COMPUTING VISIBILITY GRAPH

SHORTEST PATH FOR A POINT ROBOT

Let's see what we can say about the shape of a shortest path. Consider some path from Pstart to Pgoal. Think of this path as an elastic rubber band, whose endpoints we fix at the start and goal position and which we force to take the shape of the path.

At the moment we release the rubber band, it will try to contract and become as short as possible, but it will be stopped by the obstacles. The new path will follow parts of the obstacle boundaries and straight line segments through open space.

This is showed in the *figure 2.1*.

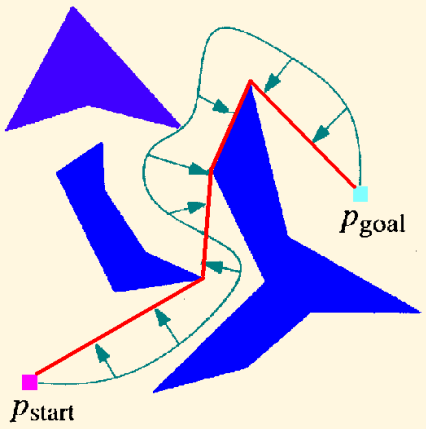


Fig. 2.1

The next lemma formulates this observation more precisely. It uses the notion of an inner vertex of a polygonal path, which is a vertex that is not the begin- or endpoint of the path.

LEMMA 1
Any shortest path between Pstart. and Pgoal among a set S of disjoint polygonal obstacles is a polygonal path whose inner vertices are vertices of S.

PROOF
Suppose for a contradiction that a shortest path t is not polygonal. Since the obstacles are polygonal, this means there is a point p on t that lies in the interior of the free space with the property that no line segment containing p is contained in t.

Since p is in the interior of the free space, there is a disc of positive radius centred at p that is completely contained in the free space. But then the part of t inside the disc, which is not a straight line segment, can be shortened by replacing it with the segment connecting the point where it enters the disc to the point where it leaves the disc.

This contradicts the optimality of t, since any shortest path must be locally shortest, that is, any subpath connecting points q and r on the path must be the shortest path from q to r.

Now consider a vertex v of t. It cannot he in the interior of the free space: then there would be a disc centred at p that is completely in the free space, and we could replace the subpath of t t inside the disc- which turns at v-by a straight line segment which is shorter.

Similarly, v cannot he in the relative interior of an obstacle edge: then there would be a disc centred at v such that half of the disc is contained in the free space, which again implies that we can replace the subpath inside the disc with a straight line segment. The only possibility left is that v is an obstacle vertex. Esto contradice la optimalidad de t, puesto que cualquier camino mínimo lo debe ser localmente, es decir, que cualquier subcamino conteniendo a dos puntos q y r, debe ser el camino mínimo para ir de q a r.

With this characterisation of the shortest path, we can construct a road map that allows us to find the shortest path. This road map is the visibility graph of S, which we denote by Gvis (S). Its nodes are the vertices of S, and there is an arc between vertices v and w if they can see each other, that is, if the segment vw does not intersect the interior of any obstacle in S.

Two vertices that can see each other are also called (mutually) visible, and the segment connecting them is called a visibility edge. Note that endpoints of the same obstacle edge always see each other. Hence, the obstacle edges form a subset of the arcs of Gvis(S).

By Lemma 1 the segments on a shortest path are visibility edges, except for the first and last segment.

To make them visibility edges as well, we add the start and goal position as vertices to S, that is, we consider the visibility graph of the set S* := S U {Pstart, Pgoal }.

By definition, the arcs of Gvis (S*) are between vertices-which now include Pstart and pgoal-that can see each other.

We get the following corollary:

COROLLARY 2
The shortest path between Pstart and Pgoal among a set S of disjoint polygonal obstacles consists of arcs of the visibility graph Gvis (S*), where S* := SU {Pstart,Pgoal}.

This Corollary implies that we can compute a shortest path from Pstart to Pgoal as follows:

Algorithm SHORTEST PATH(S,pstart,pgoal)

Input*: A set S of disjoint polygonal obstacles, and two points Pstart and Pgoal in the free space*
Output*:The shortest collision-free path connecting Pstart and Pgoal*

1*. Gvis = VISIBILITYGRAPH(S U {pstart,pgoal})*
2*. Assign each arc (v, w) in Gvis a weight, which is the Euclidean length of the segment vw.*
3*. Use Dijkstra's algorithm to compute a shortest path between Pstart and Pgoal in Gvis.*

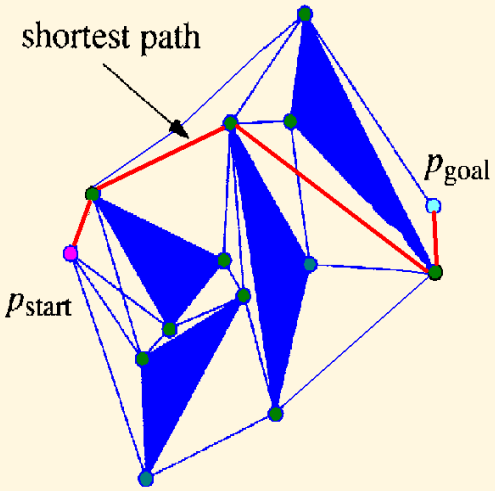


Fig. 2.2

Complexity

In the next section we show how to compute the visibility graph in 0(n2 log n) time, where n is the total number of obstacle edges. The number of arcs of Gvis is of course bounded by (n+2 2).

Hence, line 2 of the algorithm takes 0(n2) time.

Dijkstra's algorithm computes the shortest path between two nodes in graph with k arcs, each having a non-negative weight, in 0(n log n + k) time. Since k = 0(n2), we conclude that the total running time of SHORTESTPATH is 0(n2 log n), leading to the following theorem.

THEOREM

A shortest path between two points among a set of polygonal obstacles with n edges in total can be computed in O(n2 log n) time.

COMPUTING VISIBILITY GRAPH

Let S be a set of disjoint polygonal obstacles in the plane with n edges in total. To compute the visibility graph of S, we have to find the pairs of vertices that can see each other.

This means that for every pair we have to test whether the line segment connecting them intersects any obstacle. Such a test would cost 0(n) time when done naively, leading to an 0(n3) running time. We will see shortly that the test can be done more efficiently if we don't consider the pairs in arbitrary order, but concentrate on one vertex at a time and identify ah vertices visible from it.

As in the following algorithm:

Algorithm VISIBILITY GRAPH(S)

Input*: A set S of disjoint polygonal obstacles.*

***Output**: The visibility graph Gvis (S).*

***1**. Initialize a graph G= (V, E) where V is the set of all vertices of the polygons in S and E=Ø.*
***2**. For all vertices v of V*
***3**. do W VISIBLEVERTICES(V,S)*
***4**. For every vertex wÎW, add the arc (v,w) to E.*
***5**. Return G.*

The procedure VISIBLEVERTICES VISIBLEVERTICES has as input a set S of polygonal obstacles and a point p in the plane; in our case p is a vertex of S, but that is not required.
It should return ah obstacle vertices visible from p.

This procedure is explained in the following section.

VERTICES VISIBILITY

If we just want to test whether one specific vertex w is visible from p, there is not much we can do: we have to check the segment pw against all obstacles.

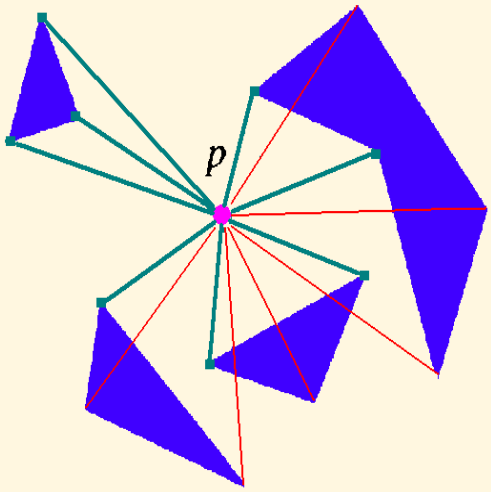


Fig. 2.3

The **Rough Algorithm** determines the visibility of the vertices checking all intersections between each pair of points against all the edges of hte obstacles.

The algorithm is as follows:

Algorithm VISIBLEVERTICES(p,S)

***Input**: A set S of polygonal obstacles and a point p that does not lie in the interior of any obstacle.*
***Salida**: The set of all obstacle vertices visible from p.*

***1**.Let V= {v1, ...,vn} the set of the vertices and E={e1,...,en} set of the edges*
***2**.W = 0*
***3**.For i 1 to n*
***4**. do if pvi does not intersect ej for all j=1 to n*
***5**. Add wi to W*
***6**.Return W.*

This algorithm is o(n3) computing time, so it is not efficient enough.

In order to improve it, we have to reduce the number of vertices to check, we might be able to use the information we get when we test one vertex to speed up the test for other vertices.

In this way, we conclude the **Optimal Algorithm** that has been implemented as second option.

Let`s consider the set of all segments pw .

What would be a good order to treat them, so that we can use the information from one vertex when we treat the next one? The logical choice is the cyclic order around p. So what we will do is treat the vertices in cyclic order, meanwhile maintaining information that will help us to decide on the visibility of the next vertex to be treated.

A vertex w is visible from p if the segment pw does not intersect the interior of any obstacle. Consider the half-line p starting at p and passing through w. If w is not visible, then p must hit an obstacle edge before it reaches w.

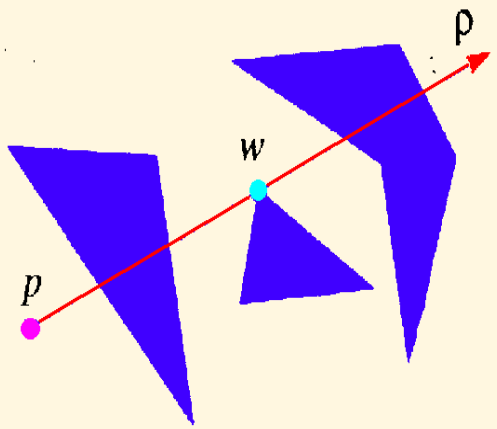


Fig. 2.4

To check this we perform a binary search with the vertex w on the obstacle edges intersected by p. This way we can find out whether w lies behind any of these edges, as seen from p. (If p itself is also an obstacle vertex, then there is another case where w is not visible, namely when p and w are vertices of the same obstacle and pw is contained in the interior of that obstacle. This case can be checked by looking at the edges incident to w, to see whether p is in the interior of the obstacle before it reaches w.

For the moment we ignore degenerate cases, where one of the incident edges of w is contained in pw)

While treating the vertices in cyclic order around p we therefore maintain the obstacle edges intersected by p in a balanced search tree t. (As we will see later, edges that are contained in p need not be stored in t.)

The leaves of t store the intersected edges in order: the leftmost leaf stores the first segment intersected by r, the next leaf stores the segment thwt is intersected next, and so on. The interior nodes, which guide the search in t, also store edges.

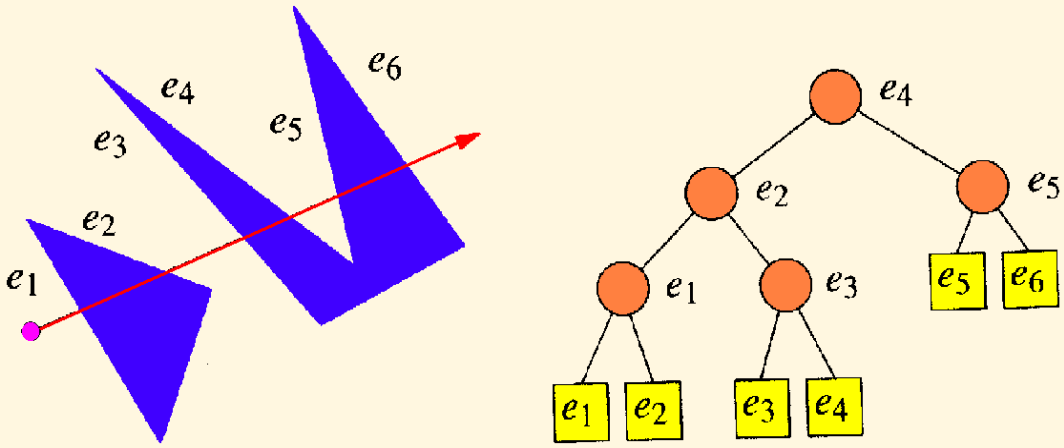


Fig. 2.5

More precisely, an interior node v stores the rightmost edge in its left subtree, so that all edges in its right subtree are greater (with respect to the order along r) than this segment ev, and all segments in its left subtree are smaller than or equal to ev (with respect to the order along r).

Treating the vertices in cyclic order effectively means that we rotate the half-line r around p. So our approach is similar to the plane sweep paradigm we used at various other places; the difference is that instead of using a horizontal line moving downward to sweep the plane, we use a rotating half-line.

The status of our rotational plane sweep is the ordered sequence of obstacle edges intersected by r. It is maintained in t. The events in the sweep are the vertices of S. To deal with a vertex w we have to decide whether w is visible from p by searching in the status structure t, and we have to update t by inserting and/or deleting the obstacle edges incident to w.

Algorithm **VISIBLEVERTICES** summarizes our rotational plane sweep. The sweep is started with the half-line r pointing into the positive x-direction and proceeds in clockwise direction.

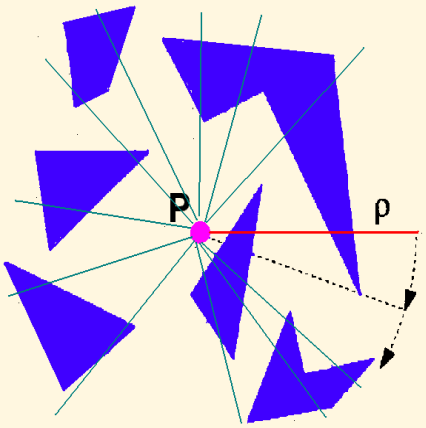


Fig. 2.6

So the algorithm first sorts the vertices by the clockwise angle that the segment from p to each vertex makes with the positive x-axis. What do we do if this angle is equal for two or more vertices? To be able to decide on the visibility of a vertex w, we need to know whether pw intersects the interior of any obstacle. Hence, the obvious choice is to treat any vertices that may he in the interior of pw before we treat w. In other words, vertices for which the angle is the same are treated in order of increasing distance to p.

The algorithm now becomes as follows: Finalmente, para decidir si un vértice w es visible, deberemos comprobar si pw intersecta el interior de algún obstáculo.

Algorithm VISIBLEVERTICES(p,S)

***Input**: A set S of polygonal obstacles and a point p that does not he in the interior of any obstacle.*
***Output**: The set of all obstacle vertices visible from p.*

***1**.Sort the obstacle vertices according to the clockwise angle that the half-line from p to each vertex makes with the positive x-axis. In case of ties, vertices closer to p should come before vertices farther from p. Let w1,... , wn be the sorted list.*
***2**. Let r be the half-line parallel to the positive x-axis starting at p. Find the obstacle edges that are properly intersected by p, and store them in a balanced search tree t in the order in which they are intersected by r.*
***3**.W = 0*
***4**.For i 1 to n*
***5**. do if VISIBLE(wi) then Add wi to W.*
***6**. Insert into t the obstacle edges incident to wi that he on the clockwise side of the half-line from p to wi.*
***7**. Delete from t the obstacle edges incident to wi that lie on the counterclockwise side of the half-line from p to wi*
***8**.Return W.*

The subroutine VISIBLE must decide whether a vertex wi is visible.

Normally, this only involves searching in t to see if the edge closest to p, which is stored in the leftmost leaf, intersects pwi. But we have to be careful when pwi contains other vertices. Is wi visible or not in such a case? That depends. See Figure 2.7 for some of the cases that can occur.pwi may or may not intersect the interior of the obstacles incident to these vertices. It seems that we have to inspect all edges with a vertex on pwi to decide if wi is visible.

Fortunately we have already inspected them while treating the preceding vertices that lie on pwi. We can therefore decide on the visibility of wi as follows: - If wi-1 is not visible then wi is not visible either.

- If wi-1 is visible then there are two ways in which wi can be invisible. Either the whole segment wi-1 wi lies in an obstacle of which both wi-1 and wi are vertices, or the segment wi-1 w1 is intersected by an edge in t. (Because in the latter case this edge lies between wi-1 and wi is must properly intersect wi-1 wi.) This test is correct because pwi = pwi-1 U wi-1 wi. (If i = 1, then there is no vertex in between p and wi, so we only have to look at the segment pwi.)

The figure 2.7. shows four examples, in the two on the left wi is visible and the two on the right is not, being wi-1 visible in both cases.

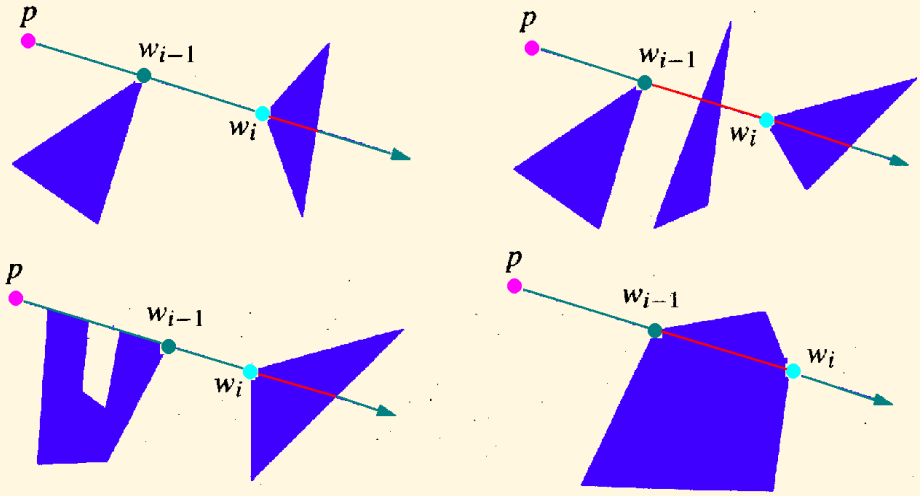


Fig. 2.7

We get the following subroutine:

Algorithm VISIBLE(wi)

***1**. if pwi intersects the interior of the obstacle of which wi is a vertex, locally at wi*
***2**. then Return TRUE*

3. *else if i=1 o wi-1 is not on the segment pwi*
4. *then Search in t for the edge e in the leftmost leaf.*
5. *if e exists and pwi intersects e*
6. *then Return FALSE*
7. *else Return TRUE*
8. *else if wi-1 is not visible*
9. *then Return FALSE*
10. *else Search in t for an edge e that intersects wi-1wi*
11. *if e exists*
12. *then Return FALSE*
13. *else Return TRUE*

The running time of VISIBLEVERTICES is O(nlog n):

The time we spent before line 4 is dominated by the time to sort the vertices in cyclic order around p, which is O(n log n).

Each execution of the loop involves a constant number of operations on the balanced search tree t, which take O(log n) time, plus a constant number of geometric tests that take constant time.

Hence, one execution takes O(log n) time, leading to an overall running time of O(n log n).

THEOREM

The visibility graph of a set S of disjoint polygonal obstacles with n edges in total can be computed in O(n2 log n) time.

OPTIMIZATION FOR CONVEX OBSTACLES

When all obstacles are convex polygons we can improve the shortest path algorithm by only considering common tangents rather than all visibihity edges.

that the only visibihity edges that are required in the shortest path algorithm are the common tangents of the polygons, as shows the following figure:

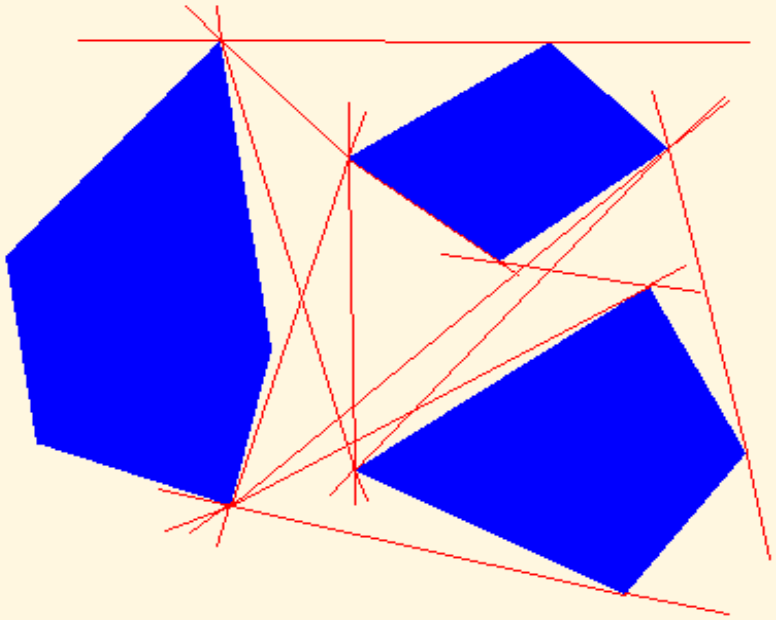


Fig. 2.8

All the common tangents are edges of the visibility graph and are included in the shortest path.

In the figure 2.9. (a) we have the computed shortest path taking into account the common tangents (in red) and (b) the calculated path without them.

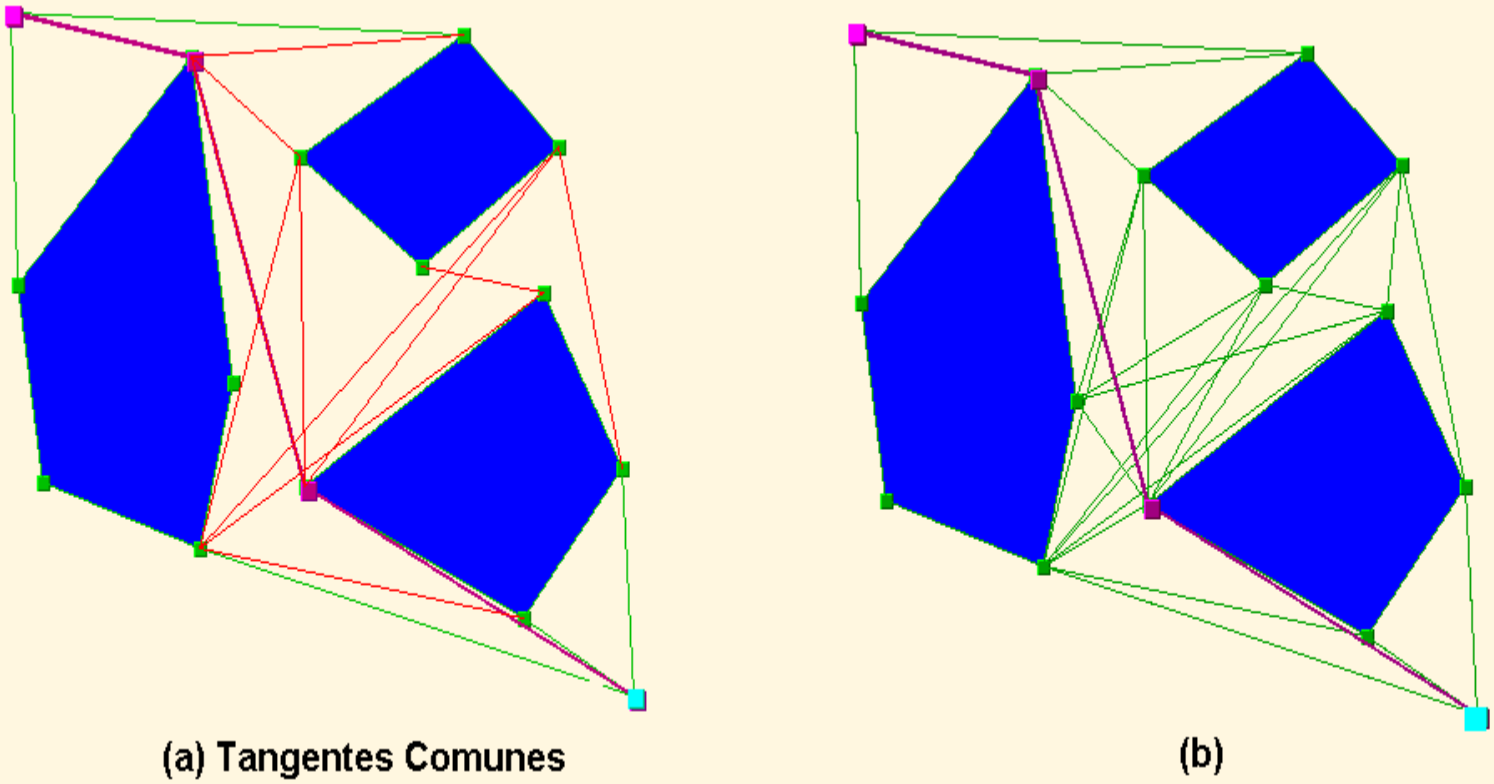


Fig. 2.9

In order to Calculate the common tangents between the polygons, several algorithms can be implemented.

The easiest one would consist in checking whether each edge for all the vertices of the polygons, is a common tangent. Thus we

obtain the algorithm:

Algorithm COMMON_TANGENTS (S):

***Input**: Let S the set of n convex polygons.*
***Output**: The list of common tangents to all polygons.*

***1**. CT =0*
***2**. For i=1 to n*
***3**. For j=1 to n*
***4**. tangents= TANGENTS(poli,polj) for i<>j*
***5**. For t =1 to k*
***6**. If tangents[t] does not intersect polz for all z<>i y z<>j*
***7**. then Add tangents[t] to CT*
***8**. Return CT*

The TANGENTS algorithm between two polygons is as follows:

Algorithm TANGENTS (pol1,pol2):

***Input**: A pair of pol1 and pol2, of n y m number of vertices respectly.*
***Output**: The list of commont tangents between the given polygons.*

***1**. LT =0*
***2**. For i=1..n*
***3**. For j=1..m*
***4**. If viwj is tangent to pol1 and to pol2*
***5**. Add viwj to LT*
***6**. Return LT*

Both algorithm are not efficient.

To obtain the commont tangents to both polygons the calipers can be used, reducing the running time.

BRIDGES AND CALIPERS ROTATION

Common tangents are simply lines that are tangent to both polygons at the same time, and so that the both polygons lie to one side of that line. In other words, a common tangent is a line of support for both polygons. An example is illustrated below:

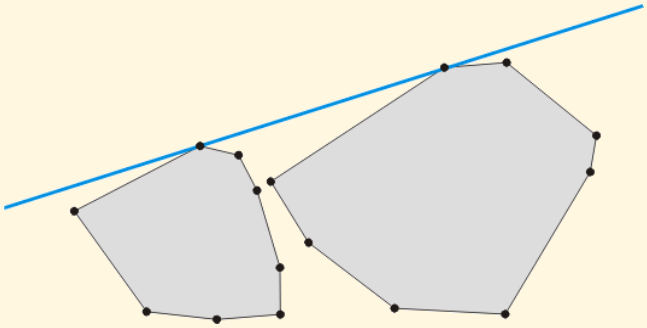
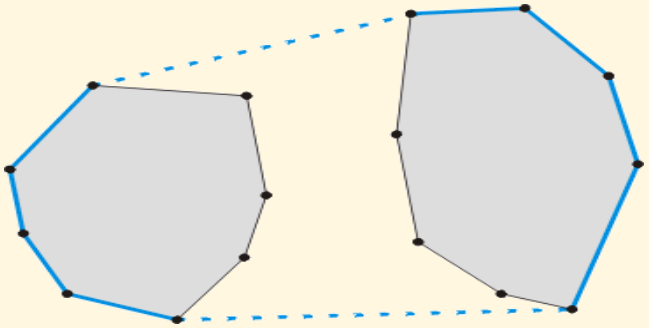


Fig. 2.10

In fact, common tangents are determined by the same pairs of points between the polygons that determine the bridges. Hence, given two disjoint polygons, there exist two common tangents between the polygons, and when the polygons intersect, there may be as many as there are vertices.

The same algorithm computing **the bridges** between two convex polygons (i.e. the merge algorithm) can therefore be used to determine common tangents.

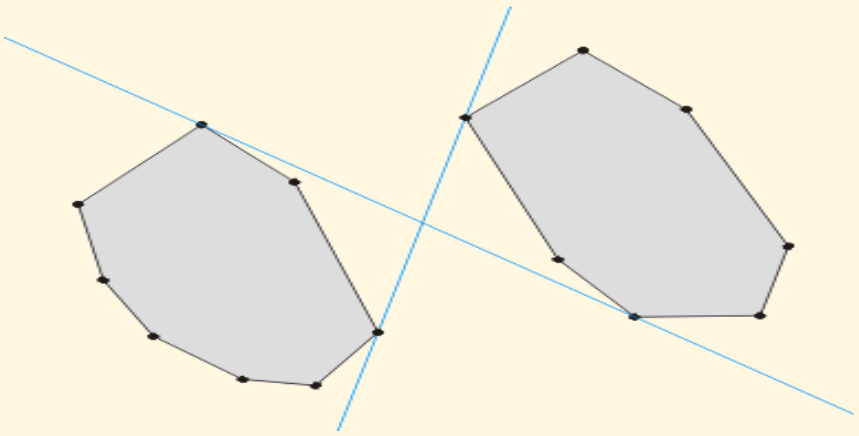


Another version of lines tangent to both polygons are **critical support lines**. In that case the polygons lie on opposite sides of the line.

Before explaining the algorithm, some concept should be reviewed:

LINES OF SUPPORT

Given a convex polygons P, a line of support l is a line intersecting P and such that the interior of P lies to one side of l. This concept is comparable to that of a tangent line.



ANTIPODAL PAIRS

If two points p and q (belonging to P) admit parallel lines of support, then they form an anti-podal pair. Two distinct parallel lines of support always determine at least one anti-podal pair.

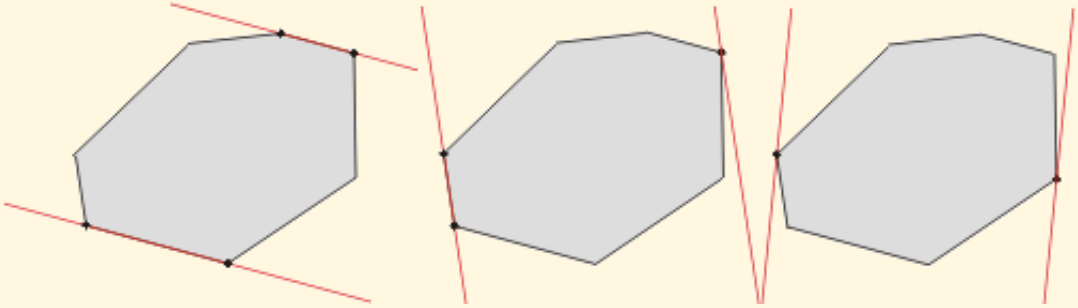


Fig. 2.11

COPODAL PAIRS BETWEEN TWO CONVEX POLYGONS

Given two polygons P and Q, a pair of points (p, q) (belonging to P and Q respectively) form an co-podal pair between P and Q if the polygons admit (directed) parallel lines of support in the same direction at p and q.
Two such lines of support always determine at least one co-podal pair.

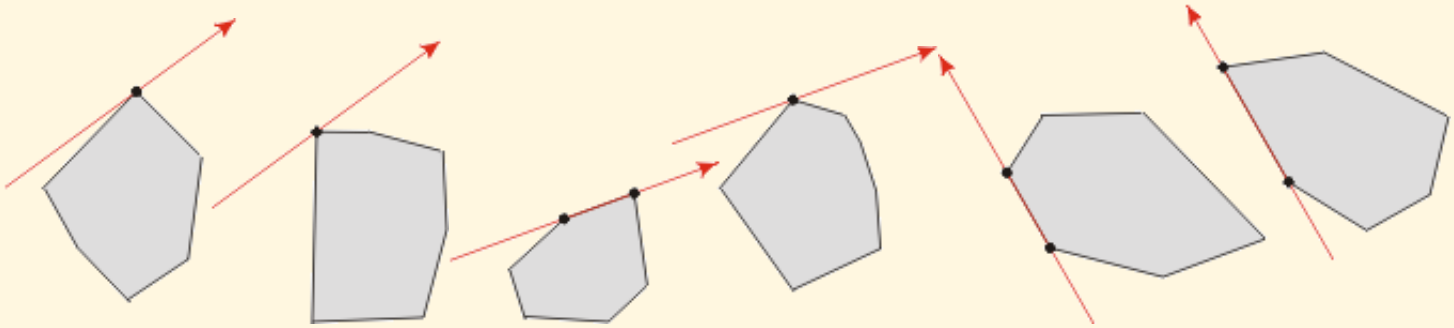


Fig. 2.12

ANTIPODAL PAIRS BETWEEN TWO CONVEX POLYGONS

Given two polygons P and Q, a pair of points (p, q) (belonging to P and Q respectively) form an anti-podal pair between P and Q if the polygons admit (directed) parallel lines of support in different directions at p and q.
Two such lines of support always determine at least one anti-podal pair.

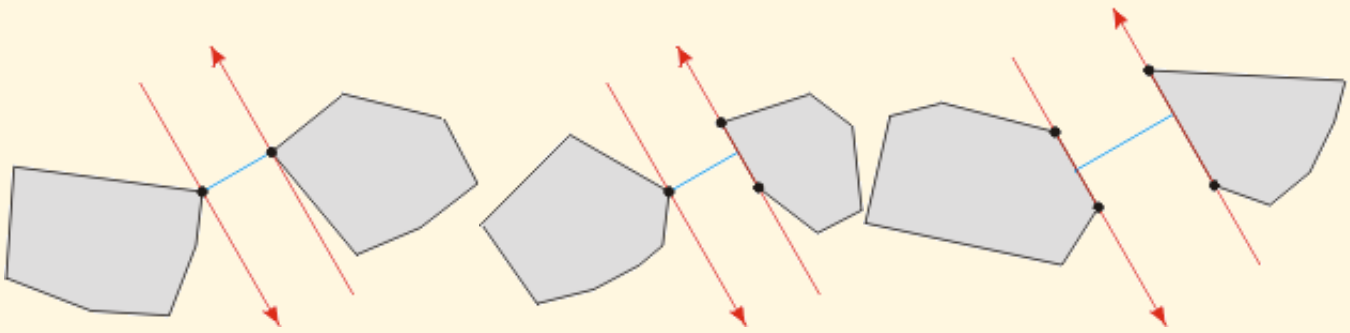


Fig. 2.13

Let us see the next results:

RESULT 1

Given convex polygons P={p(1), ... , p(m)} and Q={q(1), ... , q(n)}, a pair of points (p(i), q(j)) form a bridge between P and Q if, and only if:
Sean dos polígonos convexos P ={p(1),…,p(m)} y Q={a(1), …, q(n)} un par de puntos (p(i),q(j)) forman un puente entre P y Q si y

solo si:
1. (p(i),q(j)) form a co-podal pair
2. p(i-1),p(i+1),q(j-1),q(j+1) all lie to the same side of the line joining (p(i), q(j)).

RESULT 2

Given convex polygons P={p(1), ... , p(m)} and Q={q(1), ... , q(n)}, a pair of points (p(i), q(j)) form a bridge between P and Q if, and only if:
1. (p(i),q(j)) form a antipodal pair
2. p(i-1),p(i+1)lie to one side while q(i-1),q(i+1) lie to the opposite side of the line joining (p(i),q(j)).

The algorithm that calculates the esxtern common tangents to the polygons is as follows:

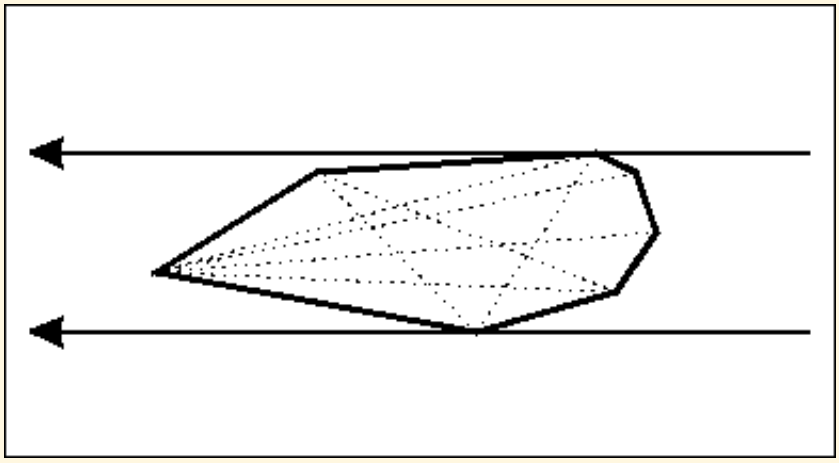
Algorithm ***BRIDGES***

Assuming the polygons are given in standard form and the vertices is clockwise order:

1. *Compute the vertices with maximum ycoordinate for both P and Q. If more than one exist, take the ones with greater x coordinates.*
2. *Construct horizontal lines of support at these points, directed such that the polygons lie to their right (thus they point to the positive end of the x axis).*
3. *Rotate both lines of support clockwise until one coincides with an edge. A new co-podal pair (p(i), q(j)) is determined. In the case of parallel edges, three co-podal pairs are determined.*
4. *For all valid co-podal pairs (p(i), q(j)): check if p(i-1), p(i+1), q(j-1), q(j+1), all lie to the same side of the line joining (p(i), q(j)). If so, then the co-podal pair is a bridge, and is labelled as such.*
5. *Repeat steps 3 and 4 until the lines of support reach their original position.*
6. *All possible bridge pairs have thus been determined.*

The running time is determined by the steps 1,5 y 6 each one of O(n) (n number of vertices). So the algorithm complexity is lineal O(n).

To determine the internal common tangents we use the algorithm of **Calipers Rotation**:



Algorithm ***CALIPERS ROTATION***

Assuming the polygons are given in standard form and clockwise order:

1. *Compute the vertex with minimum y coordinate for P (call it yminP) and the vertex with maximum y coordinate for Q (call it ymaxQ).*
2. *Construct two lines of support LP and LQ for the polygons at yminP and ymaxQ such that the polygons lie to the right of their respective lines of support. Then LP and LQ have opposite direction, and yminP and ymaxQ form an anti-podal pair between the polygons.*
3. *Let p(i)= yminP, and let q(j)= ymaxQ. (p(i), q(j)) form an anti-podal pair between the polygons. Check if p(i-1),p(i+1) lie on one side of the line (p(i), q(j)) and if q(j-1),q(j+1) lie on the other. If so, then (p(i), q(j)) determine a CS line.*
4. *Rotate the lines clockwise until one of them coincides with an edge of its polygon.*
5. *A new anti-podal pair is determined. If both lines of support coincide with edges, then a total of three anti-podal pairs (combinations of the previous vertices and the new vertices) need to be considered. For all new anti-podal pairs, perform the above checks.*
6. *Repeat steps 4 and 5, until the new pair considered is (yminP,ymaxQ).*
7. *Output the CS lines.*

The running time is lineal O(n).

NOTES AND COMMENTS

To summarize the next comments are remarked:

- The number of segments on the shortest path is bounded by 0(n)

- Algorithm VISIBILITYGRAPH calls algorithm VISIBLEVERTICES with each obstacle vertex. VISIBLEVERTICES sorts all vertices around its input point. This means that n cyclic sortings are done, one around each obstacle vertex. In the previous section we simply did every sort in 0(n log n) time, leading to 0(n2 log n) time for ah sortings. That this can be improved to 0(n2) time using dualization.

- The algorithm for finding the shortest path can be extended to objects other than polygons. Let S be a set of n disjoint disc obstacles (not necessarily of equal radius):

1. It can be proved the shortest path between two points consists of parts of boundaries of the disks and common tangents of disks.
2. Accordingly, the visibility concept seen, it can be used to find the shortest path between two points among the disks in S.

NOTAS Y COMENTARIOS

Como resumen hacer notar las siguientes afirmaciones:

- El número de segmentos que tiene el camino mínimo está limitado por O(n)

- El algoritmo del cálculo del Grafo de Visibilidad llama al algoritmo de Vértices Visibles para cada uno de los vértices de los obstáculos. Éste a su vez, ordena todos los vértices en función del vértice de entrada. Esto dignifica que se realizan n ordenaciones cíclicas, una por cada vértice. Como se ha visto en la sección anterior, se puede simplificar la ordenación a O(nlogn), teniendo O(n2log n) para todos los vértices.

Se puede mejorar todavía hasta O(n2) usando el concepto de Dualización.

- El algoritmo para encontrar el camino mínimo se puede extender a otro tipos de objetos diferente de polígonos. Por ejemplo sea S un conjunto de discos no necesariamente del mismo radio:

1. Se puede probar que el camino mínimo entre dos puntos consiste en parte de los límites de los discos y de tangentes comunes a ellos.
2. Según esto, se podría aplicar el mismo concepto de visibilidad y calcular así el camino mínimo entre dos puntos a través de los discos de S.