from λ_x. To avoid creating two bridges at λ_x in the same cell of $\mathcal{S}$, the direction of the linking is reversed. For example, if the call AttachTo(x, *left*) resulted in s_x being linked to x, then the call AttachTo(λ_x, *right*) would be performed.

In the case where T_x is not rooted at a segment of S, but instead is rooted at the bounding box, the behaviour of AttachTo is somewhat different. If AttachTo is called when no segments have yet been *attached*, the circular nature of the list of neighbours results in the connection of the entire list. Once the starting point is reached, the process terminates.

AttachTo(x, dir)

(1) If x has already been marked *examined*, then return. Otherwise, mark x as being *examined*.

(2) Let T_x be the extension tree of which x is a leaf, and let r_x be the root of T_x.

(3) If T_x is rooted at an *attached* segment, then return.

(4) If x has no neighbouring leaf in the direction *dir*, then:

  (4a) (T_x must be rooted at an *unattached* segment.) Let s_x be the segment at which T_x is rooted. Let w be that endpoint of s_x which lies in direction *dir* from r_x as viewed from x.

  (4b) Create a bridge between x and w. Mark s_x as *attached*. Mark the endpoint of s_x opposite to w as *pending*.

  (4c) Let *oppdir* be the direction opposite to *dir*. AttachTo(w, *oppdir*).

(5) Else, x has a neighbouring leaf in the direction *dir*:

  (5a) Let y be the neighbouring leaf of x in the direction *dir*.

  (5b) If y is *unvisited*, then:

    (5b1) Let s_y be the segment of which y is an endpoint.

    (5b2) Create a bridge between x and y. Mark s_y as *attached*, and the endpoint of s_y opposite to y as *pending*.

  (5c) AttachTo(y, *dir*).

4.3 Stitching Up Cells

Procedure AttachTo is not in itself sufficient to link all segments into a tree of degree 3. Even after applying AttachTo to all endpoints currently in G, it may happen that some cell of $\mathcal{S}$ may not yet be *connected*. In situations where AttachTo cannot be applied, the procedure to be outlined in this subsection takes over. Since this procedure, StitchUp, relies heavily upon special properties of subdivisions for which AttachTo cannot be applied, we shall describe StitchUp without worrying about its applicability at this stage. Its applicability and usefulness will be established after the overall algorithm has been described.

Procedure StitchUp is called upon cells which are labeled *pending*; that is, it has endpoints of both *attached* and *unattached* segments in its boundary. If c is such a cell, the goal of StitchUp is simply described: to attach all *unattached* segments having both endpoints lying on the boundary of c. How it does this depends on whether or not a bridge had previously been created within c.

If a bridge does exist, it is deleted, and replaced by new bridges threading from one endpoint of the deleted bridge, through the endpoints of the aforementioned *unattached* segments, and back to the other endpoint of the deleted bridge. The connectivity of the endpoints of the deleted bridge has in a sense been "diverted" through the new bridges and segments (see Figure 5).

If no bridge exists, then one is created between two consecutive endpoints (one on an *attached* segment, the other on an *unattached* segment). The same technique is then applied.

Once the new segments have been attached, AttachTo is called at their endpoints. This is done here instead of elsewhere so that AttachTo does not at some later stage attempt to make a connection across the interior of c.

StitchUp(c)

(1) Let $W = \{w_0, w_1, w_2, \ldots, w_{k-1}, w_k\}$, where $w_0 = w_k$, be the sequence of segment endpoints encountered as one traverses the boundary of c in clockwise order.

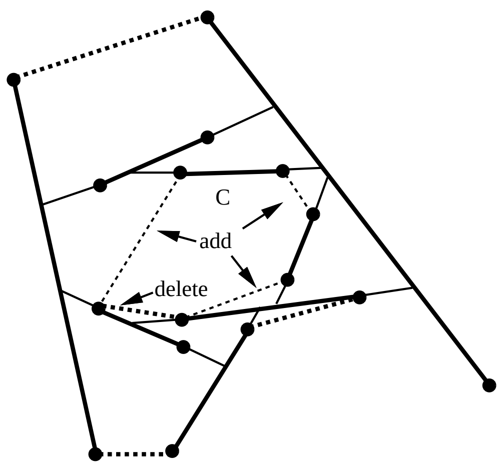


Figure 5: Stitching up cell c

(2) If no two endpoints on the boundary of c are connected by a bridge in c, then:

(2a) If so, there must be some j between 0 and $k-1$ such that endpoint w_j is *examined* and w_{j+1} is *unvisited.*

(2b) Connect w_j and w_{j+1} with a bridge. If s' is the segment of which w_{j+1} is an endpoint, then mark the other endpoint of s' as *pending.* Mark the segment itself as *attached.*

(2c) AttachTo(w_{j+1}, *right*).

(2d) If c has become *connected* as a byproduct of the previous steps, then return.

(3) Two endpoints in the boundary of c are connected by a bridge $b = (w_i, w_{i+1})$ in c, for some i between 0 and $k-1$. i may or may not be the index j from the previous step.

(3a) Let $\{s_1, s_2, \ldots s_u\}$ be the sequence of *unattached* segments entirely contained in the boundary of c, listed in clockwise order about the interior of c, and beginning at the bridge b (we shall later prove that there always exists at least one such segment). Let λ_l and ρ_l be the left and right endpoints of segment s_l, respectively.

(3b) Replace bridge b with the bridges (w_{i+1}, λ_1), (ρ_u, w_i), and (ρ_l, λ_{l+1}) for all l between 1 and $u-1$.

(3c) For all l between 1 and u, mark s_l as *attached*, and mark λ_l and ρ_l as *pending.*

(3d) For all l between 1 and u, AttachTo(λ_l, *right*) and AttachTo(ρ_l, *left*).

4.4 The Main Algorithm

Having described the procedures AttachTo and StitchUp, we are now in a position to outline the main algorithm. Following this, we shall prove that it is correct.

Encompass

(1) From the set of segments S, build its associated convex subdivision using the method outlined in Section 2.

(2) Mark each segment as *unattached.* Mark each segment endpoint and each cell *unvisited.*

(3) Connect the leaves of all trees rooted at the bounding box, as follows:

(3a) Choose such a leaf (call it x). Let s_x be the segment of which x is an endpoint.

(3b) Mark s_x as *attached*, and the endpoint of s_x opposite to x as *pending.*

(3c) AttachTo(x, *right*).

(4) While there are endpoints yet *pending* do:

(4a) Choose such an endpoint (call it y).

(4b) AttachTo(y, *right*).

(5) If there are cells yet *pending* then:

(5a) Choose such a cell (call it c).

(5b) StitchUp(c).

(5c) Go to Step (4).

Lemma 6 *Let v be an endpoint whose extension tree T_v is rooted at a segment s_v of S. A call to* AttachTo *on v always results in s_v being correctly linked to G.*

Proof Follows from the discussion of Section 4.2, and from the observation that when called from StitchUp, it is always given the correct direction from v. □

Lemma 7 *Let v be an endpoint whose extension tree T_v is rooted at a segment s_v of S. At the time when Step (4) terminates, if v is labeled as* examined*, then s_v must be labeled* attached.

Proof Follows immediately from Lemma 6. □

Lemma 8 *Whenever* StitchUp(c) *is called, there is always at least one* unattached *segment with both endpoints lying in the boundary of c.*

Lemma 9 *Let c be a cell labeled* pending *upon which* StitchUp *is called.* StitchUp *correctly attaches to G all* unattached *segments fully contained in the boundary of c.*

Proof Follows from the discussion in Section 4.3, and from Lemma 8. □

Lemma 10 Encompass *eventually terminates after taking at most $O(n \log n)$ time.*

Lemma 11 *When the execution of* Encompass *terminates, then G is a degree-3 encompassing tree for S.*

Theorem 12 *Given a set S of n disjoint line segments in the plane,* Encompass *computes a degree-3 planar encompassing tree of S in $O(n \log n)$ time and $O(n)$ space.*

Proof Follows from Lemmas 1, 6, 9, 10, and 11. □

5 Lower Bound

Finally, we show that the problem of finding a degree-3 encompassing tree of a set of disjoint line segments requires $\Omega(n \log n)$ time to solve. This implies the optimality of Algorithm Encompass.

Theorem 13 *The problem of sorting n real numbers is $O(n)$ transformable to the problem of finding a degree-3 spanning tree of a set of disjoint line segments; thus, finding a degree-3 encompassing tree of a set of disjoint line segments requires $\Omega(n \log n)$ time.*

Proof Given a set S of n positive real numbers, $x_1, \ldots, x_n$, we show how any encompassing tree algorithm can be used to sort them with only linear overhead. For convenience, let $s_1, \ldots, s_n$ represent the sorted order of the real numbers.

Let x_{max} represent the largest real number in S and x_{min} the smallest. Let $d = (x_{max}^2 - x_{min}^2) + (x_{max} - x_{min})$. Let $m = x_{max}^2$. All these values can be computed in linear time.

For each number x_i, we construct a corresponding line segment, l_i, i.e., we associate the number i with the line segment. The line segment l_i is constructed in the following way. One endpoint has coordinates (x_i, x_i^2). This endpoint is known as the *lower endpoint.* The other endpoint has coordinates $(x_i, d + 4m - 2x_i^2)$, and is known as the *upper endpoint.*

All of the lower endpoints lie on the parabola $y = x^2$ and all of the upper endpoints lie on the parabola $y = -2x^2 + d + 4m$. Since the parabola $y = x^2$ opens upwards and the parabola $y = -2x^2 + d + 4m$ opens downwards, the endpoints of l_{s_i} can only see the endpoints of $l_{s_{i-1}}$ and $l_{s_{i+1}}$. Observe that any encompassing tree of the set must consist only of the line segment edges and visibility edges between adjacent line segments; otherwise, planarity is not preserved.

Since the degree-3 encompassing tree consists of visibility edges between line segments together with the given line segments, a simple depth-first traversal of the tree starting from the leftmost vertex of degree 1 enables us to uncover the sorted order of the input in linear time from the output delivered by any algorithm. □

6 Conclusion

In this paper, we have shown that a set of disjoint line segments always admits an encompassing tree with maximum vertex degree 3, and that there exist configurations of line segments such that any emcompassing tree of the set has maximum degree 3. We presented an algorithm to compute a binary encompassing tree in $O(n \log n)$ time, and showed a lower bound of $\Omega(n \log n)$ for the problem establishing the optimality our algorithm. There are a number of open problems still to be considered.

1. Is it NP-Hard or NP-Complete to compute a simple polygon or a simple hamiltonian path through a set of disjoint line segments? Rappaport [8] has shown that it NP-Complete when the line segments can intersect at their endpoints.
2. Is it possible to compute a simple polygon through a set of disjoint line segments, where the line segments are either part of the boundary, interal diagonals or external diagonals? Urabe and Watanabe [12] have shown that if the line segments are limited to the boundary and internal diagonals, that it is not always possible.
3. Is the visibility graph of a set of disjoint line segments hamiltonian? If not, can anything be said about the longest path in the visibility graph? Is there always a path of length $O(\sqrt{n})$?

Acknowledgements We thank Eduardo Rivera-Campo and Jorge Urrutia for fruitful discussions on the topic. We thank David Thompson providing a stimulating research environment.

References

[1] D. Avis and D. Rappaport. Computing Monotone Simple Circuits in the Plane. *Computational Morphology.* Elsevier Science, G.T. Toussaint (editor), North Holland, 1988.

[2] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* Elsevier Science, New York, New York, 1976.

[3] P. Bose and G. Toussaint. Growing a Tree from its Branches, *Proc. of the First Pacific Conference on Computer Graphics and Applications*, Vol 1, pp. 91–103, Pacific Graphics '93, Seoul, South Korea, September 1993, Also: McGill Technical Report no. SOCS 92.12, McGill University, 1992.

[4] P. Bose, M. Houle, and G. Toussaint. Every Set of Line Segments Admits a Binary Tree. McGill Technical Report, McGill University, (to appear).

[5] A. Mirzaian. Hamiltonian triangulations and circumscribing polygons of disjoint line segments. *Computational Geometry: Theory and Applications*,**2**, **1**, pp. 15-30, 1992.

[6] C. Monma and S. Suri. Transitions in Geometric Minimum Spanning Trees. Bell Communications Research Technical Report, 1991.

[7] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*, Springer-Verlag, New York, New York, 1985.

[8] D. Rappaport. Computing simple circuits from a set of line segments is NP-complete. *Proc. 3rd ACM Symposium on Computational Geometry*, Waterloo, Ontario, pp. 52-60, 1987.

[9] D. Rappaport, H. Imai, and G.T. Toussaint. Computing Simple Circuits from a Set of Line Segments. *Discrete and Computational Geometry*, **5**, **3**, pp. 289-304, 1990.

[10] E. Rivera-Campo and J. Urrutia. *Personal communication.* August 1992.

[11] X. Shen and H. Edelsbrunner. A Tight Lower Bound on the Size of Visibility Graphs, *Information Processing Letters*, **26**, pp. 61-64, 1987.

[12] M. Urabe and M. Watanabe. On a counterexample to a conjecture of Mirzaian. *Computational Geometry: Theory and Applications*, **2**, **1**, pp. 51-53, 1992.