Every Set of Disjoint Line Segments Admits a Binary Tree

Prosenjit Bose* Michael E. Houle† Godfried T. Toussaint‡

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 [12] 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 [13] have shown that the decision problem is in $O(n \log n)$ 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 [7] 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 [17], and later by Grünbaum [6], but it is conjectured that the result is true if the line segments are also allowed to form external diagonals of the polygon.

The simple circuit is not the only structure to have been investigated. ElGindy and Toussaint [5] have shown that every set of line segments can be triangulated. Later, O'Rourke and Rippel [10] proved the hamiltonicity of the visibility graph of certain restricted classes of line segments.

The structures with which this paper is concerned are trees that 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

*School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, CANADA. Email: jit@scs.carleton.ca. Research supported by NSERC.

†Basser Dept. of Computer Science, The University of Sydney, Sydney, NSW 2006, AUSTRALIA, Email: meh@cs.usyd.edu.au.

‡School of Computer Science, McGill University, Montreal, Quebec, H3A 2A7, CANADA. Email: godfried@cs.mcgill.ca.

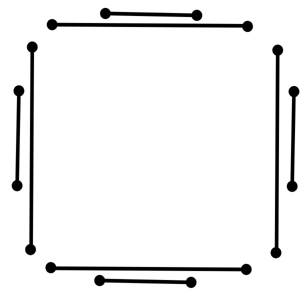

Figure 1: no encompassing tree with maximum degree 2

— 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, and that the encompassing tree of minimum total edge length has maximum degree 7. Subsequently, Rivera-Campo and Urrutia [14] 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 encompassing tree construction relies heavily on a convex subdivision of the plane induced by the set of line segments. 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.

Most of the geometric and graph theoretic terminology used in this paper is standard, and for definitions we refer the reader to O'Rourke [9], Bondy and Murty [2], and Preparata and Shamos [11].

2 The Convex Subdivision

The goal of the next three sections is to develop an algorithm to construct an encompassing tree G (as defined earlier) for a set of n disjoint line segments S. 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 in $O(n)$ time. The second and third assumptions can be realized using a perturbation scheme; however, we will not address these issues here.

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.

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; if they are downward-oriented, 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.

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 [11].

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.

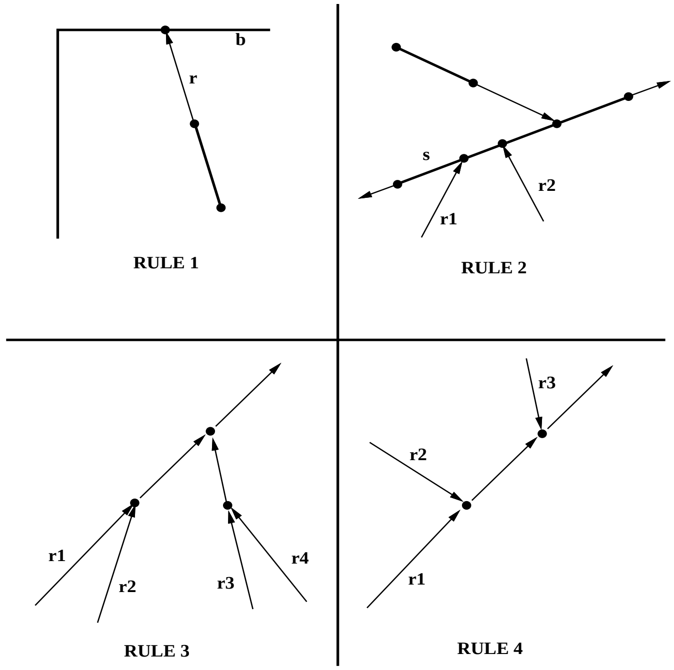

Figure 2: Extension ray intersection rules.

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.

Consider the set of line segments (and its bounding box) shown in Figure 3a. The subdivisions obtained after the first and second passes are shown in Figures 3b and 3c respectively.

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 $\mathcal{Q}$ is the subdivision for a set S of n line segments in the plane.

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

Proof Follows easily from Euler's Formula [2].

□

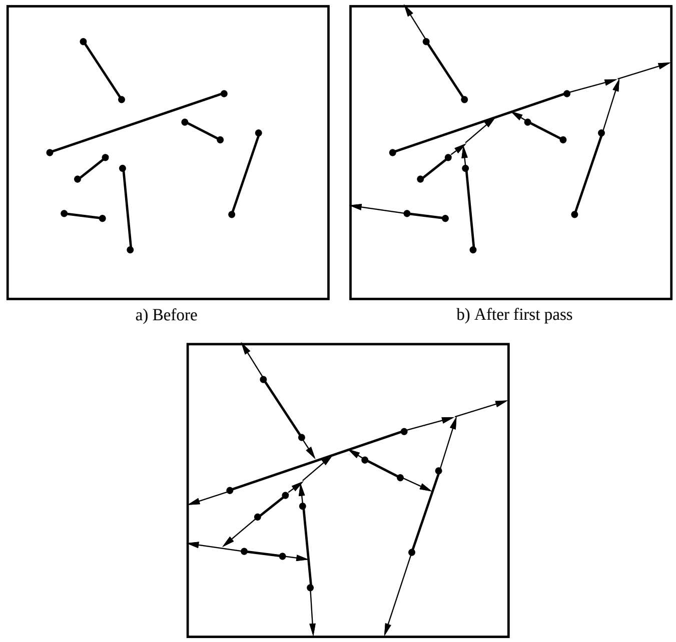

Figure 3: Constructing the convex subdivision

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

Proof Assume otherwise: that is, there exists some cycle η that does not consist entirely of box edges, and that does not contain an endpoint of any segment in S. Note that the cycle must contain at least one extension edge.

Let $\eta' = \{e_0, e_1, e_2, \ldots, e_{k-1}, e_k\}$ be the subsequence of η consisting of the extension edges of η, where $e_0 = e_k$. With respect to the ordering of η', each extension edge is oriented either *forward* or *backward*.

- Case I: the edges of η' do not all share the same orientation.

In this case, there must exist some i such that e_i is backward and e_{i+1} is forward. Clearly, e_i and e_{i+1} cannot share a common endpoint — otherwise, two rays would emanate from one point, in contravention of the rules governing intersections (3 and 4). This implies that there must be at least one non-extension edge between e_i and e_{i+1} in η. Let e be the non-extension edge occurring immediately before e_{i+1} in η.

Let v be the vertex of $\mathcal{Q}$ where e meets e_{i+1}. Vertex v cannot lie on a box edge, since no extension ray can emanate from the side of the bounding box. Therefore e must be a segment edge. However, v must then be an endpoint of the underlying segment s in S, since no extension ray can emanate from the side of s. This contradicts the assumption.

- Case II: the edges of η' all share the same orientation.

  Without loss of generality, we can assume that the extension edges are all forward edges. The arguments of Case I imply that the cycle must consist entirely of extension edges — that is, $\eta = \eta'$, and the cycle is the sequence η itself. The edges of η' therefore cannot be all upward; otherwise, each vertex in the cycle would have y-coordinate strictly greater than its predecessor, which is impossible. Similarly, the edges of η' cannot be all downward. Therefore there exists some j such that e_j is upward and e_{j+1} is downward.

  Let v' be the vertex where e_j meets e_{j+1}. The edge e_{j+1}, being forward, is oriented away from v'. Therefore e_j (and not e_{j+1}) was on the ray that was truncated at v'. But this contravenes the fourth intersection rule of Section 2, by which the downward-oriented ray containing e_{j+1} should have been truncated instead. Thus no simple cycle may have its extension edges share a common orientation. □

Lemma 2 has an immediate implication concerning the structures formed by extension edges. Let $\mathcal{F}$ be the subgraph of $\mathcal{Q}$ induced by the extension edges of $\mathcal{Q}$. Since $\mathcal{Q}$ cannot contain cycles consisting entirely of extension edges, $\mathcal{F}$ must be a *forest*; that is, each connected component of $\mathcal{F}$ 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.*

Proof According to the rules governing the intersections of extension rays, each internal node of the tree has exactly one outgoing edge. From any starting node x, let us consider the set of nodes reachable from x via a sequence of outgoing edges. Since T has no cycles, and is finite, this sequence must describe a unique path in T of finite length, oriented towards the terminus. Since

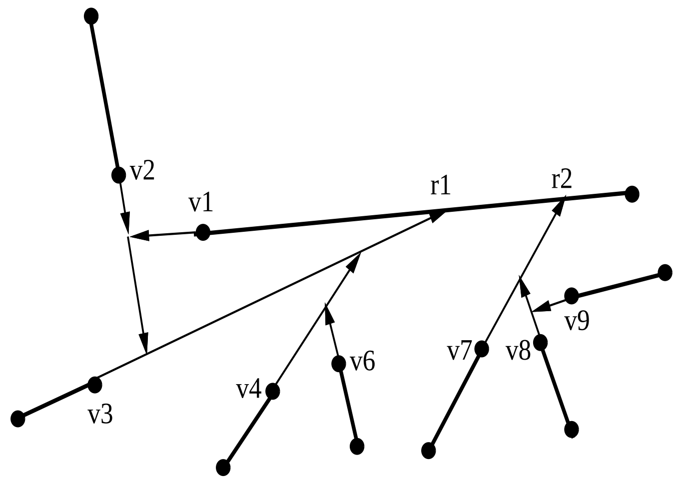

Figure 4: Two adjacent extension trees

the definition states that leaves are incident to outgoing edges, and roots to incoming edges, the terminus of this path can only be a root. This root is unique, since every internal node can have only one outgoing edge. □

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 an example of adjacent extension trees.

Consider a segment s of S, and the set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ 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 toward s from the side 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 an inorder 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}, that v_{i+1} is the *right neighbour* of v_i, and that v_i and v_{i+1} are *neighbouring* leaves (see Figure 4).

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 Q that:*

- *passes only through extension edges of trees rooted at s, or segment edges contained in s, and*
- *that is entirely contained in the boundary of some cell c of Q.*

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_v in $\mathcal{Q}$ that:*

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

Note that v can be identical to s_v, in which case the extension tree of which s_v is a leaf has its root at s. By symmetry, Observation 5 holds when v has no right neighbour and s_v is the right endpoint.

Observation 4 extends 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.

We conclude the discussion of the properties of the convex subdivision with the following lemma, that shows that all segments of S can be connected simply by ensuring that for every cell c, the segments on the boundary of every cell c are mutually connected.

Lemma 6 *Let S be a set of n disjoint line segments, and $\mathcal{Q}$ be its underlying convex subdivision. Let G be any planar graph whose vertex set is the set of endpoints of the segments in S and whose edge set includes the segments of S. Then G is connected if and only if for every cell c of $\mathcal{Q}$, the set of segment endpoints on the boundary of c is connected in G.*

Proof If G is connected, then trivially the set of segment endpoints on the boundary of any given cell are mutually reachable in G.

If the segment endpoints on the boundary of every cell c are mutually connected in G, then the fact that G is connect follows from Lemma 2 (i.e. every cycle in $\mathcal{Q}$ must contain an endpoint of S) and the fact that the planar dual [2] of $\mathcal{Q}$ is connected. □

4 Constructing an Encompassing Tree of Degree 3

The degree-3 encompassing 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 that takes a cell with both attached and unattached segments in its boundary, and attaches to G those segments that 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* (v, w) has been created between v and w.

The Encompass algorithm maintains the following invariants regarding 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 of the subdivision $\mathcal{Q}$, 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 sharing the endpoint in their common boundary.
- 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 $\mathcal{Q}$ will acquire various labels. The labels also respect certain invariant conditions, outlined below.

A segment can be labeled *unattached*, in which case it has not yet been bridged to any other segment; *attached*, which indicates that it has been integrated into the final encompassing tree G; and *semi-attached*, which indicates that it has been connected to other segments by means of bridges, but has not yet been integrated into the final encompassing tree. *Semi-attached* segments are labeled with the name of a connected component into which it has been integrated. Initially, all segments are *unattached*. Once a segment becomes *semi-attached*, it will never again become *unattached*. Once it becomes *attached*, it will always remain *attached*. The bounding box as a whole will sometimes be treated as if it were a segment. It is initialized with the label *unattached*, and will eventually receive the label *attached*.

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

AttachTo is no longer necessary. Initially, all segment endpoints are *unvisited*. Once an endpoint becomes *pending*, it will never again become *unvisited*. Once it becomes *examined*, it will always remain *examined*.

The labels of the cells of $\mathcal{Q}$ depend on the labels of the segments having endpoints contained in its boundary. If these labels are all *unattached*, then the cell is labeled *unvisited*. If the segments are all *attached*, 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 that is neither *connected* nor *unvisited* (that is, only "partly" connected) is labeled *pending*. Initially, all cells are *unvisited*. Once a cell becomes *pending*, it will never again become *unvisited*. Once it becomes *connected*, it will always remain *connected*. When all cells become *connected*, all segments are in G.

In the descriptions to come, the labels of cells are often not explicitly mentioned. We assume that every time a segment label is modified, the labels of the two cells upon which it borders are updated in accordance with the new segment 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 neighbouring leaves 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 that is already contained in some connected component (that is, either *semi-attached* or *attached*), 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^* of S, or at the bounding box. In the former case, AttachTo only proceeds if s^* is *unattached* by traversing the trees rooted at s^* towards one of the endpoints of s^* (determined by *dir*), linking the leaves when necessary as it goes along. This process is guaranteed to reach the targeted endpoint of s^*, since each of the trees traversed are all rooted at s^*.

The manner in which a leaf is linked depends on the labeling of the segment of which it is an endpoint. Let a be the current leaf in the sequence, belonging to component G_a, and let b be the next leaf in the sequence. Let s_a and s_b be the segments of which a and b are endpoints, respectively. If s_b is *unattached*, AttachTo introduces a bridge between a and b, integrates s_b into G_a by assigning it the same label as s_a, and then continues the procedure from b.

If s_b is not *unattached*, then it belongs to some connected component G_b. If $G_b = G_a$, then instead of bridging from a to b (and introducing an unwanted cycle into $G_b = G_a$), the procedure simply proceeds onward from b without creating a bridge.

If $G_b \neq G_a$, then the introduction of a bridge from a to b forces the two components to be merged. If $G_b = G$, then all segments of G_a are immediately relabeled to that of G, namely *attached*. Similarly if $G_b = G$, all segments of G_b are immediately relabeled to *attached*. If neither G_b nor G_a equals G, then the two components are merged. In all three cases, the procedure continues from b.

Whenever two components other than G are to be merged, it would be inefficient to explicitly relabel the segments of one component to match that of the other; if this is done, a given edge could potentially be relabeled many times. Instead, an efficient set union-find data structure $\mathcal{U}$ is used to keep track of equivalence classes of segment labels. Merging components is thus a matter of merging classes of labels. The explicit relabeling that occurs when a component is merged with G can only be done once per edge — once a segment receives the label *attached*, its label will never change again.

The procedure by which the leaves of neighbouring extension trees are linked finishes with the initial leaf x and s^* in the same connected component; several components may have been merged with each other or into G in the process. Once segment s^* has been attached (say at its endpoint λ), AttachTo is called again starting from λ. To avoid creating two bridges at λ in the same cell of $\mathcal{Q}$, the direction of the linking is reversed. For example, if the call AttachTo(x, *left*) resulted in s^* being linked to x, then the call AttachTo(λ, *right*) would be performed.

If λ itself is the last extension tree leaf, then by attaching λ, s^* is attached. In this case, since extension tree T_λ is rooted at the previously-visited segment s^*, no further call to AttachTo is made from λ.

Figure 5 illustrates the process by which segments are attached by showing the bridges created as a result a call to AttachTo(x, *right*), assuming the prior creation of bridge b. Note that in this example, the sequence of calls to AttachTo terminates at a node y which is simultaneously the target endpoint of its segment, and the last of the leaves of the extension trees rooted at its segment.

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 all segments are yet *unattached*, 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. For an example of how AttachTo handles this special case, see Figure 6.

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. Let s^* be the segment at which T_x is rooted.

(3) If s^* is *attached* or *semi-attached*, then return.

(4) (s^* must be an *unattached* segment.)
If x has no neighbouring leaf in the direction *dir*, then:

(4a) Let w be that endpoint of s^* which lies in direction *dir* from r_x as viewed from x. If $x = w$, then return.

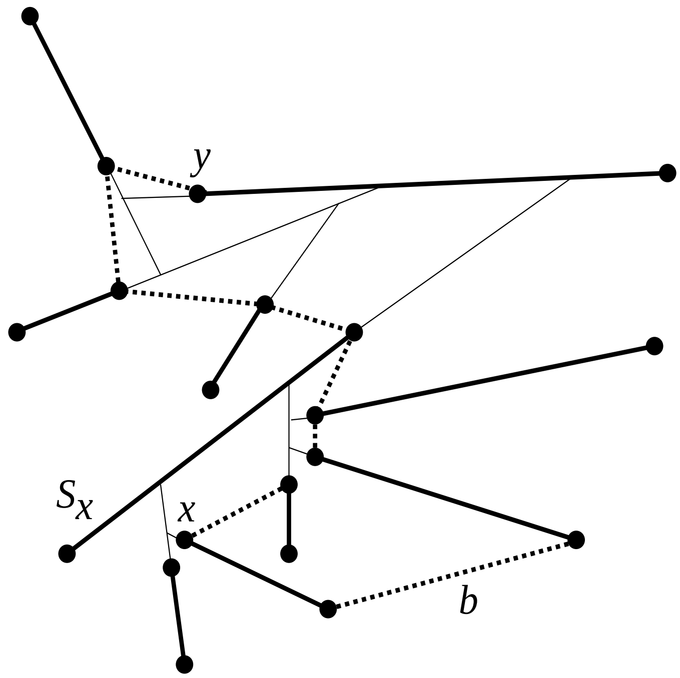

Figure 5: Bridges created by AttachTo(x, *right*)

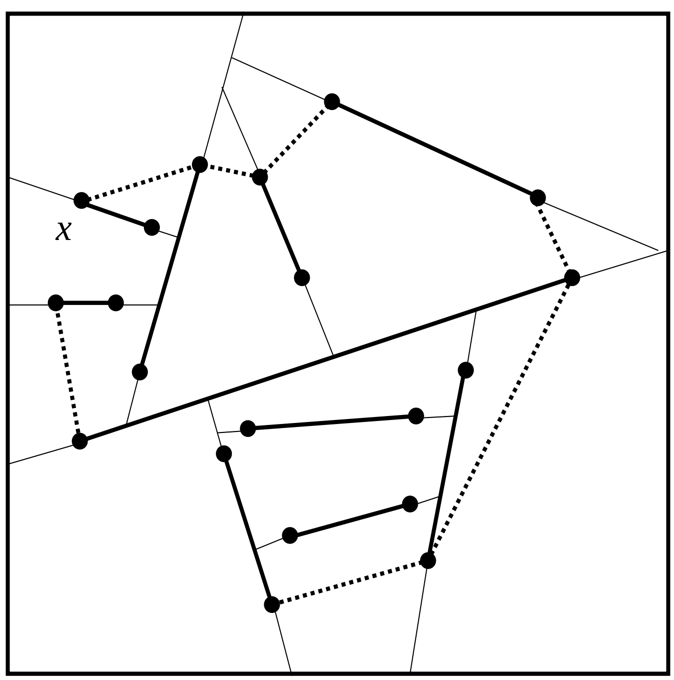

Figure 6: Bridges created by AttachTo(x, *right*)

(4b) Otherwise:

(4b1) Create a bridge between x and w. Mark s^* with the label of s_x. Mark the endpoint of s^* opposite to w as *pending*.

(4b2) Let *oppdir* be the direction opposite to *dir*. AttachTo(w, *oppdir*).

(5) Else, x has a neighbouring leaf y in the direction *dir*. Let s_y be the segment of which y is an endpoint.

(5a) If s_y is *unattached*, then create a bridge between x and y. Mark s_y with the label of s_x, and the endpoint of s_y opposite to y as *pending*.

(5b) Otherwise, s_y is *attached* or *semi-attached*. If the component of s_y is different to that of s_x, then:

(5b1) Create a bridge between x and y.

(5b2) If s_x is *attached* then relabel all segments of the connected component containing s_y as *attached*.

(5b3) Otherwise, if s_y is *attached* then relabel all segments of the connected component containing s_x as *attached*.

(5b4) Otherwise, both s_x and s_y are *semi-attached*. Merge the components containing s_x and s_y, by making their labels equivalent to each other within the union-find structure $\mathcal{U}$.

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

It should be noted at this point that AttachTo maintains each of the invariants listed in Subsection 4.1. In particular, the introduction of the bridge at Step (5b1) does not violate the invariant relating to the number of bridges that may be attached at a particular endpoint: if y already had a bridge attached to it, the endpoint would have had the label *examined* — in which case, the procedure AttachTo would have been called at y before, that would have resulted in s^* already having been labeled *attached* or *semi-attached*. A formal inductive proof of correctness will be given later in the paper.

4.3 Stitching Up Cells

Procedure AttachTo is not in itself sufficient to link all segments into a tree of degree 3. Even if AttachTo is applied such that no more endpoints are *pending*, some segments may still be *unattached*, and some cells of $\mathcal{Q}$ may not yet be *connected* (see Figure 7 for an example). In these 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 that are labeled *pending*: to be precise, those that have endpoints of both *attached* and *unattached* segments in their boundaries. If c is such a cell,

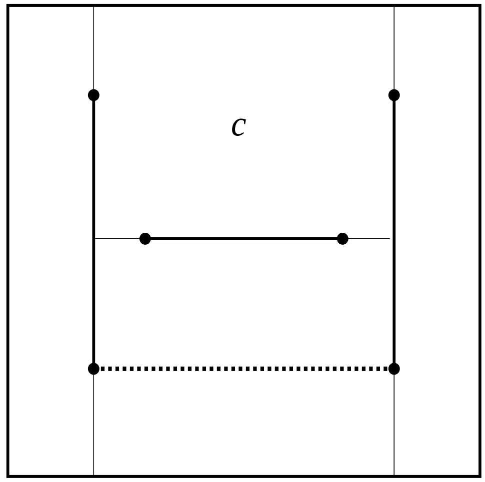

Figure 7: No more endpoints are *pending*, but c is not *connected*

the effect of calling StitchUp is to attach all *unattached* segments having one or both endpoints lying on the boundary of c.

This is done in three phases. In the first phase, a clockwise scan is performed around the boundary of c, starting from an endpoint w_0 that is guaranteed to belong to an *attached* segment. When the scan encounters an *unattached* segment with both endpoints on the boundary of c, it initiates a call to AttachTo in one of two ways, depending on the number of consecutive *unattached* segments encountered leading up to the current segment.

Once the first invocation of AttachTo has terminated, a number of previously-*unvisited* endpoints may become *pending*. Calls to AttachTo are then initiated from each *pending* endpoint, until no further *pending* endpoints remain. The result of this process is the creation of a connected component consisting of *semi-attached* segments and bridges. The scan then progresses to the next segment with both endpoints in the boundary of c, and the process is repeated to yield another connected component.

It shall be shown later that when this scan has terminated, each of the previously-*unattached* segments on the boundary of c will have been integrated into a connected component. As a result of the action of AttachTo, some components may have merged with each other, or even with the original component of *attached* edges. Furthermore, every surviving *semi-attached* component shall be shown to contain at least two endpoints on the boundary of c that are not incident to any bridges passing through c.

In the second phase of the StitchUp procedure, a clockwise scan is again performed, this time to identify endpoints of *semi-attached* components not incident to bridges through c. When two consecutive such endpoints are discovered from different *semi-attached* components, a bridge is introduced, thereby merging the components. Once the scan is complete, a single *semi-attached* component remains.

In the final phase, the remaining *semi-attached* component (call it G') is integrated into the *attached* component G, in one of two ways. If the endpoint w_0 is incident to no bridge passing through c, then w_0 may safely be bridged to either of two endpoints of G' (call them λ and ρ) identified as incident to no bridges passing through c. Otherwise, if a bridge through c is incident to w_0, it is replaced by two new bridges connecting G to G'. The connectivity of the endpoints of the deleted bridge is in a sense "diverted" through the new bridges and segments of the *semi-attached* component.

Once StitchUp has terminated for cell c, all segments are again labeled either *attached* or *unattached*, and again no endpoint of a segment of S is left with a label of *pending*. We claim, and shall prove later, that all segments on the boundary of c that were previously *unattached* have become *attached* as a result of the call to StitchUp.

StitchUp(c)

(1) Let $W = \{w_0, w_1, w_2, \ldots, w_{k-1}, w_k\}$ be the sequence of segment endpoints encountered as one traverses the boundary of c in clockwise order, where $w_0 = w_k$ is the first endpoint on the boundary of c to have been marked by the algorithm as *examined*. Let s_i be the segment of which w_i is an endpoint, for all $0 \leq i \leq k$.

(2) Initialize the union-find structure $\mathcal{U}$ to recognize equivalence classes within the set of labels $\{1, \ldots, k-1\}$. Each label is initially in its own equivalence class.

(3) For $i \longleftarrow 0$ to $k-1$, do the following:

 (3a) If segment s_i is *attached* or *semi-attached*, then set $a \longleftarrow i$.
 (a stores the most recently encountered *attached* or *semi-attached* endpoint.)

 (3b) If s_i is *unattached* and $s_i = s_{i+1}$, then:

  (3b1) (s_i has both endpoints on the boundary of c.)
  Mark s_i as *semi-attached* with component label i — segment s_i is the first in a new connected component. Mark w_i and w_{i+1} as *pending*.

  (3b2) If $i - a$ is even, then initiate AttachTo(w_i, *right*).

  (3b3) If $i - a$ is odd, then initiate AttachTo(w_{i+1}, *left*).

  (3b4) While there are endpoints yet *pending*, choose such an endpoint (call it y), and initiate AttachTo(y, *right*).

(4) Set $\lambda \longleftarrow \rho \longleftarrow \emptyset$.

(5) For $i \longleftarrow 0$ to $k-1$, do the following. If s_i is *semi-attached* and w_i has no bridge attached to it passing through cell c, then:

 (5a) If $\lambda = \emptyset$ then set $\lambda \longleftarrow i$.

 (5b) Otherwise, if $\rho = \emptyset$ then set $\rho \longleftarrow i$.

 (5c) Otherwise, if s_i and s_ρ are in different *semi-attached* connected components, then:

  (5c1) Introduce a bridge between s_ρ and s_i through c.

  (5c2) Using the union-find structure $\mathcal{U}$, merge the components containing s_ρ and s_i.

(5c3) Set $\rho \longleftarrow i$.

(5d) Otherwise, if s_i and s_ρ are in the same *semi-attached* connected component, then set $\rho \longleftarrow i$.

(6) At this point, all segments with endpoints on the boundary of c are either *attached*, or belong to a common *semi-attached* connected component.

(6a) If there is no bridge attached to w_0 passing through c, then introduce the new bridge (w_0, w_λ).

(6b) If there previously existed a bridge between w_0 and w_1, then delete the bridge, and replace it with bridges (w_1, w_λ) and (w_ρ, w_0).

(6c) Otherwise, there previously existed a bridge between w_0 and w_{k-1}. Delete this bridge, and replace it with bridges (w_0, w_λ) and (w_ρ, w_{k-1}).

(7) Relabel all *semi-attached* segments in S as being *attached*.

Note that StitchUp maintains each of the invariants listed in Subsection 4.1; in particular, a call to StitchUp cannot result in the connection of more than one bridge through c at any endpoint of any segment. The proof of this claim follows the discussion of Algorithm Encompass in the next section.

4.4 The Main Algorithm

Having described 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 the bounding box and 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*).

(3d) Mark the bounding box as *attached*.

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

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

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

(5) While there are cells yet *pending* do:

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

  (5b) STITCHUP(c).

The proof of correctness of Algorithm ENCOMPASS is by induction. It is easily seen that the first segments are correctly attached to G at Step (3) of ENCOMPASS. For each remaining segment s of S attached at Step (4) or Step (5), we assume inductively that both of the Lemmas 7 and 8 hold true before s is attached, and show that both also hold after s is attached.

The proofs of the lemmas rely on two main facts. First, ATTACHTO and STITCHUP both maintain the invariants set out in Subsection 4.1. In particular, any endpoint labeled *unvisited* or *pending* is not connected to a bridge. Both ATTACHTO and STITCHUP ensure that when a new bridge is introduced, that its endpoints will have acquired the label *examined*. Second, any endpoint with the label *examined* must have been given this label by ATTACHTO.

Lemma 7 *Let s_v be a segment labeled either* attached *or* semi-attached*, and let v be an endpoint of s_v which has become* examined *as a result of the application of* ATTACHTO *upon v. Let T_v be the extension tree of which v is a leaf, and let s be the segment of S at which T_v is rooted. If s was* unattached *at the time that* ATTACHTO *was invoked on v, then once this invocation has terminated, s must have correctly been made a member of the same connected component as s_v.*

Proof Let us assume that s is not in the same connected component as s_v after the invocation of ATTACHTO on v has terminated. If the call to ATTACHTO on v did not immediately result in s being bridged to v, then ATTACHTO attempted to link the neighbouring leaves of extension trees to an endpoint of s by means of bridges. If all these leaves were labeled *unvisited* or *pending*, then clearly the algorithm succeeded in bridging to and attaching the segment s, as no bridges had previously been attached to these leaves. Therefore at least one of the leaves (call it v') on the path to the endpoint of s must have previously been *examined*. As v' can only have been given this label by ATTACHTO, the induction hypothesis implies that v' and s are in the same connected component. However, if v' and v are not already in the same connected component, the bridge (v, v') introduced at Step (5b1) of ATTACHTO causes the components to be merged. From this contradiction, the result follows. □

Lemma 8 *Let c be a* pending *cell upon which the call* STITCHUP(c) *is made, at a time when all segments of S are labeled either* attached *or* unattached*. Then when the call to* STITCHUP *terminates,*

1. *All segments of S are again labeled either* attached *or* unattached.

2. *No endpoints of segments of S are* pending.

3. *c is correctly* connected.

Proof Consider the sequence of segment endpoints $W = \{w_0, w_1, w_2, \ldots, w_{k-1}, w_k\}$ encountered as one traverses the boundary of c in clockwise order, where $w_0 = w_k$ is the first endpoint on the boundary of c to have been marked by the algorithm as *examined*. Let s_i be the segment of which w_i is an endpoint, for all $0 \leq i \leq k$.

Imagine the boundary of c as viewed from the interior, in clockwise order starting from w_0. The extension edge emanating from each endpoint of W must itself lie on the boundary of c. If the extension edge of w_i follows w_i when scanning the boundary in clockwise order, then we will say that w_i is a *clockwise* (CW) endpoint of W. Otherwise, w_i will be called a *counterclockwise* (CCW) endpoint.

We first show that after the loop of Step (3) has terminated, all segments on the boundary of c are either *attached* or *semi-attached*. The invariants maintained during the execution of the loop are:

1. Immediately before the execution of Step (3b1), s_j is *attached* or *semi-attached* for all $0 \leq j \leq a$;

2. Immediately after the execution of Step (3b4), s_j is *attached* or *semi-attached* for all $0 \leq j \leq i+1$.

3. Immediately after the execution of Step (3b4), s_j belongs to the same connected component as s_{i+1}, for all $a < j \leq i$.

4. Except during the execution of Steps (3b1) to (3b4), no endpoints of edges are *pending*.

5. Immediately after the execution of Step (3b4), if s_i is labeled *semi-attached*, then there exist two endpoints w_{j_1} and w_{j_2} of W such that:

   - there are no bridges passing through the interior of c having w_{j_1} or w_{j_2} as an endpoint;
   - $j_1 < j_2 \leq i+1$;
   - if b is the smallest index such that s_b belongs to the same *semi-attached* component as s_i, then s_j also belongs to this component for all $b \leq j \leq j_2$.

Note that when StitchUp is invoked, no endpoints can be labeled *pending*. Also note that in the first iteration of the loop, a is set to 0, since s_0 had been *attached* prior to the call to StitchUp.

Consider now the situation at $i = p$ in which segment s_p is found to be *unattached* at Step (3a), and s_{p-1} has the label *attached* or *semi-attached*. Endpoint w_p cannot be CW; otherwise, the sequence of edges on the boundary of c between w_{p-1} and w_p would consist of extension edges, followed by a single segment edge adjacent to w_p. The extension tree of which w_{p-1} is a leaf would therefore be rooted at s_p. Since the loop invariant implies that w_{p-1} cannot be labeled *pending* (and therefore must be *examined*), Lemma 7 implies that s_p must be *attached* or *semi-attached* — a contradiction. Therefore w_p must be CCW in this situation.

Let $p < q \leq k$ be the smallest index such that either w_q is CW, or s_q is *attached* or *semi-attached*. We claim that in fact s_q must be *unattached*. Otherwise, we have two cases:

- *w_q is CW.*

  From the definition of q, we have that w_{q-1} is CCW. The only way a CCW endpoint can be followed by a CW endpoint in clockwise order about the boundary of c is if the endpoints belong to the same segment. However, the assumption that s_q is *attached* or *semi-attached* implies that $s_{q-1} = s_q$ is also *attached* or *semi-attached*. This contradicts the minimality of q.

- *w_q is CCW.*

  Since w_{q-1} is also CCW, the sequence of edges on the boundary of c between w_{q-1} and w_q would consist of extension edges, preceded by a single segment edge adjacent to w_{q-1}. The extension tree of which w_q is a leaf would therefore be rooted at s_{q-1}. Since the loop invariant implies that w_q cannot be labeled *pending* (and therefore must be *examined*), Lemma 7 implies that s_{q-1} must be *attached* or *semi-attached* — again, contradicting the minimality of q.

We are forced to conclude that w_q is CW, and also that $s_{q-1} = s_q$. This implies the following:

- Once an *unattached* segment s_p is discovered at Step (3a), the condition of Step (3b) will eventually be met at some $i \geq p$.
- Segment s_j is *unattached* for all $p \leq j \leq i + 1$.
- Endpoint w_j is CCW for all $p \leq j \leq i$.

When Steps (3b1) through (3b4) are executed, the effect is to render *semi-attached* (or possibly even *attached*) every segment s_j in the range $p \leq j \leq i + 1$. To prove that this is the case, consider the effect of initiating AttachTo at endpoint w_j, where w_j and w_{j-1} are both CCW. Arguments similar to those appearing above ensure that as a result of the call to AttachTo, segment s_{j-1} becomes a member of the same connected component as s_j. Endpoint w_{j-1} becomes either *pending* or *examined*, depending on the endpoint by which s_{j-1} becomes attached. However, Step (3b4) ensures that all *pending* endpoints become *examined* before the step terminates. Given that both endpoints of $s_i = s_{i+1}$ are labeled *pending* in Step (3b1), and that all w_j are CCW and s_j are initially *unattached* for all $p \leq j \leq i$, all w_j in the range must become *examined* by the time Step (3b4) terminates.

When Step (3b4) terminates, the second through fourth loop invariants mentioned above have been restored. Since $s_{i+1} = s_i$ becomes *semi-attached* (or possibly even *attached*) as a result, a is set to $i + 1$ in the next iteration of the loop. This guarantees that the first loop invariant holds for the next execution of Step (3b1), if any. Thus when the loop terminates, all segments in the boundary of c are indeed labeled either *attached* or *semi-attached*.

So far we have not justified the separate handling of the cases depending on the parity of $i - a$, in Steps (3b2) and (3b3). We claim that this separate handling allows the fifth loop invariant to be maintained. Let us assume then that s_i is *semi-attached* when Step (3b4) terminates.

If $i - a$ is even, then Step (3b2) prevents w_i from receiving a bridge through the interior of c before the termination of Step (3b4) — since all bridges introduced in Step (3b) merge segments into a common connected component, a second bridge through w_i would introduce a cycle. In order to identify a second endpoint that receives no bridge through c, consider the endpoints whose indices lie in the range $\{a+1, \ldots, i-1\}$. Any bridges introduced through c at any of the endpoints in the range must either link two consecutive endpoints in the range, or must link w_{a+1} with w_a. We have two cases:

- *w_{a+1} is not bridged to w_a.*

  Since no endpoint can receive more than one bridge, an even number of endpoints of the range must receive bridges. As the cardinality of the range is odd, there must be at least one endpoint that receives no bridge through c. Together with w_i, this leaves the *semi-attached* component of s_i with at least two unused endpoints with indices in the range $\{a+1, \ldots, i+1\}$.

  Let b be the smallest index such that s_b belongs to the same *semi-attached* component as $s_i = s_{i+1}$. If $b = a+1$, we are done, since s_j is in the component of s_i for all $a+1 \leq j \leq i+1$. Otherwise, $b \leq a$. Immediately before the current iteration of the loop, segment s_b must have been the segment of smallest index of a different *semi-attached* component. If so, the fifth loop invariant guarantees the existence of endpoints w_{j_1} and w_{j_2} to which no bridges were incident through c before the current iteration where $b \leq j_1 < j_2$. Since $s_{j'}$ was in the same component as s_b for all $b \leq j' \leq j_2$, then $j_2 \leq a$. And since no bridge was introduced at the current iteration between w_a and w_{a+1}, the endpoints w_{j_1} and w_{j_2} are not incident to bridges through c when the current iteration terminates. The fifth loop invariant is therefore satisfied in this case.

- *w_{a+1} is bridged to w_a.*

  If this occurs, the connected component of s_i merges with that of s_a. If s_a is *attached*, then s_i becomes *attached*, contradicting our assumption that s_i was *semi-attached* at the end of Step (3b4). If s_a is *semi-attached*, then the result of the merge is a single *semi-attached* component. By the fifth loop invariant, at least two endpoints w_{j_1} and w_{j_2} of W in the *semi-attached* component to which s_a belonged had no bridge attached to them passing through c. Let b be the smallest index of the segments in the component of s_a. As before, $b \leq j_1 < j_2 \leq a$.

  If $j_2 < a$, then the arguments of the previous case apply to show that the fifth loop invariant continues to hold. Otherwise, $j_2 = a$. Since s_j belongs to the former component of s_a for all $b \leq j \leq a$, and to the component of s_i for all $a+1 \leq j \leq i$, after the merge, s_j belongs to the component of s_i for all $b \leq j \leq i+1$. Endpoint w_{j_1} cannot have received a bridge as a result of the merge, and therefore w_{j_1} and w_i satisfy the conditions of the fifth loop invariant.

If $i - a$ is odd, Step (3b3) ensures that w_{i+1} will have no bridge attached to it through c. Considering that the range of indices $\{a+1, \ldots, i\}$ is of odd cardinality, the fifth loop invariant can be shown to hold using an argument almost identical to that of the case where $i - a$ is even.

At this point, we have shown that all five invariants are maintained by the loop of Step (3). In particular, when the loop terminates, the second, third, and fifth invariants still hold, and no endpoints of segments in S are *pending*. Each bridge passing through c must link consecutive endpoints of W, since it can only have been introduced via a call to AttachTo. Such bridges cannot interfere with any other bridges that may later be introduced between free endpoints of W.

The loop in Step (5) uses the index ρ to maintain the most recently-encountered unbridged endpoint of the current *semi-attached* component; whenever an unbridged endpoint of a new component is discovered, a bridge is introduced between w_ρ and the new endpoint, merging the components. The fifth invariant guarantees that when the first unbridged endpoint w_{j_1} of a new component is discovered, a second unbridged endpoint w_{j_2} of that component also exists, with $j_2 > j_1$. This ensures that the merged component has an unbridged endpoint that can be used to merge the next component to be discovered by the loop of Step (5). When the loop terminates, all *semi-attached* edges have been merged into one component, and λ and ρ are the minimum and maximum indices of the original unbridged endpoints taken over all *semi-attached* segments.

In Step (6), the *semi-attached* component is merged into the *attached* component via endpoints w_0, w_λ, and (perhaps) w_ρ. Since there exists no j in the interval $\{0, \ldots, \lambda\}$ such that w_j was an unbridged endpoint of a *semi-attached* segment before Step (5), the bridges (w_0, w_λ) would intersect no other bridges through c if introduced; similarly, the bridges (w_1, w_λ) and (w_ρ, w_{k-1}) would intersect no other bridges.

If w_0 is not already incident to a bridge through c, the introduction of bridge (w_0, w_λ) correctly merges the *semi-attached* and *attached* components. Otherwise, if the bridge (w_0, w_1) exists, replacing (w_0, w_1) by (w_1, w_λ) and (w_ρ, w_0) correctly splices the *semi-attached* component into the *attached* component between w_0 and w_1. The result is a single connected component that includes all endpoints of W.

To conclude the proof, we note that when StitchUp has terminated, no segments are *semi-attached* and no endpoints are *pending.* □

Lemma 9 Encompass *eventually terminates after taking at most $O(n \log n)$ time.*

Proof The construction of the underlying convex subdivision $\mathcal{Q}$ can be accomplished using planar line-sweep techniques, as outlined in Section 2, in $O(n \log n)$ time [11]. At the time of construction, pointers can be established linking the leaves of extension trees with the segments to which they are rooted, and counters can be set up to allow efficient modification of the labels of cells.

The total amount of work done in executing procedure AttachTo is proportional to the number of vertices and edges of $\mathcal{Q}$, which by Lemma 1 is in $O(n)$. The first time AttachTo is called on an endpoint, it marks it as *examined*; if called on the endpoint again, it simply exits without doing anything (this can be charged to the neighbour from which the call was made). The extension edges of $\mathcal{Q}$ can be traversed at most once in each direction when moving from neighbour to neighbour; segment edges can be traversed at most four times each (twice from each of the two cells it bounds).

The total work done in executing StitchUp on c can be divided into three categories: work involving the union-find structure $\mathcal{U}$; work involving calls to AttachTo; and the remainder of the work. The work involving calls to AttachTo has already been accounted for. Also, the relabeling of the segments in the final step of StitchUp can be charged to the segments themselves — since each segment can become *attached* only once, the total work performed in this step over all calls to StitchUp is in $O(n)$.

Let k_c be the number of subdivision edges on the boundary of cell c; this number is larger than the number of segment endpoints on the boundary. The total number of union-find operations is in $O(k_c)$, as well as the total time taken which has not already been accounted for by calls to AttachTo, or the relabeling discussed above. If standard union-find structures are used [4], the time taken to perform these operations is in $O(k_c \cdot \alpha(k_c))$, where $\alpha(k_c)$ is the very slowly-growing inverse of Ackermann's function. Since StitchUp can only be performed once per cell, the total time taken by calls to StitchUp is in $O(\sum_c (k_c \cdot \alpha(k_c))) \subseteq O(\alpha(n) \sum_c k_c)$. Since each non-box edge of the subdivision is contained in exactly two cells, and since the number of cells is in $O(n)$, $\sum_c k_c$ is proportional to the total number of edges of $\mathcal{Q}$, which is in $O(n)$. The total additional time taken by the calls to StitchUp is therefore in $O(n \cdot \alpha(n))$.

The overall work performed by Encompass is accounted for by the total work involved in calls to AttachTo, to the additional work performed by StitchUp, and to the construction of the convex subdivision. The time taken to construct the subdivision dominates, and thus the total time taken by Encompass is in $O(n \log n)$. □

Lemma 10 *When the execution of* Encompass *terminates, then G is a degree-3 encompassing tree for S.*

Proof Lemma 9 implies that Encompass does indeed terminate.

Assume that G does not encompass all segments of S. Then there exists at least one cell of $\mathcal{Q}$ that is *unvisited*. Since no cell is *pending* (by Step (5)), all cells that are not *unvisited* must be *connected*. The correctness of Step (3) ensures that all cells bordering the bounding box are not *unvisited*; therefore, they must all be *connected*.

Let $\mathcal{C}$ be the union of all cells that are *unvisited*. The boundary of $\mathcal{C}$ consists of a collection of disjoint simple cycles in $\mathcal{Q}$. There is at least one cycle in the boundary; call it C. Since C cannot contain box segments, Lemma 2 implies that there exists some segment endpoint v on C. Since v is on the boundary of an *unvisited* cell, v must be labeled *unvisited*. But v is also on the boundary of a *connected* cell, and must therefore be labeled *examined* — a contradiction. Every cell must therefore be connected. Lemma 6 then implies that G encompasses all segments of S.

The planarity and degree of G are a result of the invariants set forth in Section 4.1. □

Theorem 11 *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, 7, 8, 9, and 10. □

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, using a reduction similar to that for the convex hull problem [11]. This implies the optimality of Algorithm Encompass.

Theorem 12 *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 indices of the sorted order of the real numbers from smallest to largest; that is, x_{s_1} is the smallest of the numbers, and x_{s_n} is the largest. Let $m = x_{s_n}$ be the maximum element of S.

For each number x_i, we construct a corresponding vertical 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. The lower endpoint has coordinates $(x_i, x_i^2 - m^2 - 1)$, and the upper endpoint has coordinates $(x_i, -x_i^2 + m^2 + 1)$. Note that since m can be computed in linear time, the construction requires only linear time.

These endpoints are well defined — for all values of i, the lower endpoint is strictly below the $x - axis$, while the upper endpoint is strictly above. All of the lower endpoints lie on the upward-opening parabola $L : y = x^2 - m^2 - 1$ and all of the upper endpoints lie on the downward-opening parabola $U : y = -x^2 + m^2 + 1$. Since the endpoints are on the boundary of the a convex region (namely that bounded by U, L, and the y-axis), the fact that the segments are parallel means that the endpoints of l_{s_i} are visible from the endpoints of no other edges except $l_{s_{i-1}}$ and $l_{s_{i+1}}$.

Since the degree-3 encompassing tree consists of visibility edges between line segments together with the line segments themselves, 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. □

Although the segments of the proof were chosen to be parallel for the sake of convenience, constructions in which no two segments are parallel can also be used.

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 encompassing 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 of our algorithm. There are a number of open problems still to be considered.

1. Is it NP-hard to compute a simple polygon or a simple hamiltonian path through a set of disjoint line segments? Rappaport [12] has shown that the decision problem is NP-complete when the line segments are allowed to 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, internal diagonals or external diagonals [7]? Urabe and Watanabe [17] 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 [7]? If not, can anything be said about the longest path in the visibility graph?

Acknowledgments

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

References

[1] D. Avis and D. Rappaport, Computing monotone simple circuits in the plane, in *Computational Morphology*, G. T. Toussaint (ed.), Elsevier Science, North Holland, 1988.

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

[3] P. Bose and G. Toussaint, Growing a tree from its branches, *J. Algorithms* **19** (1995), 86–103.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, USA, 1990.

[5] H. ElGindy and G. Toussaint, Efficient algorithms for inserting and deleting edges from triangulations, in *Proc. Int. Conf. on Foundations of Data Organization*, Kyoto, Japan, 1985, pp. 163–169.

[6] B. Grünbaum, Hamiltonian polygons and polyhedra, *Geombinatorics* **3** (1994), 83–89.

[7] A. Mirzaian, Hamiltonian triangulations and circumscribing polygons of disjoint line segments, *Computational Geometry: Theory and Applications* **2** (1992), 15–30.

[8] C. Monma and S. Suri, Transitions in Geometric Minimum Spanning Trees, *Discrete Comput. Geom.* **8** (1992), 265–293.

[9] J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press, New York, 1987.

[10] J. O'Rourke and J. Rippel, Two segment classes with Hamiltonian visibility graphs, *Computational Geometry: Theory and Applications* **4** (1994), 209–218.

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

[12] D. Rappaport, Computing simple circuits from a set of line segments is NP-complete, in *Proc. 3rd ACM Symposium on Computational Geometry*, Waterloo, Ontario, 1987, pp. 322–330.

[13] D. Rappaport, H. Imai, and G. T. Toussaint, Computing simple circuits from a set of line segments, *Discrete and Computational Geometry* **5** (1990), 289–304.

[14] E. Rivera-Campo and J. Urrutia, *personal communication*, 1992.

[15] X. Shen and H. Edelsbrunner, A tight lower bound on the size of visibility graphs, *Information Processing Letters* **26** (1987), 61–64.

[16] T. Su and R. Chang, Computing the constrained relative neighborhood graphs and constrained Gabriel graphs in Euclidean plane. *Pattern Recognition* **24** (1991), 221–230.

[17] M. Urabe and M. Watanabe, On a counterexample to a conjecture of Mirzaian, *Computational Geometry: Theory and Applications* **2** (1992), 51–53.

[18] A. Yao, An $O(E \log\log V)$ algorithm for finding minimum spanning trees, *Information Processing Letters* **4** (1975), 21–23.