Optimal Parallel Algorithms for Polygon and Point-Set Problems (Preliminary Version)

Richard Cole[1]
Courant Institute, New York Univ., New York, NY 10012

Michael T. Goodrich
Dept. of Computer Science, The Johns Hopkins University, Baltimore, MD 21218

Abstract

In this paper we give parallel algorithms for a number of problems defined on polygons and point sets. All of our algorithms have optimal $T(n) * P(n)$ products, where $T(n)$ is the time complexity and $P(n)$ is the number of processors used, and are for the EREW PRAM or CREW PRAM models. In addition, our algorithms provide parallel analogues to well known phenomena from sequential computational geometry, such as the fact that problems for polygons can oftentimes be solved more efficiently that point-set problems, and that one can solve nearest-neighbor problems without explicitly constructing a Voronoi diagram.

1. Introduction

We present a number of new algorithms for parallel computational geometry [1,2,3,4,7,9,10]. Our goal is to find algorithms that run as fast as possible and are efficient in the following sense: if $P(n)$ is the processor complexity, $T(n)$ the parallel time complexity, and $Seq(n)$ the time complexity of the best known sequential algorithm for the problem under consideration, then $T(n) * P(n) = O(Seq(n))$. If the product $T(n) * P(n)$ achieves the sequential lower bound for the problem, then we say the algorithm is *optimal*. All of our algorithms are optimal in this sense and are for the EREW or CREW PRAM models. The weaker of these two is the EREW PRAM model, the synchronous shared memory model in which simultaneous reads or writes are not allowed. In the CREW PRAM we allow for simultaneous reads. Specifically, our results are the following:

1. Kernel of a simple polygon: $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model.
2. All-nearest neighbors for a set of points in the plane: $O(\log n)$ time using $O(n)$ processors in the EREW PRAM model.
3. All-nearest neighbors for the vertices of a convex polygon: $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model.
4. Convex hull of a set of points in the plane: $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model (using a "cascading calipers" technique).

Our algorithms for problems 1 and 3 show that there is a parallel analogue to a famous phenomenon of sequential computational geometry, namely, that many problems with $\Omega(n \log n)$ lower bounds when defined for arbitrary point sets can be solved in $O(n)$ time when the points are the vertices of a polygon. Our kernel algorithm (problem 1) is based on the discovery of a new way of characterizing the kernel of a simple polygon P in terms of the "curvature" of P. This idea also leads to a new $O(n)$-time sequential algorithm for this problem. Our algorithm for problem 3 is based on a composite of parallel merging, parallel prefix, and broadcasting techniques.

Our algorithm for problem 2 shows that, just as in the sequential case, one can optimally solve the all-nearest neighbor problem without explicitly constructing a Voronoi diagram (for which the best-known parallel algorithm runs in non-optimal $O(\log^2 n)$ time using $O(n)$ processors [1,2]). Our algorithm is based on the novel use of the cascading divide-and-conquer technique [2].

Finally, our convex hull algorithm (for problem 4) is based on a generalization of cascading divide-and-conquer technique which provides a non-trivial parallel analogue to Toussaint's "rotating calipers" paradigm [18].

We present our algorithms, one per section, in the discussion which follows, and conclude with some final remarks and open problems in Section 6.

[1]This research was supported in part by NSF grants DCR-84-01633 and CCR-8702271, and by ONR grant N00014-85-K-0046.

2. Kernel of a Simple Polygon

Let $P = (e_0, e_2, ..., e_{n-1})$ be a listing of the edges of a simple polygon P (with e_0 and e_{n-1} sharing a common endpoint). Each edge of P is given an orientation so that the interior of P is on its left. We let $H(e_i)$ denote the half-plane to the left of the line containing e_i. Given any list Q of oriented edges $e_0, ..., e_{m-1}$, we define the *kernel of* Q, denoted $K(Q)$, to be the intersection of all the half-planes determined by the edges in Q, i.e., $K(Q) = \cap_{i=0}^{m-1} H(e_i)$. Our problem is the following: given an oriented simple polygon P, construct $K(P)$.

Wagener [19] has shown that one can construct the convex hull of a simple polygon in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model. Since one can compute the common intersection of n half-planes by dualization to the convex hull problem [11,16], one may at first think that this problem and the kernel problem have a primal-dual relationship. This is not the case, however, because the dualization methods, even when extended to polygons [11], do not map simple polygons into simple polygons. It is not surprising, then, that our algorithm for the kernel problem is quite different from the convex hull algorithm of Wagener.

We begin our discussion with a few definitions. Let $P[e_i, e_j]$ denote the subchain of P from e_i to e_j, inclusive (edge subscripts are modulo n). Note that since each edge has an orientation, $P[e_i, e_j]$ is well defined and is different from $P[e_j, e_i]$. Given two adjacent edges e_i and e_{i+1} define the *angle between* e_i *and* e_{i+1}, denoted $\alpha_{i,i+1}$, to be the signed angle e_i makes with e_{i+1} when they are translated (as vectors) so as to share a common start vertex, where the angle is positive if we move in a counterclockwise angle in going from e_i to e_{i+1} (again, all subscripts are modulo n). We generalize this definition as follows: Given a subchain $P[e_i, e_j]$ we define the *curvature* of $P[e_i, e_j]$, denoted $\alpha_{i,j}$, to be the sum of all the edge angles from e_i to e_j. This can be expressed symbolically as

$$\alpha_{i,j} = \sum_{k=i}^{j-1} \alpha_{k,k+1}.$$

For completeness, we define $\alpha_{i,i} = 0$ for all $i \in \{0, 1, ..., n-1\}$.

If there are two edges e_i and e_j on P such that $\alpha_{i,j} \geq 3\pi$, then we say that P is a *spiral* polygon. The next lemma establishes an important property of spiral polygons.

Lemma 2.1: *If P is a spiral polygon, then $K(P)$ is empty.*

Proof: Suppose P is a spiral polygon, yet $K(P)$ is non-empty. Since $K(P)$ is non-empty, then P is star-shaped. That is, for each point $p \in K(P)$ the boundary of P is completely visible from p and the vertices of P, as listed around the boundary of P, are sorted radially around p. But, by hypothesis, there is some part of the boundary of P, say $P[e_i, e_j]$, with a curvature of at least 3π. This contradicts one of the following, however: (1) that $P[e_i, e_j]$ is sorted radially around p or (2) that all of $P[e_i, e_j]$ is visible from p. (See Figure 1.) ∎

We can trivially test if P is a spiral polygon in $O(\log n)$ time using $O(n^2)$ processors (by computing all the $\alpha_{i,j}$ values). But, since we only have $O(n/\log n)$ processors at our disposal, our method for determining if P is a spiral polygon needs to be a little more involved. We begin by computing $\alpha_{0,i}$ and $\alpha_{i,0}$ for all $i \in \{0, 1, ..., n-1\}$. This can easily be done in $O(\log n)$ time using $O(n/\log n)$ processors by two simple parallel prefix computations. Recall that a parallel prefix computation is just a reduction of a problem to the problem of computing all the prefix sums $c_k = \sum_{i=1}^{k} a_i$ of a sequence $(a_1, a_2, ..., a_m)$, where the $+$ operation is associative. (See [9] for a survey of this and other parallel techniques.) We also compute four additional quantities:

$$\begin{aligned} f_i &= \max_{0 \leq j \leq i-1} \alpha_{0,j}, \\ b_i &= \max_{i \leq j \leq n} \alpha_{j,0}, \\ low &= \min_{0 \leq j \leq n-1} \alpha_{0,j}. \end{aligned}$$

Again, all subscripts are modulo n. As with the $\alpha_{0,i}$'s and $\alpha_{i,0}$'s, these quantities can easily be computed in $O(\log n)$ time using $O(n/\log n)$ processors. The next lemma characterizes spiral polygons in terms of these quantities.

Lemma 2.2: *P is a spiral polygon if and only if (1) $b_i + f_i \geq 3\pi$ for some i, or (2) $f_{n-1} - low \geq 3\pi$.*

Proof: The "if" part of the proof is obvious. So, for the "only if" part, suppose P is a spiral polygon. Then there is some subchain $P[e_i, e_j]$ which has a curvature of at least 3π. That is, $\alpha_{i,j} \geq 3\pi$. There are two cases. Case 1: e_0 is in $P[e_i, e_j]$. In this case, since $\alpha_{i,j} \geq 3\pi$, $b_i + f_j \geq 3\pi$. But this implies that there is some k such that $b_k + f_k \geq 3\pi$. Case 2: e_0 is not in $P[e_i, e_j]$. In this case, $f_{n-1} - low \geq 3\pi$. If this were not so, there would be no way that $\alpha_{i,j} \geq 3\pi$, since i must be less than j in this case. This establishes the lemma. ∎

Let Q_1 be the lexicographically-first maximal increasing subsequence of $(e_0, ..., e_{n-1})$, using the f_i's as weights, and let Q_2 be the lexicographically-first maximal increasing subsequence of $(e_0, e_{n-1}, ..., e_1)$, using the b_i's as weights. Recall that a lexicographically-first maximal increasing subsequence is defined by placing the first item in the list in the set, then scanning through the list adding an item to the set each time its label is bigger than the biggest label encountered thus far. The following lemma establishes an even stronger relationship between $K(P)$ and the curvature properties of P.

Lemma 2.3: *If P is not a spiral polygon, then $K(P) = K(Q_1) \cap K(Q_2)$.*

Proof: Since Q_1 and Q_2 are subsets of P, $K(P) \subseteq K(Q_1) \cap K(Q_2)$. So, we have yet to show that $K(Q_1) \cap K(Q_2) \subseteq K(P)$. Clearly, if $K(Q_1) \cap K(Q_2) = \emptyset$, then we are done; so suppose $K(Q_1) \cap K(Q_2) \neq \emptyset$. The proof is by contradiction. Suppose $K(P)$ is properly contained in $K(Q_1) \cap K(Q_2)$. Then there is an edge e_i of P with $e_i \notin Q_1 \cup Q_2$ and such that $H(e_i) \cap K(Q_1) \cap K(Q_2)$ is a proper subset of $K(Q_1) \cap K(Q_2)$. Let e_j be the edge closest to e_i in P such that $f_j > f_i$ and with $j < i$. Since e_i is not in Q_1, the edge e_j must exist.
Claim: e_i is not contained in $H(e_j)$.
Proof of claim: It is sufficient to show that if e_i intersects $H(e_j)$, then P is a spiral polygon (which would be a contradiction), so suppose e_i intersects $H(e_j)$. We begin the proof of our claim by noting that the only point of intersection of e_{j+1} and $H(e_j)$ is the common vertex e_j and e_{j+1} share. If this were not the case, then e_j would not be the closest edge to e_i such that $f_j > f_i$ and $j < i$. Let e_l be the first edge in $P[e_j, e_i]$ which intersects $H(e_j)$, and let v be a vertex on $P[e_j, e_l]$ such that there is a line T parallel to e_j and tangent to $P[e_j, e_l]$ at v with all of $P[e_j, e_l]$ on the same side of T as $H(e_j)$. Since $P[e_j, e_l]$ is a finite chain beginning and ending in $H(e_j)$, the vertex v and line T must exist. Let e_m and e_{m+1} be the edges of $P[e_j, e_l]$ incident to v. The intersection of $H(e_m)$ and $H(e_{m+1})$ lies on the opposite side of T as $H(e_j)$. If this were not so (i.e., the intersection of $H(e_m)$ and $H(e_{m+1})$ lies on the same side of T as $H(e_j)$), then in going from e_m to e_{m+1} one makes a left turn. But, since T is parallel to e_j, this implies that $f_m > f_j$, which contradicts the definition of e_j. Thus, the intersection of $H(e_m)$ and $H(e_{m+1})$ lies on the opposite side of T as $H(e_j)$. But this forces $P[e_l, e_j]$ to have a cumulative amount of turning greater than 3π. Which, in turn, implies that P is a spiral polygon. (See Figure 2.) [End of proof of claim.]

Let e_k be the edge closest to e_i in P such that $b_k > b_i$ and $i < k$. As with e_j, e_k must exist, because e_i is not in Q_2. By an argument similar to the proof of the above claim we have that e_i is not contained in $H(e_k)$. These two facts imply that the edge e_i is not contained in $H(e_j) \cap H(e_k)$. But this implies that $H(e_j) \cap H(e_i) \cap H(e_k) = H(e_j) \cap H(e_k)$. In other words, $H(e_i) \cap K(Q_1) \cap K(Q_2)$ is not a proper subset of $K(Q_1) \cap K(Q_2)$, which of course is a contradiction. Therefore, $K(P) = K(Q_1) \cap K(Q_2)$. ∎

The above lemmas immediately give us the outline of our algorithm for constructing $K(P)$: test if P is a spiral polygon, and, if it is not a spiral polygon, construct $K(Q_1)$ and $K(Q_2)$ and their intersection.

We have already described how to test if P is a spiral polygon or not. So suppose P is not a spiral polygon. We begin by constructing Q_1 and Q_2. This can be done by yet another parallel prefix computation in $O(\log n)$ time using $O(n/\log n)$ processors (by computing, for each edge e_i in the list in question, the maximum prefix (or suffix) label of the edges preceding e_i). Note that the lists Q_1 and Q_2 are sorted by slopes. In addition, the list Q_1 (resp., Q_2) can easily be divided into $O(1)$ lists such that the range of label values in each list is at most π (this takes at most $O(\log n)$ time using $O(n/\log n)$ processors). By appropriately translating the origin for the edges in each of these lists so that it is contained in their common intersection we can then compute $K(Q_1)$ and $K(Q_2)$ in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model. The method is to use the dualization method of [11,16] to dualize to the problem of constructing the convex hull of a sorted point set, which can be solved in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model [10,19]. We conclude the algorithm by then computing the intersection of the two convex polygons $K(Q_1)$ and $K(Q_2)$ in $O(\log n)$ time using $O(n/\log n)$ processors by using parallel merging [6] to implement the sequential algorithm of Shamos [17]. We summarize with the following theorem.

Theorem 2.4: *Given an n-edge simple polygon P one can construct the kernel of P in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model.* ∎

3. All-Nearest Neighbors for a Point Set

Given a set S of n points in the plane, the problem is to find the nearest neighbor point of each point in S. For any point q let $N(q)$ denote q's nearest neighbor. Our algorithm runs in $O(\log n)$ time using $O(n)$ processors. We describe how to implement our algorithm in $O(n)$ space in the CREW PRAM model and then outline how it could be implemented in $O(n \log n)$ space in the EREW PRAM model.

Our algorithm is based on two non-trivial applications of the cascading divide-and-conquer technique of Atallah, Cole, and Goodrich [2]. We briefly review this technique as it applies to our problem.

3.1. A Review of Cascading Divide-and-Conquer

Suppose we are given a complete binary tree T such that there is an item (from some universe) stored at each leaf. For each node v of T we recursively define sets $A_{1,v}$, $A_{2,v}$, ..., $A_{k,v}$ in terms of sets stored at the children of v (these definitions depend on the application). Initially, the $A_{i,v}$'s are only constructed for the leaf nodes v of T. Let v be an internal node in T with children u and w. Given a sorted array A and a function f, we use the notation $f(A)$ to denote the array defined by applying f to each element of A, i.e., $f(A)[i] = f(A[i])$. We say such a function is *monotone on* A if $A[i] \leq A[j]$ implies $f(A[i]) \leq f(A[j])$. We place a restriction on the array definitions, namely, that for any internal node v the definition of each $A_{i,v}$ have the following form:

$$A_{i,v} = \bigcup_{j \in I(u)} f_u(A_{j,u}) \cup \bigcup_{j \in I(w)} f_w(A_{j,w}),$$

where $I(u)$ and $I(w)$ are subsets of $\{1, 2, ..., k\}$, and f_u and f_w are monotone. The functions f_u and f_w can be thought of as "identity changing" rules, and can often be used to avoid using set difference operations in the definition of any $A_{i,v}$. It is often very useful to also allow each element of an $A_{i,v}$ to have a label associated with it. So, for each element $A_{i,v}[k]$ let $L_{i,v}[k]$ be the corresponding label (which may actually be a vector of labels). For any element a and set B let $pred(a, B)$ (resp., $succ(a, B)$) denote the predecessor (resp., successor) of a in B if a is not in B, and simply a, if $a \in B$. Let $rank(a, B)$ denote the rank of $pred(a, B)$ in B, and let $F_{j,v}$ be a shorthand for the array $f_v(A_{j,v})$. We place a restriction on the label definitions, as well, namely, that the definition of $L_{i,v}[k]$ for any node v be expressed as the sum of labels of the form

1. $L_{j,u}[rank(A_{i,v}[k], F_{j,u})]$,
2. $L_{j,w}[rank(A_{i,v}[k], F_{j,w})]$, or
3. $L_{j,v}[rank(A_{i,v}[k], A_{j,v})]$,

so long as there are no circular definitions (j is a free variable) and the plus (+) operation can be computed in $O(1)$ time using a single processor. If the definitions of value arrays $A_{i,v}$ and label arrays $L_{i,v}$ have these forms, then we say that they are *cascading*. If we also allow for the definition of $L_{i,v}[k]$ to include labels of the form $L'_{j,v}[rank(A_{i,v}[k], A_{j,v})]$, where, given $A_{j,v}$, any such L' label can be computed $O(\log n + \lceil |A_{j,v}|/p \rceil)$ time using p processors in the EREW PRAM model, and introducing such labels allows us to define the L labels so that any array location can be accessed by at most one processor at a time, then we say the labels are *EREW-computable.* Atallah, Cole, and Goodrich [2] prove the following theorem:

Theorem 3.1: [2]: *Given a complete binary tree T with cascading definitions for sets $A_{i,v}$ and labels $L_{i,v}$ defined for each node v in T, then $A_{i,v}$ and $L_{i,v}$ can be constructed for each node v in T, level by level, in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the CREW PRAM model (if $A_{i,v}$ and $L_{i,v}$ are required for all levels simultaneously, then this of course requires $O(n \log n)$ space). If the labels are EREW-computable, then all the $A_{i,v}$'s and $L_{i,v}$'s can be constructed in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors in the EREW PRAM model.* ∎

Having reviewed this powerful technique, let us return to the problem at hand.

3.2. All Nearest-Neighbor Algorithm

Let us give a brief overview of the two phases of our algorithm. In phase one we will determine, for each $q \in S$, an approximation to $N(q)$, the nearest-neighbor ball centered at q. Specifically, we will determine a ball around each q, which we call the *neighborhood ball* about q, whose radius is the distance between q and the closest point q has "encountered" during the cascading merge procedure. During the second phase we construct for each point q a list $C(q)$ which contains points of S which may have q as their nearest neighbor. We call $C(q)$ the *candidate list* for q. It is easy to show that for any point q there can be at most six other points q' such that q is the nearest neighbor of q'. Thus, $C(q)$ need never contain more than six points. Our algorithm constructs all possible $C(q)$ lists and then performs a post-processing step to eliminate any pairs which are not nearest-neighbor pairs. The details follow.

In Phase 1 we construct, for each q in S, the neighborhood ball centered at q, denoted $B(q)$. For simplicity, let us assume that the points have distinct x-coordinates; one can easily modify our algorithm for the general case. We begin by sorting the points in S into increasing order by x-coordinates; let $S = (q_1, q_2, ..., q_n)$ denote this list. This can be done in $O(\log n)$ time using $O(n)$ processors in the EREW PRAM model [8]. We then build a complete binary tree T which has the points $q_1, q_2, ..., q_n$ as leaves (listed from left to right). For each node v in T let $Y(v)$ denote the points stored in descendents of v sorted by y-coordinates, and let $depth(v)$ denote the depth of v (with the root being at depth 0). With each point q in $Y(v)$ we store a label $b(q)$. At the end of the cascading procedure the label $b(q)$ will store the name of the point which is closest to q of all the points which q has "encountered." Specifically, for each leaf node v, which, say, stores the point q_i, we initialize $b(q_i)$ to be the closer of q_{i-1} and q_{i+1} to

q_i. For each internal node v, with children u and w, we define $b(q)$ for each $q \in Y(v)$ to be the closer of the old value of $b(q)$ and the point in $\{pred(q, Y(u)), succ(q, Y(u)), pred(q, Y(w)), succ(q, Y(w))\}$ closest to q.

Note that the definitions of $Y(v)$ and $b(q)$ can easily be written so as to be cascading. Moreover, since the value of the label $b(q)$ depends only on the old value of $b(q)$ and the points in $\{pred(q, Y(u)), succ(q, Y(u)), pred(q, Y(w)), succ(q, Y(w))\}$, it is EREW-computable. Thus, we have the following:

Lemma 3.2: *Given a list of points $S = (q_1, q_2, ..., q_n)$ the label $b(q_i)$ can be computed for each point $q_i \in S$ in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the CREW PRAM model, or in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors in the EREW PRAM model.* ∎

We define $B(q)$, the *neighborhood* ball centered at q, to be the region in $\Re^2$ consisting of all points q' such that $d(q, q') \leq d(q, b(q))$. In Phase 2 we refine each $B(q)$ into $N(q)$, the nearest-neighbor ball centered at q. Since the points in S all have distinct x-coordinates, we can partition the leaves of T by placing a vertical dividing line between q_i and q_{i+1} for $i = 1, 2, ..., n-1$. With each node v in T we associate a slab Π_v which is the region bounded by the two vertical dividing lines which separate the points stored in the descendents of v from the rest of the points in S. For each node v in T let $left(v)$ (resp., $right(v)$) denote the left (resp. right) vertical boundary of the slab Π_v. We define the following lists for each node $v \in T$:

$$\begin{aligned} L(v) &= \{q \in Y(v) \ : \ B(q) \cap left(v) \neq \emptyset\} \\ R(v) &= \{q \in Y(v) \ : \ B(q) \cap right(v) \neq \emptyset\} \end{aligned}$$

That is, $L(v)$ (resp., $R(v)$) consists of the points whose neighborhood ball intersects the left (resp., right) boundary of the slab Π_v. Our method for refining the $B(q)$'s into $N(q)$'s (i.e., Phase 2) involves a second application of the cascading divide-and-conquer method. In this second merge we not only compute $Y(v)$ for each node v but also $L(v)$ and $R(v)$, all sorted by increasing y-coordinates. Unfortunately, $L(v)$ and $R(v)$ may be proper subsets of $Y(v)$. Thus, in order for us to find cascading definitions of $L(v)$ and $R(v)$ we will need to use some kind of re-naming scheme, i.e., we need to employ identity-changing monotone functions in the recursive definitions of $L(v)$ and $R(v)$. For each q in S we define $l(q)$ (resp., $r(q)$) to be the depth of the lowest node v of T (i.e., the node nearest the root) such that v is an ancestor of the leaf storing q and $B(q)$ intersects the left (resp., right) vertical boundary line for v. In addition, for each node v in T and each point q in $Y(v)$ we maintain a label $lnext(q)$ which stores the point q' which has the smallest y-coordinate from among all those points in $\{q' \ : \ y(q') > y(q) \text{ and } l(q') < l(q)\}$. Informally, $l(q)$ (resp., $r(q)$) determines the level in T such that, for all points which cascade to levels above this level and are not in $Y(q)$ q cannot have any of them as a nearest neighbor. We define a pointer $rnext(q)$ for each q in $Y(v)$ similarly. For any node v in T with left child u and right child w we define functions f_u and f_w as follows: $f_u(q) = q$ if $r(q) < depth(u)$ and $f_u(q) = rnext(q)$ otherwise; and $f_w(q) = q$ if $l(q) < depth(w)$ and $f_w(q) = lnext(q)$ otherwise. These functions enable us to give cascading definitions of $R(v)$, $L(v)$.

Lemma 3.3: *Let v be an internal node of T with left child u and right child w. Then we have the following (cascading) definitions of $L(v)$ and $R(v)$:*

$$\begin{aligned} L(v) &= L(u) \cup f_w(L(w)) \\ R(v) &= f_u(R(u)) \cup R(w) \end{aligned}$$

Proof: The functions f_u and f_w are clearly monotone. The proof of the lemma is based on a simple induction argument, which is omitted. ∎

We must also show that the *lnext* and *rnext* labels have cascading definitions.

Lemma 3.4: *Let v be an internal node of T with left child u and right child w. Suppose $q \in Y(u)$, and let $q' = succ(q, f_w(L(w)))$. Then the following is a cascading definition of lnext (the definition of rnext is similar).*

$$lnext(q) =$$

$$\begin{cases} lnext(q) & \text{if } y(lnext(q)) < y(q') \qquad (1) \\ lnext(q) & \text{if } l(q') = l(q) \\ & \text{and } y(lnext(q)) < y(lnext(q')) \qquad (2) \\ q' & \text{if } l(q') < l(q) \\ & \text{and } y(q') < y(lnext(q)) \qquad (3) \\ lnext(q') & \text{if } l(q') = l(q) \\ & \text{and } y(lnext(q')) < y(lnext(q)) \qquad (4) \end{cases}$$

Proof: Follows immediately from the definitions of the *lnext* labels and q'. (See Figure 3.) ∎

These definitions are EREW-computable as well, since the $l(q)$ and $r(q)$ values can be computed a priori. Actually, in the EREW case we needn't bother with the *lnext* and *rnext* pointers, since we can construct $R(v)$ and $L(v)$ directly from $Y(v)$ and the $l(q)$ and $r(q)$ values (by a simple data-compression computation).

These definitions enable us to construct the $L(v)$'s and $R(v)$'s in a cascading fashion, and we use these lists to construct candidate lists $C(q)$ for each point q, which contain the (at most 6) points which may have q as their nearest neighbor. Let $SW(q)$ denote the region of $\Re^2$ consisting of all points q' such that $x(q') < x(q)$ and $y(q') < y(q)$, i.e., all points which are south-west of q. Define $SE(q)$, $NW(q)$, and $NE(q)$ similarly. For each point q in $Y(v)$ we define four pointers (labels):

$$\begin{aligned} sw(q) &= \text{point w/ max. } y\text{-coor. in } SW(q) \cap Y(v) \\ se(q) &= \text{point w/ max. } y\text{-coor. in } SE(q) \cap Y(v) \\ nw(q) &= \text{point w/ min. } y\text{-coor. in } NW(q) \cap Y(v) \\ ne(q) &= \text{point w/ min. } y\text{-coor. in } NE(q) \cap Y(v) \end{aligned}$$

These labels all have cascading definitions. We use these labels and the L and R lists to maintain $C(q)$ during the cascading. In this case, if q comes from $Y(w)$, we can compute the new list $C(q)$ at v given the old list $C(q)$ at w and at most four points in $Y(u)$. (The definition is similar if q comes from $Y(u)$.)

Lemma 3.5: *Let v be a node in T with left child u and right child w and let q be a point in $Y(v)$. If $q \in Y(u)$, then the only points in $Y(w)$ such that q could possibly be their nearest-neighbor are $pred(q, L(w))$, $sw(pred(q, L(w)))$, $succ(q, L(w))$, and $nw(succ(q, L(w)))$. (See Figure 4.) If $q \in Y(w)$, then the only points in $Y(w)$ such that q could possibly be their nearest-neighbor are $pred(q, R(u))$, $se(pred(q, R(u)))$, $succ(q, R(u))$, and $ne(succ(q, R(u)))$.*

Proof: WLOG, we prove for $q \in Y(u)$ that the only points p in $Y(w)$ with $y(p) \geq y(q)$ such that q could be the nearest neighbor of p are $succ(q, L(w))$ and $nw(succ(q, L(w)))$. Let ℓ be the vertical line separating $Y(u)$ and $Y(w)$ and let the origin, denoted o, be placed at the intersection of ℓ and the horizontal line containing $succ(q, L(w))$. Furthermore, let $p_0 = (x_0, y_0) = (x_0, 0) = succ(q, L(w))$ and $p_1 = (x_1, y_1) = nw(succ(q, L(w)))$. Suppose there is a point $p_2 = (x_2, y_2)$ such that the circle C centered at p_2 with radius $\min\{d(p_2, p_1), d(p_2, p_0)\}$ contains the origin o. This is a necessary condition for p_2 to contain q. Since C contains the origin and p_2 is above p_0 by definition, $x_2 \leq x_0$ and $y_2 \geq y_0$. Note also, then, that $y_2 \geq y_1$, by the definition of p_1. Since p_0 is in $L(w)$ by definition, $B(p_0)$ contains the origin. In addition, the radius of $B(p_0)$ is at most $d(p_0, p_1)$, since p_1 must have been one of the points encountered by p_0 in phase 1 of our algorithm. Thus, $x_1 \leq y_1$. This in turn implies that $x_1 < y_2$, since $x_1 \leq y_1 \leq y_2$ and one of these inequalities must be strict. Therefore, $d(p_2, p_1) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < \sqrt{x_2^2 + y_2^2} = d(p_2, o)$. But this means that C cannot contain the origin, which is a contradiction. ■

Thus, while performing the cascading merging procedure the generic update step is that we have an old $C(q)$ list and are given at most four new points to consider. Since $|C(q)| \leq 6$, there can be at most a total of ten points in this collection, from which we must determine which ones can possibly have q as their nearest neighbor. These points can be determined by solving the all-nearest neighbor problem for this collection of at most 10 points with a single processor in $O(1)$ steps. Thus, we have the following:

Lemma 3.6: *Given a set $S = \{q_1, q_2, ..., q_n\}$ of points in the plane, we can compute $C(q_i)$ for each q_i in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the CREW PRAM model, or in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors in the EREW PRAM model.* ■

Let N be the set of all pairs (q, q') such that $q \in C(q')$. Since $|N| \leq 6n$ we can sort the pairs in N lexicographically in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the EREW PRAM model. We complete the algorithm by performing a simple bottom-up minimum-finding computation to find the nearest neighbor point of each point in S. Thus, we have the following theorem:

Theorem 3.7: *Given a set S of n points in the plane we can compute the nearest-neighbor in S of each point in S in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the CREW PRAM model, or in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors in the EREW PRAM model.* ■

4. All-Nearest Neighbor Problem for a Convex Polygon

In this section we show how to find the nearest-neighbor vertex of each vertex on a convex polygon in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model.

Let $P = (v_1, v_2, ..., v_n)$ be the clockwise listing of the vertices of a convex polygon. A polygonal chain C has the *semi-circle* property if when v_i and v_j are two farthest vertices in C, then all the vertices of C are contained in a circle with diameter $d(v_i, v_j)$. Let v_a and v_c be two farthest vertices of P, and let v_b (resp. v_d) be a vertex which is farthest to the left (resp. right) of the line (v_a, v_c). Lee and Preparata [14] show that the vertices v_a, v_b, v_c, and v_d partition P into four polygonal chains $C_1 = (v_a, ..., v_b)$, $C_2 = (v_b, ..., v_c)$, $C_3 = (v_c, ..., v_d)$, and $C_4 = (v_d, ..., v_a)$, such that each chain has the semi-circle property. They also show that the nearest-neighbor vertex in C_i of any vertex v_j in C_i is either v_{j-1} or v_{j+1}.

One can determine v_a and v_c by using parallel merging [6] to implement the algorithm of Shamos [17] in $O(\log n)$ time using $O(n/\log n)$ processors (see [9] for details). It is an easy matter to then find the vertices v_b and v_d in $O(\log n)$ time using $O(n/\log n)$ processors by a simple maximum-finding algorithm. We can then solve the all nearest-neighbor problem for each of the polygonal chains in $O(\log n)$ time using $O(n/\log n)$ processors, since the nearest-neighbor vertex in the chain C_i of each vertex v_j in

C_i is either v_{j-1} or v_{j+1} [14]. The rest of the computation is as follows: we first "merge" the subproblem solutions to C_1 and C_2 (resp. C_3 and C_4), and then merge the two subproblem solutions separated by the line (v_a, v_c).

Let us concentrate on the generic merge step. We are given two sets of points S_1 and S_2 separated by a line L such that we have solved the all nearest-neighbor problem for each set. In addition, we are given S_1 and S_2 listed in sorted order along L. Without loss of generality, we assume that L is a vertical line and the points in S_1 and S_2 are listed by non-decreasing y-coordinates. For simplicity, we also assume the y-coordinates are distinct; our results are easily modified for the general case.

Let $d_i(p)$ denote the distance from a point p to its nearest-neighbor in S_i, and let $N_i(p)$ denote the $d_i(p)$-ball centered at p. It is known [5] that each point on L can intersect at most four $N_i(p)$'s for any $i \in \{1, 2\}$. Since we assumed that the all nearest-neighbor problem has already been solved for S_1 and S_2, we can construct, for $i \in \{1, 2\}$, the sorted list S'_i which consists of all the points in S_i whose $d_i(p)$-ball intersects L by compressing out all the points whose $d_i(p)$-ball doesn't intersect L. This can be done in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model by a parallel prefix computation [12,13].

To merge the solutions to the two subproblems we need to find for each point q in S'_2 a point p in S_1 such that p is the closest of all points contained in $N_2(q)$, if there are such points p. We begin by merging the list S_1 with the list S'_2. For each p in S_1, this gives us the predecessor of p in S'_2, which we denote by $pred(p, S'_2)$. Since any point on L intersects at most four $N_2(q)$'s, any point p in S_1 intersects at most four $N_2(q)$'s as well. Moreover, the only q's in S'_2 whose $d_2(q)$-ball could possibly contain p must be within four positions of $pred(p, S'_2)$ in S'_2. If we had $O(n)$ processors at our disposal and we were working in the CREW (concurrent-read) PRAM model, it would be a simple matter to complete this merging procedure. But using only $O(n/\log n)$ processors in the EREW PRAM model it must be a little more involved, because for any point q in S'_2 there may be many p's in S_1 that we wish to compare q to.

Recall that for each point p we wish to examine up to eight points in S'_2. Our computation consists of eight rounds, where in each round we examine one of the eight points in S'_2 for each p in S_1. For each $p \in S_1$ we examine the points in S'_2 associated with p in order by increasing y-coordinates. We will also be maintaining a label $closest(q)$ for each point $q \in S'_2$, which identifies the point p in S_1 which is closest to q from all points in S_1 compared to q so far. Initially, $closest(q)$ is ∞ for all q in S'_2.

Let us concentrate on the computation for a single round. Let S_q denote the set of all points p in S_1 such that q is the point in S'_2 we wish to examine for p in this round. Since we examine the points in S'_2 associated with each p in S_1 by increasing y-coordinates, the points in S_q comprise a contiguous subarray of S_1. Thus, we can use a parallel prefix computation to determine the subarray S_q in S_1 for all q in S_2 (some S_q's may be empty) in $O(\log n)$ time using $O(n/\log n)$ processors. We can then perform a broadcast and find-minimum operation to find a point in S_q which is closest to q. We then let $closest(q)$ be the closer of this point and the previous $closest(q)$ value. This broadcast and minimum-finding step can also be performed in $O(\log n)$ time using $O(n/\log n)$ processors, and completes the computation for this round. When the eight rounds have completed, we will have solved the all nearest-neighbor problem for each q in S_2, since we will have compared q to all points p in S_1 which are contained in $N_2(q)$ (recall that if $q \in S_2 - S'_2$, then this is true vacuously). We then repeat this procedure to solve the all nearest-neighbor problem for each p in S_1, by merging S'_1 with S_2. Thus, we have established the following:

Theorem 4.1: *Given a convex polygon P the nearest-neighbor vertex of each vertex on P can be determined in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model, which is optimal.* ∎

5. A Cascading Algorithm for Convex Hull Construction

In this section we describe our algorithm for constructing the convex hull of a set of points. Our algorithm runs in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model. There has been considerable prevous work on this problem, resulting in a number of algorithms running in $O(\log n)$ time using $O(n)$ processors [1,3,4,10,19]. The algorithm we present in this section has the same complexity as these algorithms, but differs from them in that it is based on the elegant "rotating calipers" (or "merging slopes") technique of Toussaint [18], and in fact provides the first non-trivial parallel analogue to that technique. (By substituting known parallel merging methods for the sequential one used in the convex hull algorithm by Toussaint, one can trivially get an algorithm running in $O(\log^2 n)$ time using $O(n/\log n)$ processors [6].)

Given a set of n points in the plane, the convex hull problem is to construct a representation of the smallest convex set (a polygon) which contains all these points. One can divide this problem in two by considering the boundary of the convex hull to consist of two pieces, an upper hull and a lower hull. The upper hull is that piece visible from above (i.e., $+\infty$) and the lower hull is that piece visible from below.

Consider the problem of constructing the upper hull of a set of n points (the problem of constructing the lower hull is clearly similar). For simplicity, we assume that the points have distinct x-coordinates, that no three are co-linear, and that the number of points is a power of two (it is straightforward to modify our algorithm for the general case). Our algorithm is based on the following divide-and-conquer approach [18]: Suppose we have a two disjoint upper hulls separated by a vertical line. We keep the edges of each hull sorted by slope. In order to find their common tangent we merge the two sets of edges. Suppose edges e and g come from the left hull and edge f from the right, in the order $e > f > g$ (by slopes). If the line containing f is below the common vertex of e and g then f cannot be on the hull. This eliminates exactly those edges that do not belong on the hull, and leaves two contiguous lists of edges on each side of the dividing line. The common upper tangent is determined by creating an edge which joins the rightmost vertex of the "surviving" hull on the left to the leftmost vertex of the surviving hull on the right.

To do this by cascading merging we begin by sorting the input points by x-coordinates; let $S = (q_1, q_2, ..., q_n)$ denote this list. We construct a complete binary tree T such that each leaf node v_i contains the list $H_i = (\langle(x(q_{2i-1}), -\infty), q_{2i-1}\rangle, \langle q_{2i-1}, q_{2i}\rangle, \langle q_{2i}, (x(q_{2i}), -\infty)\rangle)$, for $i = 1, 2, ..., n/2$. In other words, H_i is the upper hull determined by the edge $\langle q_{2i-1}, q_{2i}\rangle$. The main idea of this application of the cascading divide-and-conquer technique is to define an H_v list for each internal node v in such a way that (1) H_v contains, sorted by slopes, the edges of the upper hull of the edges stored in the descendents of v and (2) the construction of all the H_v's can be pipe-lined. That is, they are defined so that while performing the merge at some node v we can obtain some partial information that can be passed immediately to v's parent to allow v's parent to begin its merge. Our algorithm will consist of a sequence of $3\lceil\log n\rceil$ stages. For each internal node v, with left child u and right child w, at stage t we define H_v as follows:

$$H_v = SAMP_v(H_u) \cup SAMP_v(H_w),$$

where $SAMP_v(A)$ denotes *v-sample* of A and $\cup$ denotes the operation of merging two lists of edges sorted by slopes into a combined list sorted by slopes. The v-sample of a set A is defined to be the list consisting of every fourth element of A, so long as the height of v is greater than $t/3$. When the height of v is $t/3$ we say that v is *full* and define $SAMP_v(A)$ to consist of every fourth element of A, as before, but in the next stage $(t+1)$ we define $SAMP_v(A)$ to consist of every other element of A, and in the stage after that $(t+2)$ we define it to be all of A. Thus, initially (at time $t = 0$) only the leaf nodes are full. Then, after every 3 stages the level just above the previous full level becomes full.

This is not quite enough, however, for the current definition of the H_v's does nothing more than merge all the edges stored in the leaves of T by sorted slopes. We modify the definition slightly, so that at the moment when a node v becomes full, then we will perform some extra computations to make H_v be the upper hull of all the edges which are stored in the descendants of v. Suppose v is a node in T with left child u and right child w and that v has just become full. By induction, H_u (resp. H_w) is the upper hull for u (w) stored by sorted slopes. Thus, by merging these two lists we can determine the common upper tangent of H_u and H_w. Specifically, let t be the common upper tangent of H_u and H_w, let e (resp. f) be the first edge in H_u (resp., H_w) which has slope less than t. Note that H_u and H_w both contain an edge with slope $+\infty$ and an edge with slope $-\infty$, thus the edges e and f always exist. Let H'_u denote the list H_u with e replaced by t and every edge after e in H_u removed. Similarly, let H'_w denote the list H_w with every edge before f removed. Given the rank of t in H_u and H_w, respectively, we can easily construct H'_u and H'_w in constant time using $O(|H_u| + |H_w|)$ processors.

Given two sorted lists A and B (from the same universe), if, for any two adjacent items e and f in A, the rank of e in B differs from the rank of f in B by at most a constant c, then we say that A is a *c-cover* for B. In order for us to be able to perform each stage of our algorithm in constant time using a linear number of processors it must be the case that the list H_v at time t be a c-cover of H_v at time $t+1$. Atallah, Cole, and Goodrich [2] show that this "c-cover" property would surely be true if we were not deleting elements from H_u and H_w and then adding an edge to H_u (to create H'_u and H'_w). Thus, if we can show that the H_v definitions have a c-cover property in spite of the deletions and additions, then we will have established that each stage can be performed in $O(1)$ time using $O(n)$ processors. First, note that H_u is a 2-cover for H'_u, since in creating H'_u from H_w we deleted a contiguous block of elements from H_u and then added at most one additional edge. We also have that H_w is trivially a 1-cover for H'_w. Thus, by transitivity, H_v is a 2-cover for $H^*_v = H'_u \cup H'_w$. This, in turn, implies that H_v at time t will be a c-cover of H_v at time $t+1$. (The interested reader may wish to examine the details of the proofs in [2] to show that, in fact, H_v at time t is a 4-cover of H_v at time $t+1$.) We summarize with the following theorem:

Theorem 5.1: *Given a set S of n points in the plane one can construct the convex hull of S in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model.* ∎

6. Final Remarks and Open Problems

This paper presents parallel analogues to some famous notions from sequential computational geometry. Namely, that the all-nearest neighbor problem can be solved without constructing a Voronoi diagram, that problems for polygons can often-times be solved faster than point-set problems, and that convex hulls can be constructed by performing divide-and-conquer with edge lists sorted by slopes. Another interesting observation is that in developing an optimal parallel algorithm for the kernel problem we discovered some geometric relationships which result in a new optimal sequential algorithm (which is simpler than the previous best algorithm [15]). We leave two open problems:

1. Can the Voronoi diagram of n planar points be constructed in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model? The current best algorithm runs in $O(\log^2 n)$ time using $O(n)$ processors [1,2] (we already know how to solve the all-nearest neighbors problem in $O(\log n)$ time using $O(n)$ processors (in the EREW PRAM model)).

2. Can the convex hull of n planar points plane be constructed in $O(\log n)$ time using $O(n)$ processors in the EREW PRAM model? It is known that sorting can be done in $O(\log n)$ time using $O(n)$ processors in the EREW PRAM model [8]. The close relationship between the convex hull problem and sorting is well known in sequential computational geometry, but in the parallel setting there is currently a gap between their respective complexities.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," manuscript, 1987 (a preliminary version appeared in *Proc. 25th IEEE Symp. on Found. of Comp. Sci.*, 1985, 468–477).

[2] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *Proc. 28th IEEE Symp. on Found. of Comp. Sci.*, 1987, 151–160.

[3] M.J. Atallah and M.T. Goodrich, "Efficient Parallel Solutions to Some Geometric Problems," *J. of Par. and Dist. Comp.*, Vol. 3, 1986, 492–507.

[4] M.J. Atallah and M.T. Goodrich, "Parallel Algorithms for Some Functions of Two Convex Polygons," to appear *Algorithmica*.

[5] J. L. Bentley and M. I. Shamos, "Divide-And-Conquer in Multidimensional Space," *Proc. 8th ACM Symp. on Theory of Computing*, 1976, 220–230.

[6] G. Bilardi and A. Nicolau, "Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines," TR 86-769, Dept. of Comp. Sci., Cornell Univ., August 1986.

[7] A. Chow, "Parallel Algorithms for Geometric Problems," Ph.D. thesis, Comp. Sci. Dept., Univ. of Ill. at Urbana-Champaign, 1980.

[8] R. Cole, "Parallel Merge Sort," *Proc. 27th IEEE Symp. on Found. of Comp. Sci.*, 1986, 511–516.

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

[10] M.T. Goodrich, "Finding the Convex Hull of a Sorted Point Set in Parallel," *Info. Proc. Letters*, Vol. 26, December 1987, 173–179.

[11] L. Guibas, L. Ramshaw, and J. Stolfi, "A Kinetic Framework for Computational Geometry," *Proc. 24th IEEE Symp. on Found. of Comp. Sci.*, 1983, 100–111.

[12] C.P. Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," *Proc. 1985 IEEE Int. Conf. on Parallel Processing*, 180–185.

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

[14] D.T. Lee and F.P. Preparata, "The All Nearest-Neighbor Problem for Convex Polygons," *Info. Proc. Letters*, Vol. 7, No. 4, June 1978, 189–192.

[15] D.T. Lee and F.P. Preparata, "An Optimal Algorithm for Finding the Kernel of a Polygon," *J. ACM*, Vol. 26, No. 3, July 1979, 415–421.

[16] F.P. Preparata and D.E. Muller, "Finding the Intersection of n Half-spaces in Time $O(n \log n)$," *Theoretical Comp. Sci.*, Vol. 8, 1979, 45–55.

[17] M.I. Shamos, "Geometric Complexity," *Proc. 7th ACM Symp. on Theory of Computing, 1975*, 224–233.

[18] G.T. Toussaint, "Solving Geometric Problems with Rotating Calipers," *Proc. IEEE MELECON '83*, Athens, Greece, May 1983.

[19] H. Wagener, "Optimally Parallel Algorithms for Convex Hull Determination," unpublished manuscript, September 1985.

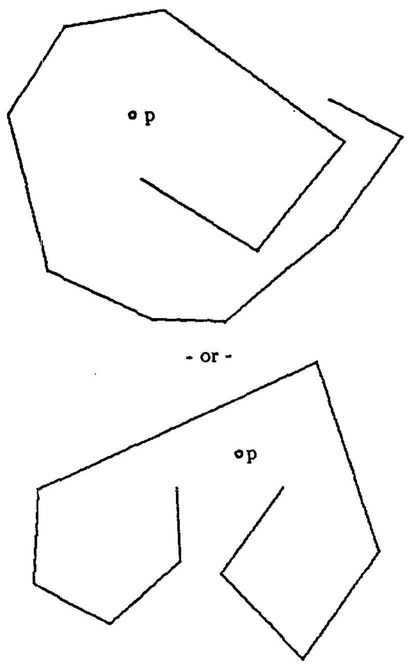

Figure 1: A spiral polygon has an empty kernel.

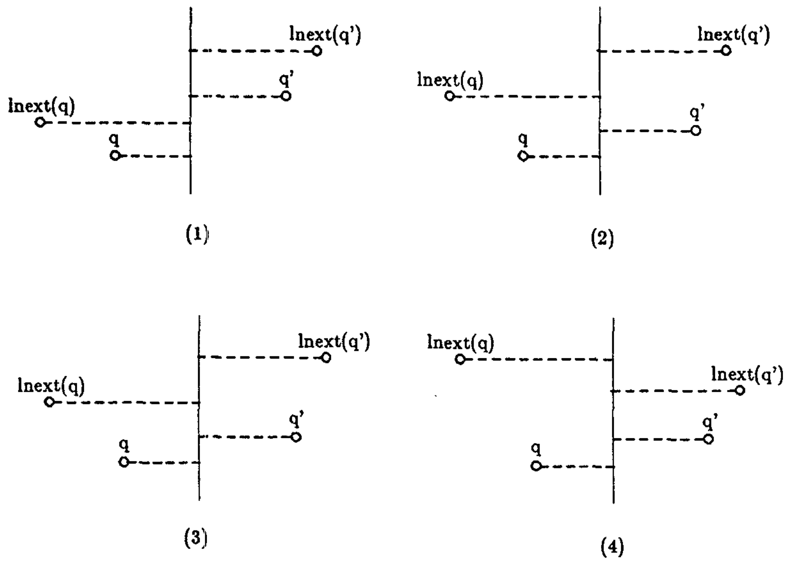

Figure 3: The different cases for updating the *lnext* label.

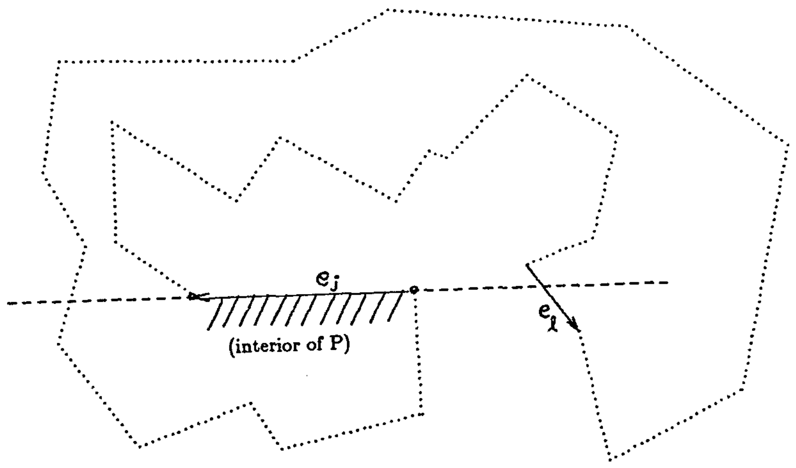

Figure 2: If e_i intersects $H(e_j)$ then P is a spiral polygon.

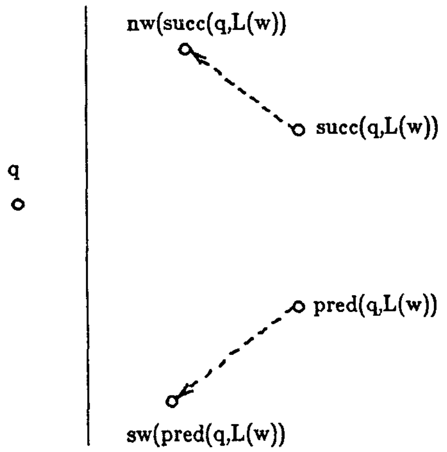

Figure 4: The only points in $Y(w)$ which could have $q \in Y(u)$ as their nearest neighbor.