Searching for empty convex polygons

David P. Dobkin, Herbert Edelsbrunner and Mark H. Overmars

RUU-CS-88-11
March 1988

Rijksuniversiteit Utrecht

Vakgroep Informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Searching for empty convex polygons

David P. Dobkin, Herbert Edelsbrunner and Mark H. Overmars

Technical Report RUU-CS-88-11
March 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht
the Netherlands

Searching for Empty Convex Polygons[1]

David P. Dobkin[2], Herbert Edelsbrunner[3] and Mark H. Overmars[4]

Abstract

A key problem in computational geometry is the identification of subsets of a point set having particular properties. We study this problem for the properties of convexity and emptiness. We show that finding empty triangles is related to the problem of determining pairs of vertices that see each other in a star-shaped polygon. A linear time algorithm for this problem which is of independent interest yields an optimal algorithm for finding all empty triangles. This result is then extended to an algorithm for finding empty convex r-gons ($r > 3$) and for determining a largest empty convex subset. Finally, extensions to higher dimensions are mentioned.

Keywords: Computational geometry, empty convex subsets, analysis of algorithms, combinatorial geometry.

[1]The first author is pleased to acknowledge support by the National Science Foundation under grant CCR-8700917. Research of the second author was supported by Amoco Foundation Faculty Development Grant CS 1-6-44862 and by the National Science Foundation under grant CCR-8714565.

[2]Department of Computer Science, Princeton University, Princeton, New Jersey 08544, USA.

[3]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, USA.

[4]Department of Computer Science, University of Utrecht, P. O. Box 80012 NL-3508 TA Utrecht, the Netherlands.

kernel. For this input there is a simpler algorithm with the same asymptotic running time. This algorithm is presented in the next section.

Step 3 will be split into two steps. In the first step, to be described in section 4 we will determine the longest convex chain in the visibility graph. In fact, we determine for each edge of the visibility graph the longest convex chain that starts there. In the second step we use this information to determine all the chains of length $r-2$. Both parts will run in time the size of the visibility graph.

3 The visibility graph of a star-shaped polygon.

We are now given a star-shaped polygon P of N vertices with one vertex p that lies in the kernel. We are interested in obtaining the visibility graph inside P, denoted as VG. For a pair of vertices of P we say that they are *visible within* P if the line segment joining them lies entirely in P (including its boundary). Note that, because of our assumption that the points lie in general position, the line segment will either intersect the boundary of P in its two endpoints or is an edge of P. The *visibility graph* inside P consists of all pairs of vertices of P that are visible within P.

The vertices of P are ordered counter-clockwise around p. We number them in this way $p_1, \ldots, p_{N-1}$. Because of the requirements we have later we will compute the visibility graph as a directed graph in which edges run from lower indexed to higher indexed vertices. (Moreover, we will not include the visibility edges involving p.) The edge (in the visibility graph) between p_i and p_j $(i < j)$ will be denoted by $\overrightarrow{ij}$.

We will construct the visibility graph VG during one counter-clockwise scan around the polygon. When we visit p_i we construct all incoming edges of p_i. With each vertex p_i we maintain a queue Q_i that stores the starting points of some of the incoming edges of p_i in counter-clockwise order. It contains those points p_j such that $\overrightarrow{ji}$ is an edge of the visibility graph and we have not yet reached another point p_k with $k > i$ such that $\overrightarrow{jk}$ is an edge of the visibility graph. Hence, Q_i is a kind of waiting list. It contains those points that could be seen by p_i but could not be seen since, because p_i blocks their view. The required operations that can be performed in constant time are the following:

1. ADD($\overrightarrow{ij}$): it creates an edge from i to j. This edge will be stored at both p_i and p_j for later use.
2. TURN($\overrightarrow{ij}$,$\overrightarrow{jk}$): it returns *left* or *right* depending on whether p_k lies to the left or to the right of the directed line passing through p_i and p_j in this order. (Note that it cannot lie on the line.)
3. FRONT(Q): it returns the index of the first point in queue Q.
4. DEQUEUE(Q): it removes the first point from queue Q.
5. ENQUEUE(k,Q): it adds the point p_k to queue Q.

The algorithm now looks as follows:

procedure VISIBILITY;

for $i := 1$ **to** $N - 1$ **do** $Q_i := \emptyset$ **end**;
for $i := 1$ **to** $N - 2$ **do** PROCEED(i,$i+1$) **end**.

procedure PROCEED(i,j);
while $Q_i \neq \emptyset$ **and** TURN($\overline{\text{FRONT}(Q_i)i}$,$\overline{ij}$)=*left* **do**
PROCEED(FRONT(Q_i),j);
DEQUEUE(Q_i);
end;
ADD($\overline{ij}$);
ENQUEUE(i,Q_j).

PROCEED adds an edge from point p_i to p_j. It also checks whether any of the points in the waiting queue of p_i are visible from p_j and, if so, recursively calls PROCEED. Because the points in the queue are sorted counter-clockwise only a first portion of the queue needs to be checked.

Note that the new edge is added after all the recursive calls. This guarantees that the points in the queues are indeed sorted counter-clockwise. Also, for each node we will collect the incoming and outgoing edges sorted counter-clockwise.

The correctness of the method follows from the following lemma:

Lemma 3.1 *Let $j > i$. $\overline{ij} \in VG$ if and only if $j = i+1$ or there is a vertex p_k, $i < k < j$, such that $\overline{ik} \in VG$, $\overline{kj} \in VG$, and TURN($\overline{ik}, \overline{kj}$) = left.*

Proof. First note that because p lies inside the kernel, $\overline{ij} \in VG$ if and only if triangle pp_ip_j empty. Now the proof goes as follows:

if: If $j \neq i+1$ take the point p_k between p_i and p_j that lies nearest to the line $\overline{ij}$. As the triangle pp_ip_j is empty, TURN($\overline{ik}, \overline{kj}$) = left. Moreover, obviously $\overline{ik} \in VG$ and $\overline{kj} \in VG$.

only if: If $j = i+1$ $\overline{ij}$ clearly is in VG. If $j > i+1$ both the triangles pp_ip_k and pp_kp_j are empty. Moreover, p_k lies beyond $\overline{ij}$ since p_i, p_k, p_j form a left turn. Hence, the triangle pp_ip_j must be empty and $\overline{ij} \in VG$. □

Lemma 3.2 *Finding the visibility graph takes time $O(|VG|)$.*

Proof. This follows immediately from the fact that with every call of PROCEED an edge will be added to the visibility graph. □

4 Finding the longest convex chain.

Given the visibility graph as a directed graph in which edges run from lower indexed vertices to higher indexed vertices (as produced by the algorithm described above), we will now determine a

longest convex chain. This is equivalent to finding a largest empty convex subset with a fixed leftmost vertex. In fact, we will determine for each edge e of the visibility graph L_e being the length of the longest convex chain starting with e.

To this end we will treat the vertices clockwise, starting at the highest indexed vertex. We will take care that after treating some vertex p_i all incoming edges of p_i have their L field set to the right value.

The method works as follows: Assume we are at some vertex p. Let the incoming edges of p be $i_1, ..., i_{imax}$ and the outgoing edges $o_1, ..., o_{omax}$ both ordered counter-clockwise by angle. Note that the algorithm for computing the visibility graph inside P gives us the edges in this order. For all outgoing edges we know the length of the longest convex chain that starts there.

We will treat the incoming edges in the reversed order, starting at i_{imax}. For this first incoming edge we look at all outgoing edges that form a convex angle with it. Let these edges be $o_l, ..., o_{omax}$. If they don't exist, we set $L_{i_{imax}}$ to 1. Otherwise, let m be the maximal value of the L fields of them. Then $L_{i_{imax}} = m + 1$. Clearly, all edges that form a convex angle with i_j form a convex angle with i_{j-1}. But we don't have to check these edges again. We already know that m is the maximal length among them. Hence, for the next edge we know that the length of the chain is either $m+1$ or there is an edge with index smaller than l with larger L field. Hence, starting at $l-1$ we look at preceding edges that form a convex angle with the incoming edge. l will become the new minimal index, m the new maximal L field (if any). In this way we continue.

The following procedures state the algorithm precisely:

```
procedure MAXCHAIN;
    for i := N − 1 downto 1 do TREAT(p_i) end.

procedure TREAT(p);
    Let i_1, ..., i_imax be the incoming edges of p, and let
    o_1, ..., o_omax be the outgoing edges of p, both ordered counter-clockwise.
    l := omax; m := 0;
    for j := imax downto 1 do
        L_{i_j} := m + 1;
        while l > 0 and TURN(i_j,o_l)=left do
            if L_{o_l} > m then
                m := L_{o_l};
                L_{i_j} := m + 1
            end;
            l := l − 1
        end
    end.
```

The correctness of the method easily follows from the above discussion. (Note that in the algorithm the function of l is slightly different than described. Rather that being the minimal index that