1990

P-Complete Geometric Problems

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Paul Callahan

Michael T. Goodrich

Report Number:

90-951

P-COMPLETE GEOMETRIC PROBLEMS

Mikhail J. Atallah
Paul Callahan
Michael T. Goodrich

CS-TR-951
February 7, 1990
(Revised 11/19/92)

P-Complete Geometric Problems*

Mikhail J. Atallah† Paul Callahan‡ Michael T. Goodrich§

Summary of Results

In this paper we show that it is impossible to solve a number of "natural" 2-dimensional geometric problems in polylog time with a polynomial number of processors (unless P = NC). Thus, we disprove a popular belief that there are no natural P-complete geometric problems in the plane. The problems we address include instances of polygon triangulation, planar partitioning, and geometric layering. Our results are based on non-trivial reductions from the monotone circuit value and planar circuit value problems.

1 Introduction

In sequential computation theory one of the primary measures of a solution's efficiency is that it run in time that is proportional to a polynomial in the size of the input. A problem solvable by such a sequential algorithm is said to be in the class P [2, 21]. An analogous notion of efficiency in parallel computation theory is that a solution run in polylog time using a polynomial number of processors. A problem solvable by such a parallel algorithm is said to be in the class NC [22, 24, 27, 31]. (The reader is referred to [24] for other notions of efficiency and related complexity classes.)

*A preliminary announcement of this research appeared in *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 317–326, 1990.

†This author's research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118. Author's address: Dept. of Computer Sciences, Purdue Univ., W. Lafayette, IN 47907.

‡This author's research was supported by an Abel Wolman Graduate Fellowship from the G.W.C. Whiting School of Engineering at Johns Hopkins University. Author's address: Dept. of Computer Science, The Johns Hopkins Univ., Baltimore, MD 21218.

§This author's research was supported in part by the National Science Foundation under Grant CCR-8810568 and CCR-9003299, and by the NSF and DARPA under Grant CCR-8908092. Author's address: Dept. of Computer Science, The Johns Hopkins Univ., Baltimore, MD 21218.

It is a long-standing open question as to whether all problems solvable in polynomial time sequentially are solvable in polylog time using a polynomial number of processors, i.e., whether P = NC or not. It is strongly believed, however, that P $\neq$ NC, much as it is strongly believed that P $\neq$ NP. As in the theory of NP-completeness, there is an analogous method for proving that establishing the membership of a particular problem in NC is as hard as showing that P = NC. This method is to show that each problem in P admits an NC-reduction to the problem at hand. Such problems are usually said to be *P-complete* [22, 27]. Most of the problems shown to be P-complete to date are essentially combinatorial problems, dealing with problems defined on graphs and algebras (see [22, 27], for example).

Geometric problems in the plane, as a rule, admit more structure than purely combinatorial problems, however. This structure usually allows one to apply parallel divide-and-conquer methods to obtain fast algorithms. In fact, all the well-known 2-D geometric structures, including convex hulls, Voronoi diagrams, and line arrangements, can be constructed in polylog time using a polynomial number of processors [1, 4, 5, 6, 8, 14, 15, 16].

Even the otherwise P-complete problem of linear programming can be parallelized when restricted to the plane (see [11, 12] for the P-completeness result). In addition, using the algebraic cell-decomposition framework of Kozen and Yap [23, 34], there are a host of less well-known geometric problems in the plane that can also be computed in parallel. Because of this, a general belief seems to have developed that "natural" geometric problems in the plane tend to be parallelizable.

The class NC consists of those problems that can be solved by a uniform family of polynomial size boolean circuits with polylog depth. This is usually taken to be equivalent to the set of problems that can be solved in polylog time with polynomially many processors in some "reasonable" model, such as a PRAM. However, in the case of geometric problems we need to be careful, because we often require infinite precision arithmetic operations, and these are not in NC.

One way to resolve this is to define a new complexity class, such as NC^* or NC^+ [1, 34] that allows infinite precision arithmetic as a basic operation. We consider NC to be a more natural class for the discussion of complexity issues, so we take a different approach.

In most common geometric problems, all arithmetic operations can be encapsulated within oracles that provide yes or no answers. For example, Voronoi diagrams can be constructed using only discrete operations, provided we have an oracle that, given four points, can answer whether the first lies within the circle passing through the remaining points. Additionally, the number of possible oracle queries tends to be bounded by a polynomial (in the previous case $O(n^4)$).

Hence, for present purposes, we assume that *an instance of a geometric problem includes the answer to all possible oracle queries.* Given this representation, it now becomes reasonable to pose the problem in terms of formal languages, and discuss whether it lies in NC or is P-complete.

In this paper we show that a number of simple geometric problems are in fact P-complete. Each of these problems involves a collection of line segments in the plane. The problems we address are as follows:

- *Plane-sweep triangulation.* One is given a simple n-vertex polygon P (which may contain holes) and asked to produce the triangulation that would be constructed by the following sequential algorithm: sweep the plane from top to bottom with a horizontal line L, such that each time L encounters a vertex v of P one draws from v all diagonals of P that do not cross previously drawn diagonals. This problem is a special case of the well-known polygon triangulation problem (see [13, 26, 32]). Contrast the P-completeness of this problem with the fact that so many problems solvable by plane-sweeping have recently been shown to be in NC [1, 4, 6, 14, 16, 17, 18] (with solutions that use a small number of processors).

- *Weighted planar partitioning.* One is given a collection of n non-intersecting segments in the plane, where each segment s has a distinct weight $w(s)$, and asked to construct the partitioning of the plane produced by extending the segments in order of their weights. The extension of a segment "stops" at the first segment (or segment extension) that is "hit" by the extension. This problem has applications to *art gallery* problems [28], as was shown by Czyzowicz *et al.* [10]. We show it to be P-complete even if there are only 3 possible orientations for the line segments. It is straightforward to solve this problem sequentially in $O(n \log^2 n)$ time (using the dynamic point-location data structure of [33]), and in $O(n \log n)$ time by a more sophisticated method [10].

- *Visibility layers.* One is given a collection of n non-intersecting segments in the plane, and asked to label each segment by its "depth" in terms of the following layering process (which starts with $i = 1$): compute the upper envelope of the segments (i.e., those visible from $(0, +\infty)$), label each segment with a piece in this upper envelope as being at depth i, remove each such segment, increment i, and repeat until no segments are left. This is an example of a class of problems in computational geometry known as *layering* problems or *onion peeling* problems [7, 26, 29], and we show it to be P-complete even if all the segments are horizontal. It can be solved in $O(n \log n)$ time sequentially [30]. Hershberger [20] independently developed a P-completeness proof for this problem.

Our methods are based on non-trivial reductions from the monotone circuit value problem (MCVP) and planar circuit value problem (PCVP), which are known to be P-complete [19, 24, 27]. The main difficulty in these reductions is showing how to use geometry to simulate a circuit just by using the relative positions of objects in the plane. As is often the case with completeness results, we expect the techniques (and perhaps even the problems themselves) to be useful in showing other geometric problems to be P-complete.

In the sections that follow we outline our reductions for each of the above problems in turn. For pedagogical reasons, we present our proofs as NC-reductions, but we could have just as easily presented them as logspace-reductions (which is an alternate framework for P-completeness proofs, e.g., see [22, 31]).

2 A Framework For Geometric Reductions

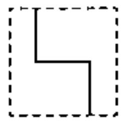
We show the first two problems to be P-complete using reductions from the planar circuit value problem (PCVP). Because our constructions are geometric, particular care must be taken in the routing of values. We will handle routing within a general framework in which we insert our constructions as components.

We will assume an instance of PCVP is given as a circuit composed of $\vee$ gates and inverters that is embedded in a grid. Inputs are placed at the top, and the circuit is organized as an alternating sequence of routing layers and logic layers. In each row, certain columns are assigned to the value of gates in the circuit. One can easily show that PCVP remains P-complete under these assumptions.

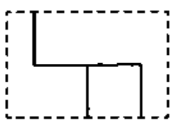
2.1 Routing

We describe the routing layer in terms of 4 components: vertical wires, right shifts, left shifts, and fan-out gates. These components are shown below:

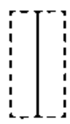
left shift



right shift



fan-out gate



vertical wire

Each figure is simply a "wiring" diagram, and when two figures are placed with their wires touching, a value is transmitted through each wire in the natural way. Values can only be transmitted down in the vertical direction, though they may be transmitted either left or right horizontally.

We construct a routing layer using a geometric placement of these components in which their bounding rectangles (shown dotted) do not overlap. This restriction is significant, because the objects in our geometric problems must not intersect. To formalize the idea of routing, we define a *column value assignment*, and a class of functions, called *planar routing functions*, which transform one column value assignment to another.

Because we use a grid embedding, we need to assign boolean values to columns in the grid. To make routing possible, we also need to leave some columns empty. We give empty columns the value $*$. A *column value assignment* is an n-tuple $\langle v_1, \ldots, v_n \rangle \in \{0, 1, *\}^n$, where 0 and 1 represent boolean values, and $*$ stands for "unassigned."

A *routing function* r is a function from column value assignments to column value assignments (of equal size) and is represented by an n-tuple $C = \langle c_1, \ldots, c_n \rangle \in \{0, \ldots, n\}^n$, with the following interpretation.

Let $\langle v'_1, \ldots, v'_n \rangle = r(\langle v_1, \ldots, v_n \rangle)$. Then, for all i,

$$v'_i = \begin{cases} * & \text{if } c_i = 0 \\ v_{c_i} & \text{otherwise} \end{cases}$$

Intuitively, such an r represents the interconnections from outputs on one level to inputs on the next level. It is more general than a permutation, because values can be duplicated, and not all columns need to be connected.

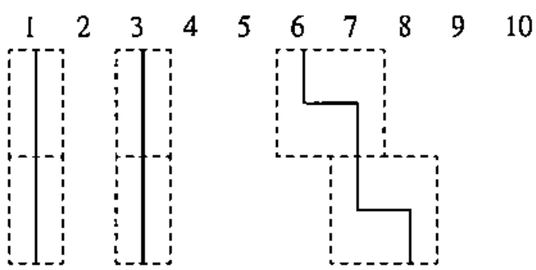
A *planar routing function* is a routing function represented by $C = \langle c_1, \ldots, c_n \rangle$, such that for all $1 \le i < j \le n$, if $c_i, c_j \ne 0$ then $c_i \le c_j$.

Intuitively, a routing function is planar if it can be embedded in the plane as a network of non-crossing wires. The formal restriction above insures that values transmitted across such a network appear in the same order on output as on input, and it is easy to see that this is equivalent to the non-crossing restriction.

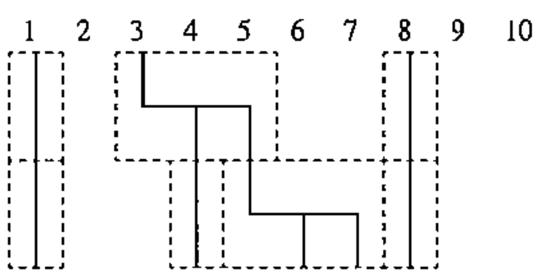
Lemma 2.1: *Suppose r is a planar routing function represented by $C = \langle c_1, \ldots, c_n \rangle$, where $|\{i: c_i = 0\}| \ge n/2$. Then r can be realized using an n-column routing layer consisting of vertical wires, left shifts, right shifts, and fan-out gates, such that the bounding rectangles of these components do not overlap. Moreover, the problem of constructing this routing layer is in NC.*

Proof. The restriction $|\{i: c_i = 0\}| \ge n/2$ is merely to insure that we have enough empty space to fit our routing components. We perform routing in a naive manner, because we need only guarantee that the size of the instance in the geometric framework is a polynomial function of the size of the original circuit instance.

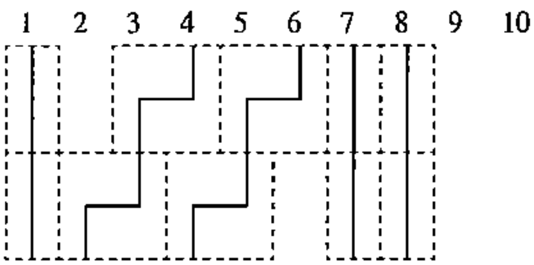
We construct the routing layer in three phases. For example, suppose we wish to route the function r, where $C = \langle 1, 3, 0, 3, 0, 0, 3, 6, 0, 0 \rangle$. First, we spread values to allow enough room for fan-out gates:



Second, we perform the fan-out:



Finally, we place values in the appropriate columns:



The first and third phases consist of the same operation: routing wires without crossing or branching. The second phase is an even simpler process of copying values into adjacent columns to the right. We describe its operation first.

To perform fan-out, we assume that each column corresponding to a value has at least $2(m-1)$ empty columns to its right where m is the number of copies of that value produced by the routing function (this is why we need $|\{i: c_i = 0\}| \geq n/2$). Using these empty columns, we make $m-1$ new copies of the value using $m-1$ levels of gates. At each level, a fan-out gate is placed beneath the rightmost copy, and a vertical wire is placed beneath all other copies. Note that each fan-out gate takes up two columns to the right of its predecessor. Thus, $2(m-1)$ empty columns are sufficient.

It is easy to see that in this scheme the placement of each fan-out gate and vertical wire can be precomputed using a simple formula. Therefore, this phase can be computed in NC.

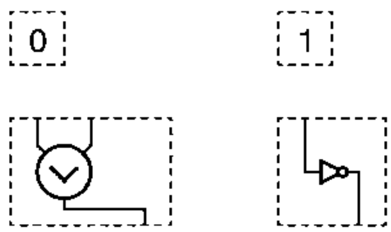
We perform the routing of the first and third phases in a naive manner in order to keep the computation simple. We split the set of values into three categories: those to be routed to the left, those to be routed to the right, and those to be routed straight down.

We route each value to its destination column in the most obvious way: we use a sequence of shifts in the desired direction. While one value is routed, all other values are passed along using vertical wires (note that our illustration of the third phase deviates from this naive approach in order to save space). At any stage in the routing, the leftmost value to be routed to the left, and the rightmost value to be routed to the right may each be routed in this way. (any value which blocks such a routing must cross the value being routed, and this violates the planarity restriction).

To perform the routing, we order the values to be routed to the left from left to right, and those to be routed to the right from right to left (these orderings may be interleaved arbitrarily). We then route each value successively in the obvious manner. From this ordering, the column containing each value prior to the routing of a particular value can clearly be computed in NC. Given this information, the placement of each shift and vertical wire can be computed according to a simple formula. Therefore, the first and third phases can be computed in NC. □

2.2 Inputs and gates

We also introduce the following components for inputs and gates:



$\vee$ *gate* *inverter*

We use input components to realize an initial column value assignment and gate components to realize a *logic layer function.*

Definition. A *logic layer function* l is a function from column value assignments to column value assignments of equal size. It is represented by a *column gate assignment*, $\mathcal{G} = \langle g_1, \ldots, g_n \rangle \in \{\vee, \neg, *\}^n$ where each $\neg$ is preceded by at least one $*$, and each $\vee$ is preceded by at least two $*$'s. Intuitively, this is to make room for the input columns. It is interpretted as follows.

Let $\langle v'_1, \ldots, v'_n \rangle = l(\langle v_1, \ldots, v_n \rangle)$. Then, for all i,

$$v'_i = \begin{cases} v_{i-2} \vee v_{i-1} & \text{if } g_i = \vee \\ \neg v_{i-1} & \text{if } g_i = \neg \\ * & \text{otherwise} \end{cases}$$

It follows immediately from the definition that we may realize a logic layer function in terms of logic components. This realization is a simple matter of local replacement, and hence it is in NC.

2.3 Specifying an instance of PCVP

We will assume that an instance of PCVP is given as an input assignment $\langle v_1, \ldots, v_n \rangle$, an alternating sequence $\langle r_1, l_1, \ldots, r_m, l_m \rangle$ of planar routing functions and logic layer functions, and a distinguished column i, called the output. We may pose this instance as a decision problem by asking the following question.

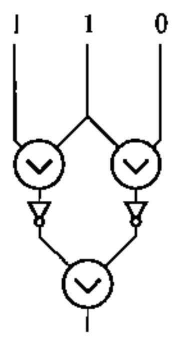
Let $\langle v'_1, \ldots, v'_n \rangle$ denote the result of

$$l_m(r_m(\ldots l_1(r_1(\langle v_1, \ldots, v_n \rangle))))$$

Does $v'_i = 1$?

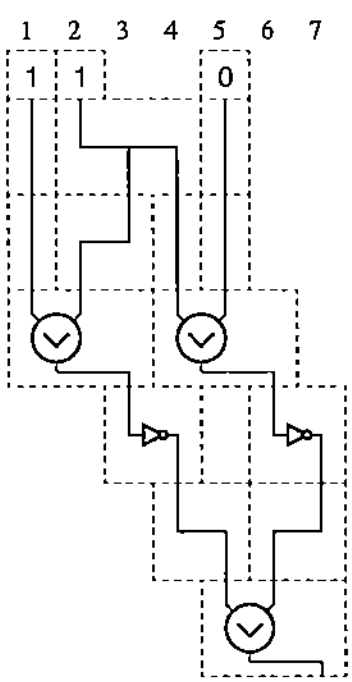
The PCVP has been shown to be P-complete for the basis of boolean functions $\{\vee, \neg\}$ [19], and it is easy to see that an instance of PCVP specified in any "reasonable" format is NC-reducible to an instance in the above format.

As an example of the reduction to the geometric framework, consider the following instance of PCVP:



Input:	$\langle 1,1,*,*,0,*,*,* \rangle$
r_1:	$\langle 1,2,0,2,5,0,0,0 \rangle$
l_1:	$\langle *,*,\vee,*,*,\vee,*,* \rangle$
r_2:	$\langle 0,0,3,0,0,6,0,0 \rangle$
l_2:	$\langle *,*,*,\neg,*,*,\neg,* \rangle$
r_3:	$\langle 0,0,0,0,4,7,0,0 \rangle$
l_3:	$\langle *,*,*,*,*,*,\vee,* \rangle$
Output:	column 7

It is represented in the geometric framework as:



Using the framework developed above, we will now show our first two geometric problems to be P-complete. Our third problem will require some modifications to this framework, which we will develop when needed.

3 The Plane-Sweep Triangulation Problem

We consider the problem of triangulating a simple n-vertex polygon P, which may contain holes. We recall that the *plane-sweep triangulation* is the one constructed by the following sequential algorithm: sweep the plane from top to bottom with a horizontal line L, such that each time L encounters a vertex v of P one draws from v all diagonals of P that do not cross previously drawn diagonals.

It is easy to see that this problem can be solved using only discrete operations if we assume an oracle that accepts two line segments as input and determines whether they intersect. We need only perform queries on line segments between points in the given polygon, so there are $O(n^4)$ possible queries. Hence, we can assume the input to this problem consists of the suitably encoded polygon, along with the answers to all oracle queries.

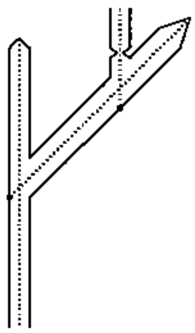
The problem of finding some arbitrary triangulation is known to be in NC (see [14, 17]). In this section, we prove the following theorem.

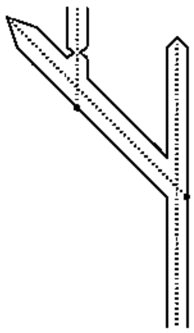
Theorem 3.1: *The plane-sweep triangulation problem is P-complete.*

Proof. Given the observations of the preceding section, and Lemma 2.1, it is sufficient to construct "gadgets" for each of the components needed to embed a planar circuit. We present each gadget as the object one would see within a rectangular window placed over the corresponding part of the polygon. These gadgets fit together in precisely the same manner as the components in our geometric framework. We will first prove the correctness of our gadgets, and we will then show how to express the resulting polygon in a standard encoding scheme.

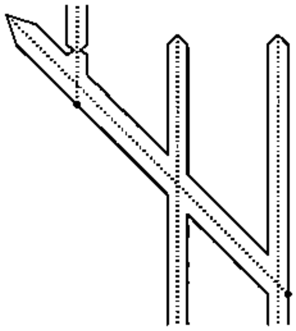
We encode each value in the circuit by the presence or absence of a corresponding edge in the triangulation. The presence or absence of an edge will represent 1 or 0, respectively. All edges that correspond to an input or output value of a gate will be vertical. The correspondence between edges and column values follows from the geometry.

The gadgets for the left shift and right shift, respectively, are:

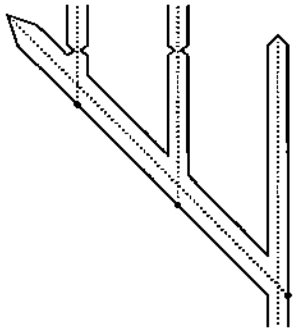


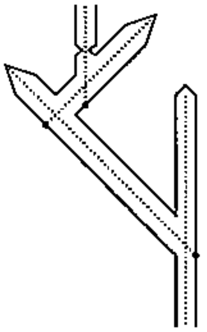


The gadget for the fan-out gate and vertical wire, respectively, are:



The gadgets for the $\vee$ gate and inverter, respectively, are:





The shapes of our constructions may appear mysterious at first, but the ideas governing them are quite simple. We construct logic gates by making use of the interaction between crossing pairs of line segments. These interacting edges are shown as dotted lines. Any polygon we form will introduce spurious edges in the triangulation graph. We must be careful to construct our gates to prevent these edges from altering the meaning of edges that encode logic functions. We show how this is done by considering some general features of gates.

In each construction we add several new vertices to the polygon (shown with dots). These may be considered 180° corners. We will call these vertices *targets*, because the output of each gate will be computed by attempting to draw a line segment to each target from some vertex above.

In every gate, the input edge is a vertical line segment passing through an opening made sufficiently small to insure that only one vertex above the opening is visible from the target directly below. This opening can be made as small as necessary by the placement of two reflex corners. The maximum width of this opening depends on the length of the longest vertical line segment leading to a target, and it can easily be computed given the placement of the gadgets.

Functionally, most gadgets work in two phases corresponding to "events" in the plane sweep. The first phase corresponds to sweeping past the highest vertex and determining if its target is visible. It will be visible unless some input edge is blocking it. In this case, one would add the edge connecting the highest vertex to its target. The second phase corresponds to sweeping past the next highest vertex (or vertices in the case of the fanout gate), and "attempting" to add a vertical edge from it to its target in some gate below. This is possible iff the edge of the first phase was not added. As one sweeps past the rest of the construction, one may add other edges, but these will not affect the output edges, since they have already been added.

An exception to the above description is the inverter, which works in three phases. In each phase, the edge interactions are the same as those above. To invert a value we must use an odd number of such interactions. After one interaction, we always change the orientation of the edge encoding the value. Thus, we need at least three interactions to invert a value while maintaining the orientation of the edge encoding it. The treatment of spurious edges is somewhat more complicated in this gadget, but its operation is easily verified.

We form inputs by closing off the openings of the top row of gates with gadgets that insure the presence of an edge in the case of a 1, or the absence in the case of a 0, as follows:

We close off the output of the circuit using the same gadget that encodes 1, placed upsidedown. The vertical edge connecting to the lowest vertex in this gadget will be in the triangulation iff the output of the corresponding circuit is 1.

To show that this polygon can be constructed in NC, we assume that it is encoded as a set of sequences of points in the plane. Each sequence is a boundary of the polygon (either the outer boundary or a hole contained within), and points are given in the order obtained by traversing the boundary either clockwise or counterclockwise from an arbitrary starting point back to itself.

We assume that gadgets other than the vertical wire are given as a graphs in which each vertex corresponds to a corner, and each edge corresponds to a side connecting two corners. For the vertical wire, we introduce two dummy vertices corresponding to each side, and use the resulting graph (with no edges) to represent the gadget. We construct a graph for the whole polygon by introducing edges between any two vertices adjacent to incomplete sides that "touch" in the geometrical placement.

The construction of the above graph requires only local replacement. Encoding the polygon as a set of sequences can be done by using list ranking [3] and parallel prefix [25] and then compressing out the dummy vertices. All of these steps can be done in NC. It is easy to see that the results of all possible oracle queries can be computed in NC as well. Thus, the plane-sweep triangulation problem is P-complete, by an NC reduction from PCVP. □

4 The Weighted Planar Partitioning Problem

In this section we consider the weighted planar partitioning problem, showing it to be P-complete. We recall that in this problem, one is given a collection of n non-intersecting segments in the plane, where each segment s has a distinct weight $w(s)$. One is asked to construct the partitioning of the plane produced by extending the segments in order of their weights. The extension of a segment "stops" at the first segment (or segment extension) that is "hit" by the extension.

We begin our discussion by noting that a line segment can only block another of unequal slope. Thus, a natural restriction is to limit the number of possible slopes of line segments to some constant k. We will call this restricted problem the *k-oriented* weighted planar partitioning problem.

As in the preceding section, our oracle will answer segment intersection queries. However, the segments may not always be from the input set, since they may be extensions of the original line segments. The reader may easily verify that each line segment has $O(n)$ possible extensions in each direction, so the number of possible oracle queries is again polynomial.

Theorem 4.1: *The 3-oriented weighted planar partitioning problem is P-complete.*

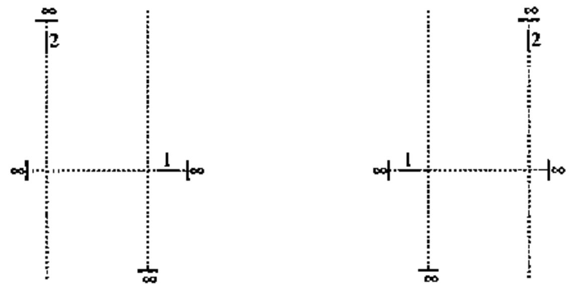
Proof. As in the proof of Theorem 3.1, we construct components for routing and logic. Throughout our proof, only 3 distinct slopes are used. In fact, with the exception of a single line segment used in the construction of the inverter, only horizontal and vertical line segments are used.

These gadgets work on principles similar to those of the preceding problem. Specifically, line segments are considered in a particular order, and if they are added, they block the placement of other line segments to be added later. In the preceding problem, however, a blocked line segment is not placed at all, whereas in this problem a blocked line segment is extended up to the intersection with the one that blocks it. It is easy to verify that this does not affect the behavior of the gadgets in terms of the logical functions they are designed to perform, however.

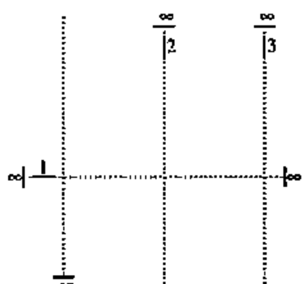
In this problem, unlike the preceding, the ordering of the line segments does not depend on the geometry, but is imposed separately by the weights. Thus, we need to specify an ordering along with the line segments to make our construction work. Our figures are not as self-explanatory as those of the preceding section.

Each line segment in a figure is labelled with either a number or ∞ (dotted lines represent extensions). The numbers indicate the order in which the line segments fall with respect to the weights of other line segments within a particular gadget. We order gadgets in a manner corresponding to the sequential evaluation of the instance of PCVP. One such ordering is row by row from the top, with an arbitrary ordering among elements in a row. We may then order the complete list of numbered line segments by combining these orderings (via a parallel prefix computation), with the latter being the most significant. We place line segments labelled ∞ after all other line segments in the final ordering. Their order with respect to each other is arbitrary. Intuitively, line segments labelled with ∞ are placed to force numbered line segments to be extended in only one direction, or to prevent a segment from being extended beyond the gate to which it belongs. We do not care how these blocking line segments are extended.

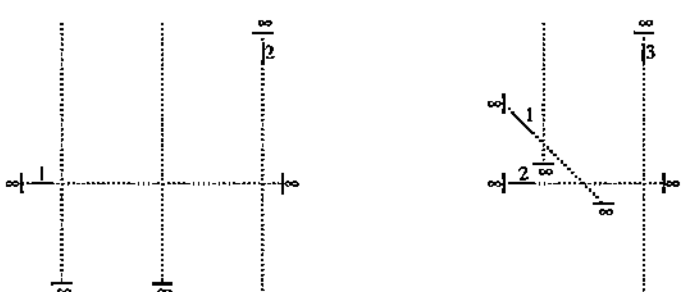
The gadgets for the left shift and right shift, respectively, are:



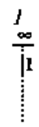
The gadget for the fan-out gate is:



There is no explicit gadget corresponding to a vertical wire. A value is transmitted vertically as the extension of a line segment within some gate. The gadgets for the $\vee$ gate and inverter, respectively, are:



We assign inputs by using the following gadget to represent 1:



There is no explicit gadget for 0, since it is represented by the absence of an extension.

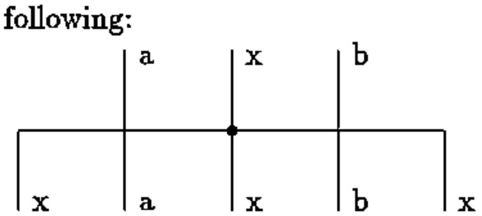
The correctness of these gadgets is easily verified, including the fact that we may insert them into our general framework. The computation of the weight for each line segment is clearly in NC, as is the computation of oracle queries. The remainder of the reduction consists of local replacement. Therefore, the weighted planar partitioning problem is P-complete. □

5 The Visibility Layers Problem

In this section we will show that the visibility layers problem is P-complete. We recall that in this problem, one is given a collection of n non-intersecting segments in the plane, and asked to label each segment by its "depth" in terms of the following layering process (which starts with $i = 1$): compute the upper envelope of the segments (i.e., those visible from $(0, +\infty)$), label each segment with a piece in this upper envelope as being at depth i, remove each such segment, increment i, and repeat until no segments are left.

In this case, the problem is fully specified by the ordering of line segment endpoints and the partial ordering of line segments according to "aboveness." Hence, we can dispense entirely with arithmetic operations by assuming that these relations are given along with the input.

As with the previous two constructions, the primary consideration is in the routing. However, in this case we will not be restricted to planar routing functions. We introduce a more powerful routing construction, which we call a crossing fan-out gate. This allows a very general fan-out in which we may copy the value of any column i to any subset of columns, including i. We allow this gate to cross any number of columns without affecting them. This permits the realization of arbitrary routing functions. An example of such a gate is the following:



Our reduction is from the monotone circuit value problem (MCVP). As we did for PCVP, we assume that an instance of MCVP is given as an input assignment $\langle v_1, \ldots, v_n\rangle$, an alternating sequence $\langle r_1, l_1, \ldots, r_m, l_m\rangle$ of routing functions and logic layer functions, and a distinguished column i, called the output. In the case of MCVP, however, we do not insist that the routing functions r_i be planar. The logic layer functions l_i are similar to those used for PCVP, except that their column gate assignments are drawn from the set $\{\wedge, \vee, I, *\}$, where I is the identity function. We define these logic layer functions as follows.

Let $\langle v'_1, \ldots, v'_n\rangle = l(\langle v_1, \ldots, v_n\rangle)$. Then, for all i,

$$v'_i = \begin{cases} v_{i-2} \vee v_{i-1} & \text{if } g_i = \vee \\ v_{i-2} \wedge v_{i-1} & \text{if } g_i = \wedge \\ v_i & \text{if } g_i = I \\ * & \text{otherwise} \end{cases}$$

For convenience, we assume that the value of a column is never reassigned. We formalize this notion as a restriction on routing and logic functions (an intuitive explanation is given afterwards).

Suppose $r_i = \langle c_1, \ldots, c_n\rangle$ is a routing function in which $c_j \notin \{0, j\}$. Then the following must be true:

- $v_j = *$
- For all $r_k = \langle c'_1, \ldots, c'_n\rangle$ such that $k < i$, $c'_j = 0$.
- For all $l_k = \langle g_1, \ldots, g_n\rangle$ such that $k < i$, $g_j = *$.

Analogously, suppose $l_i = \langle g_1, \ldots, g_n\rangle$ is a logic function in which $g_j \notin \{*, I\}$. Then the following must be true:

- $v_j = *$
- For all $r_k = \langle c_1, \ldots, c_n\rangle$ such that $k \leq i$, $c_j = 0$.
- For all $l_k = \langle g'_1, \ldots, g'_n\rangle$ such that $k < i$, $g'_j = *$.

The above restrictions insure that once the value of a column has been assigned by a gate or an input, it may only be propagated straight down or discarded (note that we introduced the identity function I to allow values to be propagated straight down through a logic layer). In no event can a new value be assigned to a column that has had a value assigned to it at a higher level. The need for this restriction will become apparent when we discuss our construction. It is easy, however, to see that this does not limit the kinds of circuits we can describe, and that an instance of MCVP in any "reasonable" form is NC-reducible to this form.

Theorem 5.1: *The visibility layers problem is P-complete.*

Proof. We divide our construction into levels, so that at each level i the values 0 and 1 are represented by the layer numbers $4i$ and $4i+1$, respectively. Note that the 0-valued layer always comes before the 1-valued layer (this is crucial to the correctness of our constructions). Intuitively, each level will consist of 4 visibility layers.

Our mechanism for assigning columns is capable of changing a 1 to a 0, but not vice versa. Thus, columns whose value is initially unassigned in our input (based on our general framework) will be represented by $4i+1$, the same as 1.

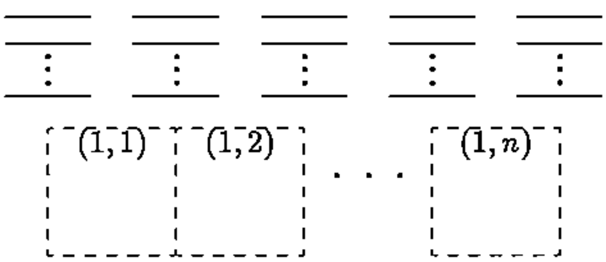
Because our values must be synchronized precisely, we need to be especially careful about placement. In order to determine the placement of our gadgets, we will consider the plane to be a discrete set of unit square cells, each centered at coordinates (i,j). We interpret these coordinates so that i is the row number, increasing from top to bottom, and j is the column number, increasing from left to right. Note that this *not* the standard interpretation of cartesian coordinates. It is more like the interpretation of matrix subscripts. The latter scheme is more natural, since layer numbers roughly increase with respect to row numbers.

In the two preceding sections, we could point to specific geometric objects which each represented a value by its presence or absence. In this case, however, our encoding scheme is more subtle. In particular, there is no explicit object that transmits a value vertically. Intuitively, information seems to travel down the layers in a wave spread across the whole set of line segments. It is somewhat surprising that we can, in fact, produce the effect of a distinct value propagating down each column.

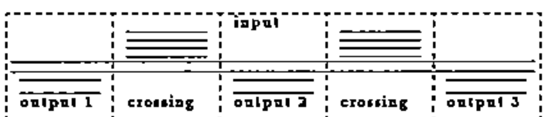
For this purpose, we introduce a set of $n+1$ *blockers*. Each blocker is a stack of line segments placed above the grid (the set of cells containing routing and logic) to insure that segments passing through each grid cell are only visible through a narrow column in the center. We will call this column the *aperture*. There must be enough line segments in each blocker to insure that this condition remains until all non-blocker line segments have been given layer numbers. An upper bound on this number can be computed after placing all other line segments in the construction, by finding the maximum number of such line segments stabbed by a vertical line.

For the purpose of evaluating the layer numbers of non-blocker line segments, we need not concern ourselves with those parts of line segments that pass beneath the blockers. These will never be visible as we evaluate the gadgets corresponding to our circuit.

We place the blockers as follows:



In order to form the routing component, we need to construct a crossing fan-out gate. The gadget corresponding to the crossing fan-out gate shown earlier is:



The gate will extend across an entire row of the grid. Its main component is the *conduit*, which consists of two long line segments, placed so that they pass across the apertures of all cells in the row. The idea is that as soon as a line segment in the conduit becomes visible through the aperture of the input column, it is removed. The effect of this removal is then transmitted to the output columns.

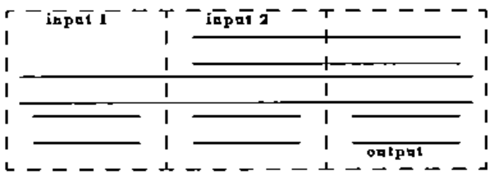
We construct the input and output columns of a crossing fan-out gate by placing either *taps* or *crossovers* in the appropriate cells. A tap consists of two line segments *below* the conduit, entirely within the cell, passing across the aperture. A crossover is similar, but consists of four line segments *above* the conduit. In general, if the ith column is the input or one of the outputs, we place a tap in the ith cell. Otherwise we place a crossover.

Intuitively, the purpose of a crossover is to prevent the value above an aperture from affecting the value of the conduit. By our encoding scheme, we are guaranteed that at least one of the line segments in a crossover will persist until *after* both line segments of the conduit have been removed. Thus, a value entering at a column with a crossover will not affect the value of any of the outputs. A tap simply resynchronizes the layer numbers to preserve our encoding scheme by adding 2 to the layer number. Note that the conduit also adds 2.

From this construction it may appear that we are not distinguishing the input from outputs. However, this distinction comes from the fact that we do not reuse columns. The input will always be from a column whose value has already been assigned, and the output will always be to a column that has not yet been assigned. By our encoding scheme, the only column that can possibly be 0 is the input column.

The gadget for the $\wedge$ gate is:

and the gadget for the $\vee$ gate is:



The correctness of the $\wedge$ gate follows from the fact that the output is the minimum of the two inputs plus 4. The $\vee$ gate is slightly more complicated.

We verify its operation by considering the layer numbers of the segments in the output column. The top two segments are assigned layer numbers that depend on the value entering at input 2, and the middle two depend on input 1 in a similar manner. The bottom two segments will not be removed until all those above them have been removed, so they depend on the maximum layer number of those above. Each pair of segments adds two to the value it depends on, so the final output is the maximum of the two inputs plus 4.

We assume that the output column is unassigned above each gate. This is to insure that the output depends only on inputs 1 and 2, and not on some value above the output column. In keeping with the definition of the logic layer function, we declare the input columns to be unassigned below the gate.

The construction for the identity function is the same as the construction for the crossover in the crossing fan-out gate. The only difference is that there is no conduit that it must cross. Its only effect is to add 4 to the layer number.

We construct our input component above the blockers, by placing a single line segment across the aperture of a column to represent 1 or unassigned. We represent 0 by the absence of such a line. It is easy to see that this assigns appropriate initial values to the layers immediately below.

We consider the output of the circuit to be the bottom line segment in the output column designated in the original instance of MCVP.

All of the steps in the preceding reduction can clearly be done in NC. Therefore, the visibility layers problem is P-complete, by an NC reduction from MCVP. □

6 Discussion and Open Problems

We have shown three simple geometric problems in the plane to be P-complete. Thus, these problems cannot be solved with a polynomial number of processors in polylogarithmic time (unless P=NC). Two of the problems were decomposition problems and the third was a layering problem.

A important layering problem whose membership in NC remains an open problem is the well-known convex layers problem[1] [7]. This problem is analogous to the visibility layers problem, but the input is a set of points, and we remove the points of the convex hull at each step of our iterative procedure.

Some other open problems in this domain include the following:

- Suppose we restrict plane-sweep triangulation to polygons without holes. Does the problem remain P-complete? We suspect that this restriction places the problem in NC.
- Consider 2-oriented weighted planar partitioning. Is this problem P-complete? In fact, this problem can be reduced to a case of MCVP with a very restricted (though not planar) topology, but it is not clear that this places it in NC.
- Suppose we restrict visibility layers to horizontal line segments of unit width. This makes it impossible to form crossovers. Does the problem remain P-complete? We suspect that it does not, but so far no NC algorithm has been found.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," *Algorithmica*, **3**(3), 1988, 293–328.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] R.J. Anderson and G.L. Miller, "Deterministic Parallel List Ranking," *AWOC 88*, 81–90.

[1]The membership in NC of the convex layers problem was posed as an open problem by Atallah at the 11th NYU Computational Geometry Day and by Chazelle at the 1st DIMACS Workshop on Geometric Complexity.

[4] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *SIAM Journal on Computing*, Vol. 18, No. 3, June 1989, 499–532.

[5] M.J. Atallah and M.T. Goodrich, "Efficient Parallel Solutions to Some Geometric Problems," *J. of Parallel and Dist. Comp.*, **3**(4), Dec. 1986, 492–507.

[6] S. Chandran, "Merging in Parallel Computational Geometry," Ph.D. thesis, Dept. of Computer Science, Univ. of Maryland, CS-TR-2333, 1989.

[7] B. Chazelle, "On the Convex Layers of a Planar Set," *IEEE Trans. on Info. Theory*, Vol. IT-31, 1985, 509–517.

[8] A. Chow, "Parallel Algorithms for Geometric Problems," Ph.D. thesis, Comp. Sci. Dept., Univ. of Illinois, 1980.

[9] R. Cole, "Parallel Merge Sort," *SIAM J. Computing*, Vol. 17, No. 4, August 1988, 770–785.

[10] J. Czyzowicz, I. Rival, and J. Urrutia, "Galleries, Light Matchings and Visibility Graphs," *Lecture Notes in CS: 382, Proc. Workshop on Algorithms and Data Structures (WADS)*, Springer-Verlag, 1989, 316–324.

[11] D. Dobkin, R.J. Lipton, and S. Reiss, "Linear Programming is Log-space Hard for P," *Info. Proc. Letters*, Vol. 9, 1979, 96–97.

[12] D. Dobkin and S. Reiss, "The Complexity of Linear Programming," *Theoretical Computer Science*, Vol. 11, 1980, 1–18.

[13] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, NY, 1987.

[14] M.T. Goodrich, "Efficient Parallel Techniques for Computational Geometry," Ph.D. thesis, Dept. of Comp. Sci., Purdue Univ., August 1987.

[15] M.T. Goodrich, "Finding the Convex Hull of a Sorted Point Set in Parallel," *Information Processing Letters*, Vol. 26, Dec. 1987, 173–179.

[16] M.T. Goodrich, "Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors," *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures*, 127–137.

[17] M.T. Goodrich, "Triangulating a Polygon in Parallel," *Journal of Algorithms*, Vol. 10, 1989, 327–351.

[18] M.T. Goodrich, M. Ghouse, J. Bright, "Generalized Sweep Methods for Parallel Computational Geometry," manuscript, 1990.

[19] L.M. Goldschlager, "The Monotone and Planar Circuit Value Problems are Log Space Complete for P," *SIGACT News*, Vol. 9, No. 2, 1977, 25–29.

[20] J. Hershberger, "Upper Envelope Onion Peeling," *Proc. 2nd Scandinavian Workshop on Algorithm Theory*, 368–379.

[21] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Adison-Wesley, 1979.

[22] R.M. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines," Report UCB/CSD 88/408, EECS Dept., Univ. of California, Berkeley, 1988.

[23] D. Kozen and C.K. Yap, "Algebraic Cell Decomposition in NC," *Proc. 26th IEEE Symp. on Foundations of Computer Science*, 1985, 515–521.

[24] C.P. Kruskal, L. Rudolph, and M. Snir, "A Complexity Theory of Efficient Parallel Algorithms," *Lecture Notes in CS: 317, Proc. 15th ICALP*, Springer-Verlag, 1988, 333–346.

[25] Ladner, R.E., and Fischer, M.J., "Parallel Prefix Computation," *J. ACM*, 1980, 831–838.

[26] D.T. Lee and F.P. Preparata, "Computational Geometry—A Survey," *IEEE Trans. on Computers*, Vol. C-33, No. 12, December 1984, pp. 872–1101.

[27] S. Miyano, S. Shiraishi, and T. Shoudai, "A List of P-complete Problems," Technical Report RIFIS-TR-CS-17, Research Institute of Fundamental Information Science, Kyushu University (Fukuaka 812, JAPAN), 1989.

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

[29] J. O'Rourke, "Computational Geometry," *Ann. Rev. Comput. Sci.*, Vol. 3, 1988, 389–411.

[30] M. Overmars, personal communication, October 1989.

[31] I. Parberry, *Parallel Complexity Theory*, Pitman Publishing, 1987.

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

[33] F.P. Preparata and R. Tamassia, "Fully Dynamic Techniques for Point Location and Transitive Closure in Planar Structures," *Proc. 29th ACM Symp. on Theory of Computing*, 1988, 558–567.

[34] C.K. Yap, "What can be Parallelized in Computational Geometry?," manuscript, 1987.