1993

Optimal Parallel Hypercube Algorithms for Polygon Problems

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

Danny Z. Chen

Report Number:
93-027

OPTIMAL PARALLEL HYPERCUBE ALGORITHMS FOR POLYGON PROBLEMS

Mikhail J. Atallah
Danny Z. Chen

CSD-TR-93-027
April 1993
Revised September 1993

Optimal Parallel Hypercube Algorithms for Polygon Problems*

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

Danny Z. Chen‡
University of Notre Dame
chen@cse.nd.edu

Abstract

We present parallel techniques on hypercubes for solving optimally a class of polygon problems. We thus obtain optimal $O(\log n)$-time, n-processor hypercube algorithms for the problems of computing the portions of an n-vertex simple polygonal chain C that are visible from a given source point, computing the convex hull of C, testing an n-vertex simple polygon P for monotonicity, and other related problems as well. Previously it was not known how to achieve these complexity bounds on hypercubes, one of the main difficulties being that there is no known optimal sorting hypercube algorithm that achieves these bounds. In fact these are the first optimal geometric hypercube algorithms that do not assume that the input is given already sorted by x or y coordinates. The hypercube model we use is the standard one, with $O(1)$ local memory per processor, and with one-port communication.

Key Words: Algorithms, computational geometry, convex hulls, hypercubes, kernel, monotonicity, simple polygons, visibility

*Part of this research was done while the authors were visiting the Leonardo Fibonacci Institute in Trento, Italy.

†Department of Computer Sciences, Purdue University, West Lafayette, IN 47907. This author's research was supported by the National Science Foundation under Grant CCR-9202807.

‡Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556.

1 Introduction

Let $C = (p_1, p_2, \ldots, p_n)$ be an n-vertex polygonal chain that is simple (i.e., C does not self-intersect), where the p_i's are the vertices of C. C can be *closed*, i.e., it can form a simple polygon (in this case, we denote by P the polygon formed by C). We present parallel techniques on hypercubes for solving optimally a class of problems on C. We obtain optimal $O(\log n)$-time, n-processor hypercube algorithms for the following polygon problems:

- Computing the portions of C visible from a given source point [9, 12, 14, 15]. Contrast this with the problem of computing the visibility of arbitrary nonintersecting line segments, for which no such optimal hypercube bounds are known [19]. Recall that given C and a specified point q in the plane, a point p on C is said to be *visible* from q if and only if the interior of the line segment $\overline{pq}$ (whose endpoints are p and q) does not intersect C.

- Computing the convex hull of C [13, 16, 22]. Contrast this with the problem of computing the convex hull of arbitrary points in the plane, for which no such optimal hypercube bounds are known [20]. Recall that given a set S of geometric objects in the plane, the problem of computing the convex hull of S is that of finding the convex polygon with the minimum area which contains S.

- Testing polygon P for monotonicity [23]. Recall that a chain C' is monotone with respect to a line L if and only if for every line L' that is orthogonal to L, $C' \cap L'$ is either empty or a single point, and that P is monotone if there exists a line L such that the boundary of P can be partitioned into two chains each of which is monotone with respect to L. The problem of testing the monotonicity of P is that of finding a description for all the lines with respect to each of which P is monotone (if such a line exists), or reporting that no such lines exist (and hence P is not a monotone polygon).

- Computing the kernel of P [17]. Recall that the kernel K of P is the subset of the points in P such that any point in K can "see" the whole polygon (K can be empty).

- Computing the maximal elements in the set of vertices of C [22]. Recall that for two points p and q in the plane, p is said to *dominate* q if and only if both the x and y coordinates of p are equal to or larger than those of q, respectively. For a set S of points in the plane, a point $p \in S$ is called a *maximal element* of S if and only if there is no point $q \in S$ such that $q \neq p$ and q dominates p.

Previously, it was not known how to achieve these complexity bounds on hypercubes, one of the main difficulties being that there is no known optimal sorting hypercube algorithm that achieves these bounds. In fact, these are the first optimal geometric hypercube algorithms that do not assume that the input is given already sorted by x or y coordinates. For example, the problems of computing the visibility of n nonintersecting line segments from a point and computing the convex hull of n points in the plane have been considered on hypercubes; $O(Sort(n))$-time, n-processor hypercube algorithms for these problems were given in [19, 20], where $Sort(n)$ is the time complexity of the best algorithm for sorting n values on an n-processor hypercube. Currently, $Sort(n) = O(\log n(\log\log n)^2)$ [8]. Note that the best known upper bound for the time complexity of the general routing algorithms in the hypercube model we consider is also $Sort(n)$.

The hypercube model we use is the standard one: It has n processors, each with $O(1)$ local memory, and with one-port communication. For a detailed discussion of the hypercube models, the reader is referred to [18]. That the bounds which we achieve are optimal follows from two facts: (i) n processors are needed just to store the input (since a processor can store only $O(1)$ input data items), and (ii) the diameter of the hypercube is logarithmic.

We assume that the input C to our problems is given sorted by the *chain order* $\prec_c$, i.e., the order in which the vertices appear along C. The order $\prec_c$ is described implicitly by the way in which the elements (e.g., the vertices and/or edges) of C are initially stored in the processors of the n-processor hypercube: An element in processor PE_i is $\prec_c$ an element in PE_{i+1}. The output for the problems considered consists of a subset of C to be produced according to some sorted order $\prec$ that is different from $\prec_c$; the total order $\prec$ of the output is usually implicit in the statement of a problem being solved. For example, if we are computing the portions of C visible from a given source point q, then $\prec$ is the sorted order of the vertices of the visibility chain of C according to their polar angles with respect to q; hence the desired output is the set of (say) m vertices of the visibility chain of C, stored in processors $PE_1, PE_2, \ldots, PE_m$, in the sorted order of $\prec$. Figure 1 illustrates, for the above-mentioned visibility problem, how $\prec$ is quite different from $\prec_c$. The vertices of the output visibility chain of C in this example are in the following order which is not the same as $\prec_c$: $p_{13}, p_{12}, p_9, p_8, p_7, p_5, p_3, p_1, p_2, p_4, p_6, p_{10}, p_{11}, p_{14}, p_{15}, p_{18}, p_{19}, p_{22}, p_{24}, p_{26}, p_{28}, p_{30}, p_{29}, p_{27}, p_{25}, p_{23}, p_{21}, p_{20}, p_{17}, p_{16}$.

As in the PRAM algorithms [1, 5, 6, 7] for solving the polygon problems we consider, our algorithms use the following divide-and-conquer scheme. The n-vertex polygonal chain C is partitioned into g contiguous subchains (with respect to $\prec_c$) $C_1, C_2, \ldots, C_g$ of size $n^{1-1/d}$ each, where $g = n^{1/d}$ for some constant integer $d > 1$ and the size of C_i is the number of edges on C_i, denoted by $|C_i|$.

Figure 1: Illustrating the definitions for visibility from a source point at $(0, +\infty)$.

Then the g subproblems on all the C_i's are solved recursively in parallel, resulting in subsolutions for the C_i's, denoted by $Sol(C_i)$'s, $1 \leq i \leq g$. After all the g recursive calls return, the $Sol(C_i)$'s are "combined" into the overall solution $Sol(C)$, in $O(\log n)$ time. The main difficulty for implementing such a scheme optimally on an n-processor hypercube (as opposed to, say, the PRAMs) lies in the apparent need for (currently unknown) linear-processor, logarithmic-time sorting and routing algorithms.

Sorting comes into the picture at two different places in the above scheme:

- Determining for each $Sol(C_i)$ which portions of $Sol(C_i)$ appear in $Sol(C)$.
- Combining the portions of the $Sol(C_i)$'s that appear in $Sol(C)$ into the overall solution $Sol(C)$, sorted according to $\prec$.

It is the fact that we know $\prec_c$ that enables us to obtain $Sol(C)$ sorted by the $\prec$ order. We make crucial use of the geometry, and hence we do not need to rely on new insights on the general sorting problem on hypercubes. Our observations are useful in solving a number of geometric problems involving polygonal chains (we mention five such problems later on).

The rest of this paper is organized as follows. The next section gives the basic machinery that our algorithms need. Section 3 describes the overall algorithmic structure on which all our solutions are based. Section 4 illustrates how the visibility problem is solved based on such an algorithmic structure. The algorithm for testing the monotonicity of a simple polygon is given in Section 5,

and the algorithms for the other three polygon problems are given in Section 6. Section 7 mentions some open problems.

2 Main Building Blocks

This section develops the machinery that will be needed in our solutions. The purpose of much of this machinery is to avoid using general sorting routines and to avoid using data structures that require general routing (which were involved in the PRAM solutions). The geometric observations presented in this section hold for most of the problems we consider (except for the kernel problem). None of these observations was used in the existing PRAM algorithms [1, 5, 6, 7] (in fact, so far as we know, this is the first use of these observations in parallel geometric algorithms). Since the proofs of these observations are problem-dependent, we postpone them to the later sections about each of the polygon problems we consider.

We begin with the following definitions.

Definition 1 *Let S be a sequence of vertices and segments, and let $<$ be a total order on the elements of S (the sequence S is not given sorted according to $<$). A* subsequence of S *consists of a sequence of (not necessarily contiguous) portions of S that are obtained from S by walking from left to right along S, and choosing these portions of S during the walk. Such a subsequence S' of S is* monotonic with respect to $<$ *if the order in which the elements of S' occur along S is the same as $<$. A monotonic S' can either be in increasing or decreasing order. A subsequence S' that is not monotonic with respect to $<$ is said to be* tangled with respect to $<$.

In Figure 1, for example, the given sequence of the vertices of C is tangled with respect to the total order $\prec$, but subsequence $S' = p_1, p_4, p_6, p_{11}, p_{14}, p_{18}, p_{22}, p_{24}, p_{27}$ is monotonic with respect to $\prec$.

Definition 2 *For any sequence S, let reverse(S) denote the sequence obtained by going through S in the reverse direction. If S contains geometric entities (e.g., segments) then these too are reversed, so that (e.g.) a segment $\overline{uv}$ in S becomes $\overline{vu}$ in reverse(S).*

Recall that in our algorithms, C is partitioned into contiguous equal-sized subchains C_1, C_2, $\ldots$, C_g (where $C_1 \prec_c C_2 \prec_c \cdots \prec_c C_g$). The $Sol(C_i)$'s in the sorted order of $\prec$ are returned by the recursive calls. Let Q_i denote the list consisting of the portions of $Sol(C_i)$ that appear in $Sol(C)$ (i.e., $Q_i = Sol(C_i) \cap Sol(C)$). In Figure 1, the places marked by the x's describe the partition of C

into the C_i's. The p_i's shown in Figure 1 do not include all the vertices of C and are there just to illustrate some of the definitions.

Assume for the time being that, in the conquer stage of our algorithms, we have already obtained all the Q_i's from the $Sol(C_i)$'s (more on how this is done later), so that our main problem becomes how to merge the g $(= n^{1/d})$ lists Q_i, $1 \leq i \leq g$, in logarithmic time; the merge of the Q_i's is the $Sol(C)$ (in the sorted order of $\prec$) that we seek. If one used pairwise merging to obtain $Sol(C)$ from the Q_i's, then this computation would take $O((\log n)^2)$ time; this is because there are too many such Q_i's ($n^{1/d}$ of them), and merging a pair of Q_i's takes $O(\log n)$ time.

We illustrate our method for obtaining $Sol(C)$ from the Q_i's by using the example in Figure 1 for visibility. In Figure 1, Q_1 contains p_5, p_3, p_1, p_2, p_4 in that order, Q_2 contains p_9, p_8, p_7, p_6 in that order, and Q_5 contains $p_{18}, p_{19}, p_{21}, p_{20}$ in that order.

Let Q be the sequence obtained by concatenating the Q_i's. That is, $Q = Q_1 Q_2 \cdots Q_g$. Assume that the elements of Q are stored in processors PE_1, PE_2, ..., $PE_{|Q|}$, one element per processor, by the order of their appearance in Q. In the example of Figure 1, the p_i's appear in Q in the order

(*) $p_5, p_3, p_1, p_2, p_4, p_9, p_8, p_7, p_6, p_{13}, p_{12}, p_{10}, p_{11}, p_{14}, p_{15}, p_{17}, p_{16}, p_{18}, p_{19}, p_{21}, p_{20}, p_{22}, p_{24}, p_{25}, p_{23}, p_{26}, p_{27}, p_{28}, p_{30}, p_{29}$.

Note that having Q is quite different from the $Sol(C)$ we seek, which for Figure 1 would contain the p_i's in the order:

(**) $p_{13}, p_{12}, p_9, p_8, p_7, p_5, p_3, p_1, p_2, p_4, p_6, p_{10}, p_{11}, p_{14}, p_{15}, p_{18}, p_{19}, p_{22}, p_{24}, p_{26}, p_{28}, p_{30}, p_{29}, p_{27}, p_{25}, p_{23}, p_{21}, p_{20}, p_{17}, p_{16}$.

Although the p_i's in Figure 1 do not show all the vertices of C, they do illustrate the ordering relationship between $Sol(C)$ (which we seek) and Q (which we assumed we already have). The key observation to make here is that when walking along $Sol(C)$ from left to right (i.e, in the increasing order of $\prec$), the vertices of C

- first follow the decreasing order of $\prec_c$, until the smallest vertex (e.g., p_1 in Figure 1) of $Sol(C)$ in the $\prec_c$ ordering is encountered,
- then follow the increasing order of $\prec_c$, until the largest vertex (e.g., p_{30} in Figure 1) of $Sol(C)$ in the $\prec_c$ ordering is encountered, and
- finally follow again the decreasing order of $\prec_c$.

This property on the $\prec_c$ ordering of the vertices along $Sol(C)$ will be proved in Section 4 together with other observations for the visibility problem.

Our strategy for obtaining $Sol(C)$ from Q consists of two stages:

1. We partition Q into three subsequences L, M, and R (mnemonics for "left", "middle", and "right", respectively), as follows:

   (a) Let s be the smallest, t be the largest, vertices of Q in the $\prec_c$ ordering. Without loss of generality, assume that $s \prec t$. (In Figure 1, $s = p_1$ and $t = p_{30}$.)

   (b) Obtain from Q all the vertices v for which $v \prec s$. This is the subsequence L. This computation is easily accomplished by a parallel prefix and a monotone routing [18]. In Figure 1, the resulting L would contain the following p_i's, in that order:

   $$p_5, p_3, p_9, p_8, p_7, p_{13}, p_{12}.$$

   Note that L is *tangled* with respect to both $\prec$ and $\prec_c$.

   (c) Obtain from Q all the vertices v for which $t \prec v$. This is the subsequence R. In Figure 1, the resulting R would contain the following p_i's, in that order:

   $$p_{17}, p_{16}, p_{21}, p_{20}, p_{25}, p_{23}, p_{27}, p_{29}.$$

   Note that R too is *tangled* with respect to both $\prec$ and $\prec_c$.

   (d) Obtain from Q all the vertices v for which $s \preceq v \preceq t$. This is the subsequence M. In Figure 1, the resulting M would contain the following p_i's, in that order:

   $$p_1, p_2, p_4, p_6, p_{10}, p_{11}, p_{14}, p_{15}, p_{18}, p_{19}, p_{24}, p_{26}, p_{28}, p_{30}.$$

   Note that M is *sorted* with respect to both $\prec$ and $\prec_c$.

2. We sort L, in the following way. Note that L consists of the concatenation of $\beta \leq g$ pieces, that is, $L = \sigma_1 \sigma_2 \cdots \sigma_\beta$, where:

   (a) σ_i is the portion that is $\prec s$ of some Q_j,

   (b) every piece σ_i is increasing with respect to the $\prec$ order, and

   (c) for any $1 \leq i < j \leq \beta$, if $v \in \sigma_i$ and $w \in \sigma_j$ then $w \prec v$.

First, we reverse the order of each σ_i so that it is decreasing with respect to $\prec$. This segmented routing of the σ_i's can be easily performed in logarithmic time (see Section 3.5.1 of [18]), and it gives $L' = reverse(\sigma_1)reverse(\sigma_2)\cdots reverse(\sigma_\beta)$. L' is sorted in decreasing order with respect to $\prec$, and hence computing $reverse(L')$ by reversing L' gives us the version of L sorted by increasing order of $\prec$. This is the portion of $Sol(C)$ that is $\prec s$.

3. We sort R, in a way similar to L.

4. $Sol(C)$ is simply the concatenation of the sorted version of L, M, and the sorted version of R.

All of the above assumes that we have already obtained the Q_i's from the $Sol(C_i)$'s. Doing so in logarithmic time is nontrivial but follows without difficulty from (i) the above-mentioned observations, and (ii) generalizations of the geometric observations we made earlier for visibility [1] to the other polygon problems we consider here. Of particular importance are the observations that enabled us in [1] to get around the problem of computing, in logarithmic time and using a sublinear number of PRAM processors, the two intersections between two visibility chains. Similar observations hold for the other polygon problems considered.

In general, the implementation of the computation of the Q_i's from the $Sol(C_i)$'s requires parallel searches on each $Sol(C_i)$, involving g^c "search queries" in $Sol(C_i)$ for some constant integer $c > 1$ (c depends on the specific problem). The outcome of these searches is a determination of which portions of $Sol(C_i)$ are in $Sol(C)$ (and hence are in Q_i).

3 Overall Structure of the Algorithms

All our hypercube algorithms are based on the same structure, which is given below.

Input. The polygonal chain C (in the sorted order of $\prec_c$), where $|C| = n$.

Output. The solution $Sol(C)$ (in the sorted order of $\prec$), which depends on the specific problem.

1. If $|C| = 1$, then $Sol(C)$ is trivially obtained from C by using a sequential algorithm on a single processor. Otherwise, the following steps are taken.

2. C is partitioned into g contiguous (with respect to $\prec_c$) subchains $C_1, C_2, \ldots, C_g$, where $g = n^{1/d}$ for some constant integer $d > 1$ (d is problem-dependent). The size of each C_i is $n^{1-1/d}$.

3. Then the subproblems on all the C_i's are solved recursively in parallel, resulting in subsolutions $Sol(C_i)$'s, $1 \leq i \leq g$.

4. After all the g recursive calls return, the $Sol(C_i)$'s are "combined" into the overall solution $Sol(C)$, in $O(\log n)$ time on an n-processor hypercube.

The above scheme would result in algorithms that run in totally $O(\log n)$ time on an n-processor hypercube, if the "conquer" stage of the scheme could be performed in the claimed complexity bounds. This is because the recurrence relations for the time complexity $T(n)$ of such a scheme would be as follows:

$$T(m) = T(m^{1-1/d}) + a \log m, \text{ if } m > 1$$

$$T(1) = b$$

for some positive constants a and b. It is trivial to show from these recurrence relations that $T(n) = O(\log n)$.

In the following sections, we show how to perform in the desired complexity bounds the conquer stage of the above scheme for the visibility problem, and how to reduce the other polygon problems that we consider to the visibility problem and to other problems for which optimal hypercube algorithms are already known.

4 Visibility

In this section, we first prove the property of the $\prec_c$ ordering of the vertices along $Sol(C)$, and then describe our hypercube algorithm for computing the visibility of chain C from a given source point q. Without loss of generality, we assume that $q = (0, +\infty)$ (see [1] for the method of dealing with the case where q is a finite point).

Given C, let S be the polygon consisting of all the points in the plane which are visible from q when C is considered to be the only "opaque" object. Note that S is star-shaped because the vertices of S are sorted by their x-coordinates. Then $VIS(C)$, the *visibility chain* of C with respect to q, equals the boundary of S minus the (at most two) edges on the boundary of S that are incident to the point at infinity. Note that in this problem, the $\prec$ order is the $<$ order of the x-coordinates of the points on $VIS(C)$, and that $Sol(C) = VIS(C)$.

Let a segment $\overline{uv}$ of $VIS(C)$ belong to an edge $\overline{p_i p_{i+1}}$ of C. Then for every point $p \in \overline{uv} - \{p_{i+1}\}$, we define the *rank* of p as i. This ranking information is useful in the PRAM algorithm [1] and also in this algorithm. Let the x (resp., y) coordinate of a point p be $x(p)$ (resp., $y(p)$).

Definition 3 *Let $\overline{uv}$ and $\overline{vw}$ be two consecutive edges of $VIS(C)$. We call v a* switching vertex *of $VIS(C)$ if and only if one of the following conditions holds:*

(1) *$v \prec_c u$ and $v \prec_c w$ in C but $x(u) < x(v) \leq x(w)$,*

(2) *$w \prec_c v$ and $u \prec_c v$ in C but $x(w) \leq x(v) < x(u)$,*

(3) *$v \prec_c u$ and $v \prec_c w$ in C but $x(w) \leq x(v) < x(u)$, and*

(4) *$w \prec_c v$ and $u \prec_c v$ in C but $x(u) < x(v) \leq x(w)$.*

In Figure 1, both p_1 and p_{30} are switching vertices of $VIS(C)$.

The switching vertices of $VIS(C)$ are at the places of $VIS(C)$ where the chain order $\prec_c$ of C is inconsistent with the chain order $\prec$ of $VIS(C)$, since between two consecutive switching vertices, the consecutive segments of $VIS(C)$ form a chain that is in either increasing or decreasing chain order $\prec_c$ of C.

The following lemma is crucial for the machinery in Section 2.

Lemma 1 *There are at most* two *switching vertices on $VIS(C)$. Furthermore, they are the vertices of $VIS(C)$ whose ranks are, respectively, the smallest and largest among all the points on $C \cap VIS(C)$.*

Proof. Let p_s (resp., p_l) be the vertex on $VIS(C)$ that has the smallest (resp., largest) rank among all the points on $C \cap VIS(C)$. Without loss of generality, we assume that $x(p_s) < x(p_l)$ (the other case is proved similarly).

We first show that when walking along $VIS(C)$ from p_s to p_l, the ranks of the points on $C \cap VIS(C)$ so encountered are in increasing order of $\prec_c$ (note that the points so encountered are in increasing order of their x-coordinates, i.e., $\prec$). Suppose this is not the case. Then there are two points p_v and p_w on $C \cap VIS(C)$ such that $x(p_s) < x(p_v) < x(p_w) < x(p_l)$, and $p_w \prec_c p_v$. Then we must have the situation of Figure 2 (a), in which either p_w or p_s is not visible from q because the subchain of C from p_v to p_l blocks the visibility between p_w (or p_s) and q (a contradiction), or C is not simple (again a contradiction).

Now let p_u be the vertex on $VIS(C)$ such that $x(p_u)$ is the smallest among all the points on $C \cap VIS(C)$ whose x-coordinates are smaller than $x(p_s)$. If such a p_u does not exist, then we are done. So assume that p_u does exist. Then we show that when walking along $VIS(C)$ from p_u to p_s, the ranks of the points on $C \cap VIS(C)$ so encountered are in decreasing order of $\prec_c$. Suppose this is not the case. Then let p_z be a point on $C \cap VIS(C)$ such that $x(p_u) < x(p_z) < x(p_s)$ and

Figure 2: Illustrating the proof of Lemma 1.

$p_u \prec_c p_z$. Then we must have the situation of Figure 2 (b), in which either p_u or p_s is not visible from q because the subchain of C from p_z to p_l blocks the visibility between p_u (or p_s) and q (a contradiction), or C is not simple (again a contradiction).

The case for the subchain of $VIS(C)$ which is to the right of $x(p_l)$ can be proved similarly. □

An implication of Lemma 1 is that, given every Q_i, that is, the list containing the portions of $VIS(C_i)$ that appear in $VIS(C)$, $VIS(C)$ can indeed be obtained in the sorted order of $\prec$, in $O(\log n)$ time. Therefore, what left to be shown is that given $VIS(C_i)$, how to compute Q_i in $O(\log n)$ time, for every $i = 1, 2, \ldots, g$.

We already showed in [1] that, for two subchains C' and C'' of C that are disjoint except possibly at a common endpoint, the visibility chains $VIS(C')$ and $VIS(C'')$ can intersect each other at most *two* times (Lemma 4.2 of [1]), and that for each i, $Q_i = VIS(C) \cap VIS(C_i)$ consists of at most *three* connected components (Lemma 4.1 of [1]). (Note that the intersections between the visibility chains must be located in order to find the Q_i's.) Also, when computing the two intersections between two visibility chains, it is possible to avoid the linear work lower bound which holds for computing the intersections between two simple polygonal chains [4], by exploiting the geometric properties of the visibility chains in our case (Lemmas 4.5 and 4.6 of [1]). The PRAM algorithm in [1] performs $O(1)$ parallel searches on the $VIS(C_i)$'s to compute the intersections between the visibility chains. After all the intersections are found, the Q_i's can be decided by another $O(1)$ parallel searches. Hence, one of the difficulties we face in the conquer stage of our hypercube algorithm is how to perform a parallel search in $O(\log n)$ time.

In our hypercube algorithm, when the g recursive calls return with the $VIS(C_i)$'s, every $VIS(C_i)$ is stored in its sub-hypercube, in the sorted order of $\prec$. This provides the basis for performing paral-

lel searches on the $VIS(C_i)$'s. In order to perform a parallel search on the hypercube in logarithmic time, we must appropriately choose the constant integer d which controls the divide-and-conquer scheme for our algorithm. Instead of using the *quarter-root* divide-and-conquer strategy in [1], we use here a *fifth-root* divide-and-conquer strategy, that is, we choose $d = 5$. This is because such a divide-and-conquer scheme enables us to implement the parallel searches of [1] in logarithmic time on the hypercube, by using the algorithm of [21] for sorting a small set of numbers and by using parallel merge [3]. The parameter c which is the constant integer for controlling the number of "search queries" of each parallel search on every $VIS(C_i)$ is chosen to be 3. That is, in each parallel search, there are $O(n^{c/d}) = O(n^{3/5})$ search queries being performed on each $VIS(C_i)$ simultaneously, with $O(n^{4/5})$ processors available for every C_i. Summing over all $i = 1, 2, \ldots, g$, the total number of search queries performed simultaneously on the $VIS(C_i)$'s is $O(n^{4/5})$. Therefore, all the routing operations for moving $O(n^{4/5})$ data items around can be performed on the n-processor hypercube in $O(\log n)$ time by using the sorting algorithm in [21]. In this hypercube scheme, the number of parallel searches in the conquer stage is still $O(1)$, and each parallel search still takes $O(\log n)$ time. Therefore, the Q_i's can all be obtained in $O(\log n)$ time.

The details for computing the Q_i's from the $VIS(C_i)$'s on the n-processor hypercube are given below. We need to review several definitions from [1]. For every subchain C_i of C, let $B_i = C_1 \cup C_2 \cup \cdots \cup C_{i-1}$ and $A_i = C_{i+1} \cup C_{i+2} \cup \cdots \cup C_g$ (i.e., B_i is the subchain of C *before* C_i and A_i the subchain *after* C_i).

Note that if a point p on $VIS(C_i)$ is hidden by either A_i or B_i (or equivalently, by either $VIS(A_i)$ or $VIS(B_i)$), then p cannot belong to $Q_i = VIS(C_i) \cap VIS(C)$. Therefore, we only need to show how to compute the portions of $VIS(C_i)$ that are hidden by $VIS(B_i)$ (the case for $VIS(A_i)$ is symmetrical). By Lemma 4.2 of [1], $VIS(C_i)$ and $VIS(B_i)$ can intersect each other at most two times, and we need to compute these intersections (if any) in order to find the portions of $VIS(C_i)$ hidden by $VIS(B_i)$. In general, for a pair of $VIS(C_i)$ and $VIS(B_i)$, the following computation is performed.

(1) Find the number I_i of the intersections between $VIS(C_i)$ and $VIS(B_i)$.

(2) If $I_i = 2$, then reduce the problem of computing the *two* intersections between $VIS(C_i)$ and $VIS(B_i)$ to two separate problems of computing the *one* intersection between two subchains of $VIS(C_i)$ and $VIS(B_i)$ (by Lemmas 4.5 and 4.6 of [1]).

(3) If $I_i \geq 1$, then solve the (at most two) problems of computing the one-intersection between two subchains of $VIS(C_i)$ and $VIS(B_i)$.

(4) Using the intersections (between $VIS(C_i)$ and $VIS(B_i)$) computed above, find the portions of $VIS(C_i)$ hidden by $VIS(B_i)$.

As shown in [1], each of the above four steps can be performed by doing $O(1)$ parallel searches. Therefore, we only show how a parallel search is actually implemented on the hypercube. In particular, we show how to perform a parallel search on the hypercube for the one-intersection computation in Step (3) (the parallel searches for other steps above are implemented similarly).

It should be pointed out first that the $VIS(B_i)$'s are not explicitly available, since we have obtained from the g recursive calls only the $VIS(C_i)$'s, not the $VIS(B_i)$'s. Yet we need to know a lot of information about the $VIS(B_i)$'s. Such information is obtained from the $VIS(C_i)$'s. For example, given a value x, the point p on a $VIS(B_i)$ such that $x(p) = x$ is obtained by "probing" the $VIS(C_j)$'s, for every $j = 1, 2, \ldots, i-1$, as follows:

(i) In parallel for every $VIS(C_j)$, $j = 1, 2, \ldots, i-1$, compute the intersection between $VIS(C_j)$ and the vertical line $l_x = x$ (i.e., $VIS(C_j) \cap l_x$), and find the highest point z_j on $VIS(C_j) \cap l_x$.

(ii) Among the $i-1$ points z_j so obtained, compute the highest point (which is the point p that we seek).

Our hypercube algorithm, in fact, performs many such "probes" in a parallel search. Hence we must handle in $O(\log n)$ time the following tasks: (a) Routing the probe values to the sub-hypercubes that store the appropriate $VIS(C_j)$'s, and (b) finding, within every $VIS(C_j)$, the point z_j for each probe value x. As we show next, (a) is handled by using the sorting algorithm [21] and (b) by using the parallel merge algorithm [3].

To compute the one-intersection between (without loss of generality) $VIS(C_i)$ and $VIS(B_i)$ in Step (3) above, we perform $O(1)$ times the following parallel search:

(A) In parallel for every $VIS(C_a)$, $a = 1, 2, \ldots, i$, find $g+1$ vertical lines (by finding $g+1$ probe values) which together partition $VIS(C_a)$ into g subchains of equal size, with two of these $g+1$ lines passing through the endpoints of $VIS(C_a)$.

(B) Sort the $O(g^2)$ vertical lines so obtained according to their x-coordinates.

(C) Compute the highest points on the intersections between these $O(g^2)$ lines and $VIS(C_i) \cup VIS(B_i)$.

(D) Find out that in between which two consecutive vertical lines l' and l'' (in the sorted order of the $O(g^2)$ lines), the one-intersection between $VIS(C_i)$ and $VIS(B_i)$ lies. (Let p' and p'' be the two highest points respectively on l' and l'' computed in Step (C); then one of p' and p'' must be on $VIS(C_i)$ and the other on $VIS(B_i)$.)

Such a parallel search either finds the one-intersection between $VIS(C_i)$ and $VIS(B_i)$ (which is on one of the $O(g^2)$ vertical lines), or allows us to reduce the search range on $VIS(C_a)$ for the one-intersection by a factor of g, for every $a = 1, 2, \ldots, i$. Since $g = n^{1/d}$ for some constant integer $d > 1$, it takes $O(1)$ such parallel searches to locate the one-intersection.

Because of the facts that there are g pairs of $VIS(C_i)$ and $VIS(B_i)$, that each pair generates $O(g^2)$ vertical lines (for probing), and that for each vertical line l, $O(g)$ intersections $l \cap VIS(C_j)$ (for each $j \leq i$) are computed, we have totally $O(g^3)$ lines and need to make $O(g)$ copies for each such line. Therefore, in a parallel search, we must route $O(g^4)$ values on the n-processor hypercube. Such a routing operation can be done in $O(\log n)$ time by using the sorting algorithm [21] *if* $O(g^4) = O(n^{4/d}) = O(n^{\alpha})$, for some positive constant $\alpha < 1$. This can be guaranteed by choosing $d = 5$. Hence all the routing and sorting operations in a parallel search can indeed be carried out in $O(\log n)$ time on the n-processor hypercube.

To compute the intersections between a $VIS(C_i)$ and the vertical lines, first observe that every $VIS(C_i)$ deals with $O(g^c) = O(g^3)$ vertical lines in each parallel search. These $O(g^3)$ intersections can be easily computed in $O(\log n)$ time by first sorting the $O(g^3)$ lines according to their x-coordinates (by using [21]) and then merging the x-coordinates of these lines with those of the vertices of $VIS(C_i)$ (by using [3]). This is because each $VIS(C_i)$ is stored in the sorted order of $\prec$ in a sub-hypercube of size $n^{1-1/d} = n^{4/5} = g^4$.

The rest computation of a parallel search can be easily accomplished by using broadcast and parallel prefix operations. Therefore, a parallel search can be performed in $O(\log n)$ time on the n-processor hypercube. This concludes our discussion on the conquer stage of the visibility algorithm.

5 Monotonicity of a Polygon

For the problem of testing the monotonicity of a simple polygon P, our hypercube algorithm computes a description for all the lines with respect to each of which P is monotone, or report that P is not monotone if no such lines exist. Using the description (which is to be defined below) we compute, it is easy to find a line with respect to which P is monotone, and it is easy to check whether P is monotone with respect to any query line.

Figure 3: Illustrating the polar diagram of a simple polygon P.

Before discussing our hypercube algorithm for testing the monotonicity of P, we first review some definitions and preliminary results in [23] that are needed by our algorithm.

Suppose that the vertices $p_1, p_2, \ldots, p_n$ of P are stored in the processors of the hypercube in the order in which they are visited by a *counterclockwise* walk along the boundary of P starting at p_1, with processor PE_i storing vertex p_i. The edge $\overline{p_i p_{i+1}}$ of P is denoted by e_i. (Throughout this section, we assume that all the indices of the form $n+i$ are equal to indices i, for every $i = 1, 2, \ldots, n$.) The *polar diagram* [23] of P is defined as follows. For each edge e_i of P, draw a semi-infinite ray r_i from the origin O in the direction from p_i to p_{i+1} (see Figure 3). The polar angle of r_i is denoted by $\theta(r_i)$. The polar rays $r_1, r_2, \ldots, r_n$ together partition the polar range $[0, 2\pi)$ into n consecutive wedges (a *wedge* is a sector in the polar diagram bounded by two polar rays). Note, of course, that r_{i+1} may not be adjacent to r_i in the polar ordering. Suppose these consecutive wedges are $\beta_1, \beta_2, \ldots, \beta_n$ in counterclockwise order starting from β_1, where β_1 is the wedge on the counterclockwise side of r_1. Let α_i, $1 \leq i \leq n$, be the wedge from r_i *counterclockwise* to r_{i+1} if the angle from r_i counterclockwise to r_{i+1} is $\leq 180°$, and the wedge from r_i *clockwise* to r_{i+1} otherwise (e.g., see Figure 3). The *multiplicity* of a wedge w is defined to be $|\{\alpha_k \mid w \subseteq \alpha_k, k \in \{1, 2, \ldots, n\}\}|$, i.e., the number of wedges α_k that contain w. It is not difficult to see that for any wedge w, the multiplicity of w is no smaller than 1, since the boundary of P is not self-intersecting. If each of a sequence of consecutive wedges w has multiplicity k, then we say that the wedge which is the union of all the w's in the sequence also has multiplicity k. Two wedges are said to be *antipodal* if their union contains a line passing through the origin O.

The following lemma characterizes the monotonicity of P.

Lemma 2 (Preparata and Supowit [23]) *A simple polygon P is monotone if and only if its*

polar diagram contains at least one pair of antipodal wedges β_i and β_j both of multiplicity 1. *If this is the case, then P is monotone with respect to any infinite line contained in the union of two such antipodal wedges.*

Proof. See [23]. □

The description D that our hypercube algorithm computes for all the lines with respect to each of which P is monotone consists of all the pairs of the antipodal wedges both of whose multiplicities are 1. The bounding rays of the wedges in the description D are stored in processors PE_1, PE_2, ..., PE_m of the hypercube in the sorted order of their polar angles, where m is the total number of the bounding rays in D. Note that the bounding rays of all the wedges in D belong to the set of rays r_1, r_2, ..., r_n. If P is not monotone, then D is simply empty. After D is computed, then given any query line l that passes through the origin O, it is easy to check whether there is a pair of antipodal wedges in D whose union contains l. This is done by performing among the set of wedges in D a binary search for the direction of l; such a binary search takes $O(\log n)$ time and actually performs $O(\log n)$ operations on the n-processor hypercube.

As in [6], the idea for our hypercube algorithm is to reduce the monotonicity-test problem to solving a visibility problem. The reduction we use here is, however, different from that in [6]. Note that one of the two PRAM algorithms given in [6] reduces the monotonicity-test problem to the problem of computing the visibility of nonintersecting line segments from a point in the plane (the best known hypercube algorithm for this visibility problem is not optimal [19]). Our reduction here is from the monotonicity-test problem to a visibility problem which can be solved by using the optimal hypercube algorithm in Section 4.

The reduction that we use is as follows. Without loss of generality, we assume that we are dealing with polar coordinates. Recall that, given the origin O and the polar axis in the plane, the *polar coordinates* of a point $p \neq O$ are respectively $r(p)$ and $\theta(p)$, where $r(p)$ is the length of the line segment $\overline{pO}$ and $\theta(p)$ is the polar angle from the polar axis counterclockwise to the ray that originates at O and passes through p (i.e., $p = (r(p), \theta(p))$).

Let ω be a positive constant (any positive constant will do). Our reduction transforms the polar diagram of P into a simple chain C_P consisting of circular-arcs and line segments. The reduction has the following steps.

1. For every i, $i = 2, 3, \ldots, n$, map ray r_i to two different points $v_i = (\frac{\omega(n-i+1)}{n}, \theta(r_i))$ and $v'_{i-1} = (\frac{\omega(n-i+2)}{n}, \theta(r_i))$. Also map ray r_1 to points $v_1 = (\omega, \theta(r_1))$ and $v'_n = (\omega/n, \theta(r_1))$.

2. For every i, $i = 1, 2, \ldots, n$, connect points v_i and v'_i by a circular-arc A_i, such that A_i is

Figure 4: Illustrating the reduction for testing the monotonicity of P.

on the circle whose center is at O and whose radius is $\frac{\omega(n-i+1)}{n}$, and that A_i is contained in wedge α_i.

3. For every i, $i = 2, 3, \ldots, n$, connect points v'_{i-1} and v_i by line segment $\overline{v'_{i-1}v_i}$.

The outcome of such a reduction is the chain C_P that is simple (i.e., it does not self-intersect) and that consists of n circular-arcs (connecting v_i and v'_i) and $n-1$ line segments (connecting v'_{i-1} and v_i). For example, the result of applying the reduction to the polar diagram of the polygon P in Figure 3 is the chain C_P in Figure 4. Observe that the circular-arcs A_i on C_P have a one-to-one correspondence with the wedges α_i.

We say that a point p on chain C_P is visible from the point at infinity if and only if p is the first point on C_P which is hit by a ray that originates from the point at infinity and that is directed towards the origin O. We have the following lemma for characterizing the visibility of chain C_P.

Lemma 3 *For every wedge α_i, a wedge $w \subseteq \alpha_i$ has multiplicity* 1 *if and only if each point on w's corresponding circular-arc on arc A_i is visible* both *from the origin O and from the point at infinity.*

Proof. Exactly the same as Lemma 3 in [6], and hence omitted. □

Based on Lemma 3, we have the following hypercube algorithm for the monotonicity-test of P.

(1) Obtain the polar diagram of P.

(2) Compute chain C_P from the polar diagram of P.

(3) Compute the portions of chain C_P that are visible both from O and from the point at infinity.

(4) From the outcome of Step (3), find all the pairs of the antipodal wedges whose multiplicities are both 1.

Steps (1) and (2) of the above algorithm can be easily done in $O(\log n)$ time by performing monotone routing [18] and some local operations. Each of the two visibility problems in Step (3) (respectively with the source point O and the point at infinity) can be solved in a way which is similar to that of the visibility algorithm in Section 4. In fact, the visibility problems here in Step (3) are much simpler than the one in Section 4, because the visibility chains in this case do not intersect each other at all. Therefore, the same algorithm as the one in Section 4 works for this case. Actually, the implementation of the visibility algorithm here can be carried out by using a *third-root* (instead of a *fifth-root*) divide-and-conquer strategy. Note that the portions of C_P which are visible from O (resp., the point at infinity) are obtained by our hypercube algorithm in the sorted order of their polar angles. Therefore, the portions of C_P which are visible *both* from O and from the point at infinity can be easily computed by using parallel merge [3] and parallel prefix [18]. For Step (4), a parallel merge [3] is sufficient to find all the pairs of the antipodal wedges with multiplicity 1 (the parallel merge computation for this step has been previously used in [6]).

Each of the four steps of the above algorithm can be implemented in $O(\log n)$ time on the n-processor hypercube. Hence the monotonicity-test problem can be solved optimally on the hypercube, as we have earlier claimed.

6 Other Polygon Problems

We now present optimal hypercube algorithms for solving the other three polygon problems: Computing the convex hull of chain C, computing the maximal elements of the vertices of C, and computing the kernel of polygon P.

6.1 Convex Hull and Maximal Elements of a Chain

Observe that all the points on the visibility chain of C with respect to the source point $(0, +\infty)$ (resp., $(0, -\infty)$) are in the sorted order according to their x-coordinates. Suppose that $VIS(C)$ with respect to each of $(0, +\infty)$ and $(0, -\infty)$ has been computed (by using the hypercube algorithm in Section 4).

The convex hull $CH(C)$ of C can be partitioned into the *upper* convex hull $UH(C)$ and the *lower* convex hull $LH(C)$ of C. The upper (resp., lower) convex hull $UH(C)$ (resp., $LH(C)$) consists of the portion of $CH(C)$ which is visible from $(0, +\infty)$ (resp., $(0, -\infty)$) if $CH(C)$ is treated as an opaque

object. For computing the upper convex hull $UH(C)$, we need to obtain the set VS of the vertices of $VIS(C)$ with respect to $(0, +\infty)$. (Note that VS is available from the description of $VIS(C)$ and is already in sorted order.) Then we use the optimal $O(\log n)$-time, n-processor hypercube algorithm by Miller and Stout [20] to compute the upper convex hull of VS. (Note that the upper convex hull of VS is the same as $UH(C)$.) The lower convex hull $LH(C)$ is computed in a similar fashion. Overall, the time complexity of this algorithm is $O(\log n)$ on an n-processor hypercube.

To compute the maximal elements of the vertices of C, we first obtain $VIS(C)$ with respect to $(0, +\infty)$, and the vertex set VS of $VIS(C)$. Note that the maximal elements of the vertices of C are exactly those of VS. A square-root divide-and-conquer strategy together with parallel prefix [18] will easily compute the maximal elements of VS in $O(\log n)$ time on an n-processor hypercube.

6.2 Kernel of a Polygon

Our hypercube algorithm for computing the kernel of polygon P is based on the geometric observations by Cole and Goodrich [7] for solving this problem in PRAM models. Essentially, Cole and Goodrich [7] show that the problem of computing the kernel of P can be solved by performing $O(1)$ times the following parallel operations: (i) Parallel prefix, (ii) computing the convex hull of a set of points sorted by their x-coordinates, and (iii) merging two sorted lists. All these three operations can be implemented optimally in $O(\log n)$ time on an n-processor hypercube. Specifically, (i) can be done easily [18], (ii) can be done by using the hypercube algorithm of [20], and (iii) can be done by using the merging algorithm of [3]. Therefore, the problem of computing the kernel of a simple polygon can be solved in $O(\log n)$ time on an n-processor hypercube.

7 Final Remarks

We have shown that, as far as many polygon problems are concerned, a sorted output can be obtained optimally on hypercubes even if the input is given sorted by the polygonal chain order (rather than by the x or y coordinates of the vertices).

One problem which our techniques do not solve is that of obtaining the sorted order of the intersections of a line with a simple polygonal chain. This might be possible in logarithmic time, by exploiting the given polygonal chain ordering. Some of the ideas that we have introduced here might be useful in that respect.

References

[1] M. J. Atallah, D. Z. Chen, and H. Wagener. "An optimal parallel algorithm for the visibility of a simple polygon from a point," *J. of the ACM*, 38 (1991), pp. 516–533.

[2] M.J. Atallah, R. Cole, and M.T. Goodrich. "Cascading divide-and-conquer: A technique for designing parallel algorithms," *SIAM J. Comput.*, 18 (3) (1989), pp. 499-532.

[3] K. Batcher. "Sorting networks and their applications," *Proc. the AFIPS Spring Joint Computing Conference*, Vol. 32, 1968, pp. 307–314.

[4] B. Chazelle and D. P. Dobkin. "Intersection of convex objects in two and three dimensions," *J. of the ACM*, 34 (1) (1987), pp. 1-27.

[5] D. Z. Chen. "Efficient geometric algorithms on the EREW-PRAM," *Proc. 28th Annual Allerton Conf. on Communication, Control, and Computing*, 1990, pp. 818–827.

[6] D. Z. Chen and S. Guha. "Testing a simple polygon for monotonicity optimally in parallel," *Proc. of the 7th International Parallel Processing Symp.*, 1993, pp. 326–330. To appear in *Inform. Process. Lett.*

[7] R. Cole and M. T. Goodrich. "Optimal parallel algorithms for polygon and point-set problems," *Proc. 4th Annual ACM Symp. Computational Geometry*, 1988, pp. 211–220. Also appeared in *Algorithmica*.

[8] R. Cypher and C. G. Plaxton. "Deterministic sorting in nearly logarithmic time on the hypercube and related computers," *Proc. 22-nd Annual ACM Symp. on Theory of Computing*, 1990, pp. 193–203.

[9] J. A. Dean and J. R. Sack. "Efficient hidden-line elimination by capturing winding information," *Proc. 23rd Annual Allerton Conference on Communication, Control, and Computing*, 1985, pp. 496-505.

[10] F. Dehne, A. Ferreira and A. Rau-Chaplin, "Parallel fractional cascading on a hypercube multiprocessor," *Proc. 27th Annual Allerton Conf. on Communication, Control, and Computing*, Monticello, Illinois, 1989, pp. 1084–1093.

[11] F. Dehne and A. Rau-Chaplin, "Implementing data structures on a hypercube multiprocessor, with applications in parallel computational geometry." *J. Parallel Distrib. Computing*, Vol. 8, 1990, pp. 367–375.

[12] H. ElGindy and D. Avis. "A linear algorithm for computing the visibility polygon from a point," *J. of Algorithms*, 2 (1981), pp. 186-197.

[13] R. L. Graham and F. F. Yao. "Finding the convex hull of a simple polygon," *J. of Algorithms*, 4 (4) (1983), 324–331.

[14] B. Joe and R. B. Simpson. "Corrections to Lee's visibility polygon algorithm," *BIT*, 27 (1987), pp. 458-473.

[15] D. T. Lee. "Visibility of a simple polygon," *Computer Vision, Graphics, and Image Processing*, 22 (1983), pp. 207-221.

[16] D. T. Lee. "On finding the convex hull of a simple polygon," *Int'l J. Comput. and Inform. Sci.*, 12 (2) (1983), 87–98.

[17] D. T. Lee and F. P. Preparata. "An optimal algorithm for finding the kernel of a polygon," *J. of the ACM*, 26 (1979), pp. 415–421.

[18] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.

[19] P. D. MacKenzie and Q. Stout. "Asymptotically efficient hypercube algorithms for computational geometry," *Proc. of the Third Symp. on the Frontiers of Massively Parallel Computation*, 1990, pp. 8–11.

[20] R. Miller and Q. F. Stout. "Efficient parallel convex hull algorithms," *IEEE Trans. Computers*, C-37 (12) (1988), pp. 1605-1618.

[21] D. Nassimi and S. Sahni. "Parallel permutation and sorting algorithms and a new generalized connection network," *J. of the ACM*, 29 (1982), 642–667.

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

[23] F. P. Preparata and K. J. Supowit. "Testing a simple polygon for monotonicity," *Inform. Process. Lett.*, 12 (1981), pp. 161-164.

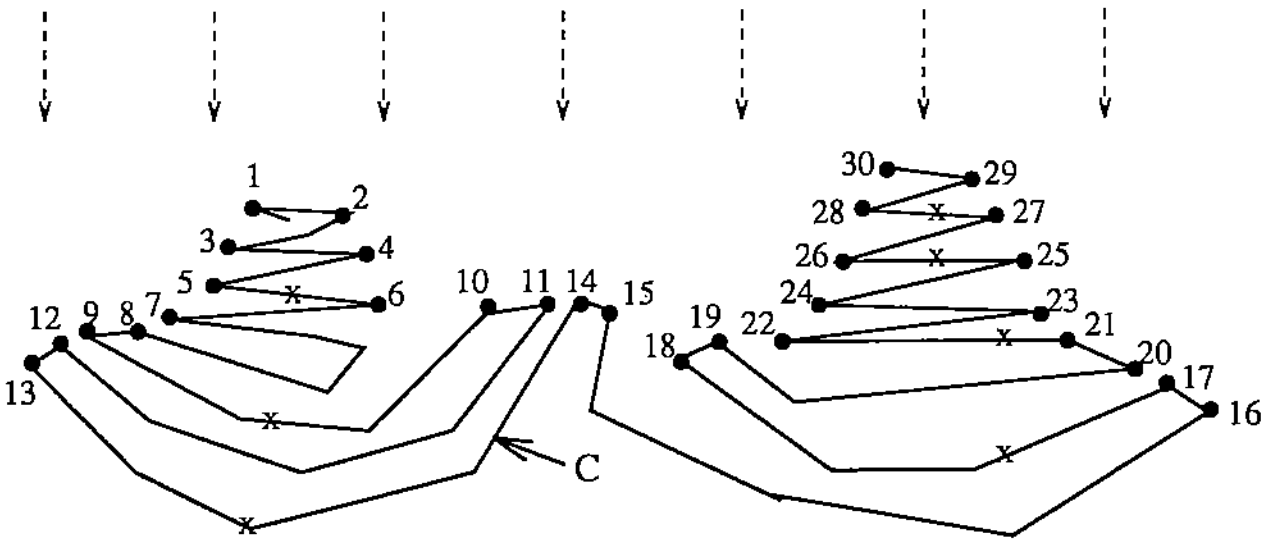
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
C
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

Figure 1

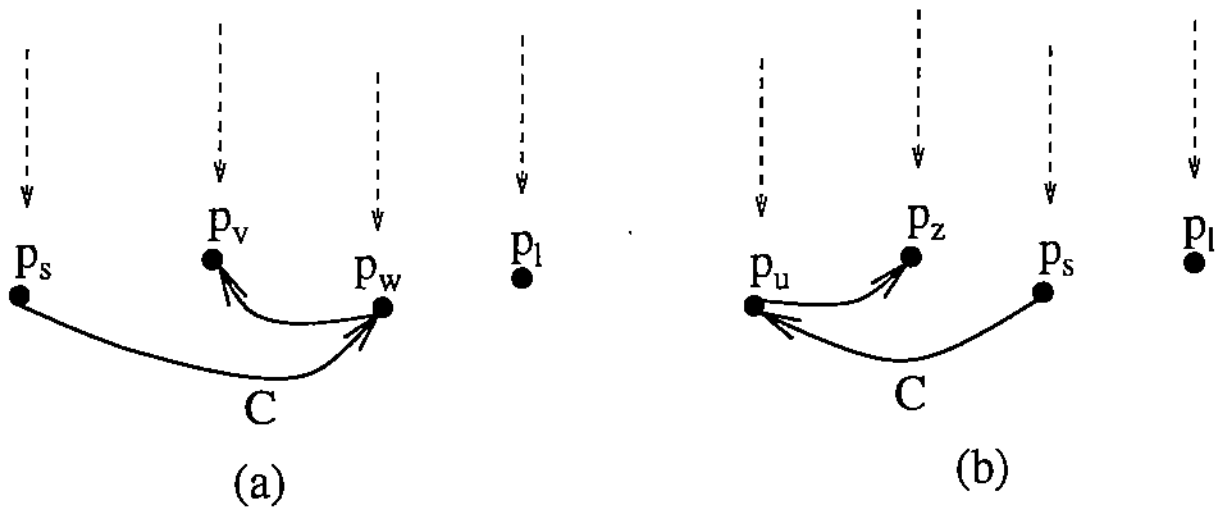

Figure 2

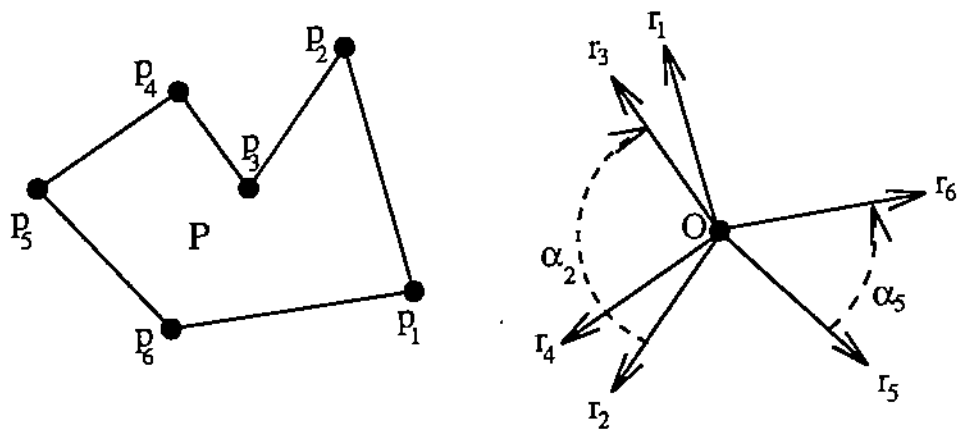

Figure 3

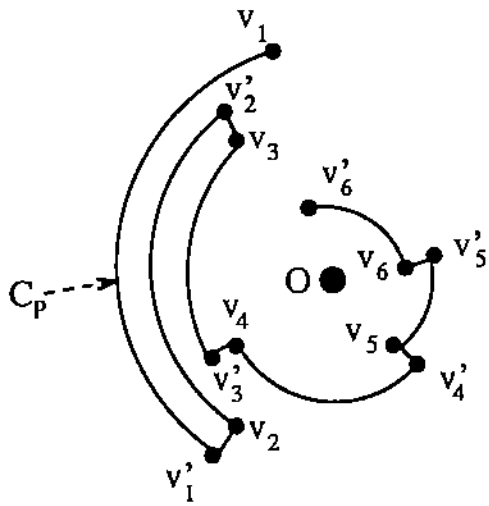

Figure 4