18. Pseudotriangulations

Lecture on Monday 8th December, 2008 by Michael Hoffmann <hoffmann@inf.ethz.ch>

We have seen that arrangements and visibility graphs are useful and powerful models for motion planning. However, these structures can be rather large and thus expensive to built and to work with. Let us therefore consider a simplified problem: for a robot which is positioned in some environment of obstacles, which and where is the next obstacle that it would hit, if it continues to move linearly in some direction?

This type of problem is known as a *ray shooting* query because we imagine to shoot a ray starting from the current position in a certain direction and want to know what is the first object hit by this ray. If the robot is modeled as a point and the environment as a simple polygon, we arrive at the following problem.

Problem 11 (Ray-shooting in a simple polygon.) *Given a simple polygon* $P = (p_1, \ldots, p_n)$*, a point* $q \in \mathbb{R}^2$*, and a ray* r *emanating from* q*, which is the first edge of* P *hit by* r*?*

In the end, we would like to have a data structure to preprocess a given polygon such that a ray shooting query can be answered efficiently for any query ray starting somewhere inside P.

As a warmup, let us look at the case that P is a convex polygon. Here the problem is easy: Supposing we are given the vertices of P in an array-like structure, the boundary segment hit by any given ray can be found in $O(\log n)$ time using binary search. This works because in a convex polygon P every point inside sees all vertices of P in the same order, namely their order along the boundary of P. Obviously, this is not true for non-convex polygons.

Let us consider a different, comparatively complicated method to solve the same problem. Triangulate P in a balanced way, that is, such that the tree dual to the triangulation is balanced (see Figure 18.1). The crucial property of a balanced triangulation

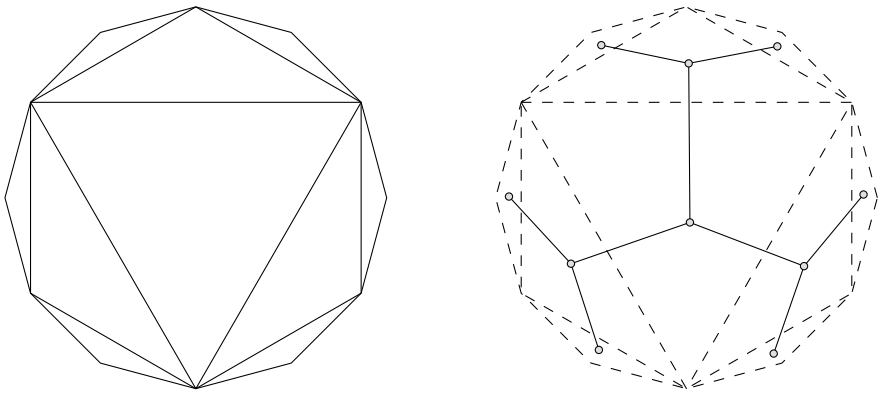


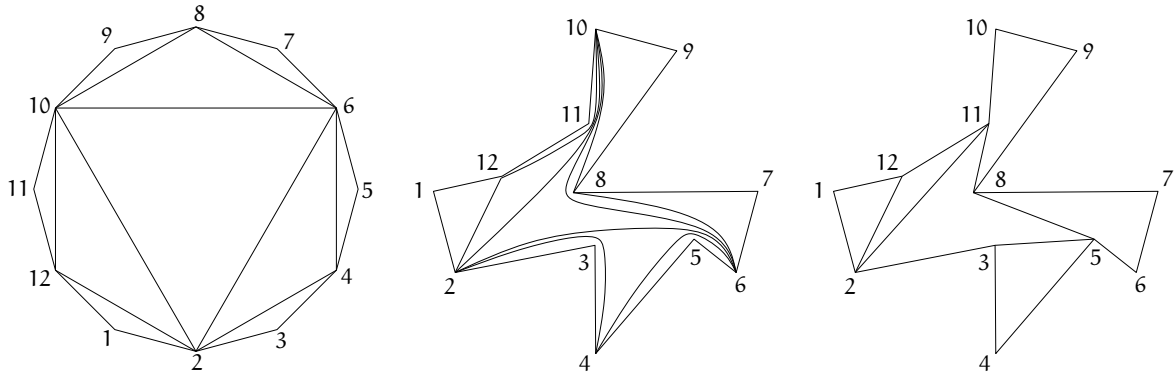
Figure 18.1: A balanced triangulation of the convex dodecagon and its dual tree.

is that any ray starting inside P intersects at most a logarithmic number of triangles: A

walk along the ray corresponds to a path on the triangles of the triangulation because once a ray leaves a triangle, it cannot re-enter it. That is, we can consider the ray as traversing a path in the dual tree and this tree has logarithmic diameter.

The data structure consists of a balanced triangulation B for P and a Kirkpatrick point location hierarchy on B. Both can be obtained in linear time and space. A ray shooting query is processed as follows. First find the triangle that contains the source of the ray. Then iteratively find the edge through which the ray leaves the current triangle and continue with the triangle on the other side, until the ray passes through an edge of P. The leaving edges takes can be found in constant time each and since the number of triangles traversed is logarithmic, so is the query time.

Balanced Pseudotriangulations. So what did we gain with this approach that is arguably more complicated than the plain binary search? The whole point is that it generalizes to non-convex polygons. To see this, consider a simple polygon P. Imagine to transform P in a continuous way—by moving, stretching, and shrinking its edges—such that eventually its vertices form a convex polygon P'. Now construct a balanced triangulation B' for P'. Finally, imagine what happens to the triangulation B' when transforming P' back to P: The edges of B' (imagine them as rubber bands) are stretched to bend around reflex corners of P, that is, vertices of P whose interior angle is greater than π (for instance, vertices 3, 5, 8, 11, and 12 in Figure 18.2(b)). That is, an edge of B' in P' becomes a



(a) Convex polygon P'. (b) Non-convex polygon P. (c) Pseudotriangulation of P.

Figure 18.2: Transforming a balanced triangulation.

path in P whose endpoints are convex vertices of P and whose interior vertices are reflex vertices of P. Such a path is also called a *geodesic path* because it corresponds to the shortest path that connects its endpoints while staying completely inside the polygon. (The name stems from the setting where the earth surface is considered a subset of $\mathbb{R}^3$ and a shortest path between two points on the surface has to stay on the surface, whereas the true shortest path in $\mathbb{R}^3$ would simply pass through the interior of the planet.)

In this way, a triangle of B' becomes a polygon in P; but not an arbitrary polygon but a polygon that we will call a *kite*: a pseudotriangle (simple polygon with exactly three

convex vertices) with a possibly empty path attached to each convex vertex. Leaving out these paths, we obtain a balanced pseudotriangulation $\mathcal{B}$ for P (Figure 18.2(c)).

Lemma 18.1 *Let $\mathcal{B}$ be a balanced pseudotriangulation of a simple polygon P on n vertices. Then any ray emanating from a point inside P (including the boundary) intersects at most $O(\log n)$ edges of $\mathcal{B}$.*

Proof. A traversal along the ray corresponds to a path in the dual $\mathcal{B}^*$. As $\mathcal{B}^*$ is a subgraph of $(B')^* = B^*$ which in turn is a tree on $n-2$ vertices, $\mathcal{B}^*$ has logarithmic diameter. □

Computing geodesic paths. The construction of a balanced pseudotriangulation hinges on the computation of geodesic paths within the polygon P. We will show next how to efficiently compute these paths.

As a preprocessing step, construct an arbitrary triangulation T for P. In principle, this can be done in linear time [**?**]. But for our purposes here, the $O(n \log^* n)$ algorithm using a trapezoidal decomposition as discussed in the lecture does the job. Also construct the dual tree T^* of T that can be obtained from T in linear time.

How to find a geodesic path between two vertices p and q of P? If p and q are vertices of some triangle from T, this is obviously trivial. Hence suppose there is no such triangle in T. First compute the unique path $t_1, \ldots, t_k$ of triangles in T^* such that $p \in t_1$, $p \notin t_2$, $q \notin t_{k-1}$, and $q \in t_k$. This path can be found in $O(k)$ time starting from, say, t_1; since we know the ordering of the vertices around P, it needs an index comparison only to determine the next triangle on the path.

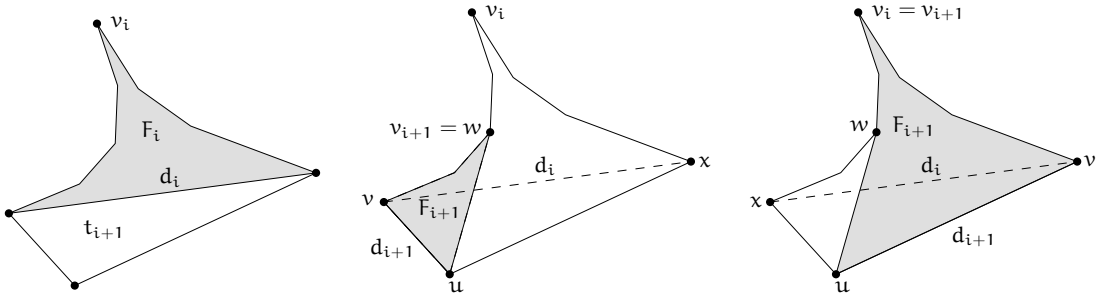
Denote by d_i the edge shared by t_i and t_{i+1}, for $1 \leq i < k$. Iteratively compute the so-called *funnel* F_i from some vertex v_i to d_i, the region bounded by the shortest paths from v_i to the endpoints of d_i. Initially $v_1 = p$ and $F_1 = t_1$. We would like to maintain the following invariants.

1) F_i is a pseudotriangle one concave chain of which is the line segment d_i, and v_i is the vertex opposite to d_i.

2) For any point $v \in t_{i+1}$, a shortest path from p to v passes through v_i.

In order to obtain F_{i+1} from F_i we distinguish two cases. By definition d_i and d_{i+1} share a vertex. Denote this vertex by v, and let u denote the other endpoint of d_{i+1} and let x denote the other endpoint of d_i. Seen from d_i, the funnel F_i consists of two concave chains which meet in v_i. Consider the inner tangent from u to these chains. If the point w of tangency is on the chain from v to v_i (Figure 18.3(b)), then set $v_{i+1} = w$ and let F_{i+1} be bounded on one side by the chain from w to v and on the other side by the edge wu. Otherwise (Figure 18.3(c)), let $v_{i+1} = v_i$ and obtain F_{i+1} from F_i by replacing the chain from w to x by the edge wu.

It is easy to see that this procedure maintains the above invariants and therefore the sequence of distinct vertices v_i that appear in the course of the construction form the desired geodesic path.



(a) Funnel F_i. (b) w on chain to v. (c) w on the other chain.

Figure 18.3: Constructing funnels for geodesic paths.

The search for the point w of tangency can be done using a simple linear search along the boundary of F_i, starting from x. As every point considered during this search is permanently discarded for the remainder of this path computation, the overall number of steps spent in tangency searches is linear, that is, $O(k)$.

Lemma 18.2 *For a simple polygon P, a geodesic path that traverses k triangles of a triangulation for P, can be computed in $O(k)$ time.*

Altogether, we obtain the following theorem.

Theorem 18.3 *For a simple polygon P on n vertices, a balanced pseudotriangulation can be constructed in $O(n \log n)$ time.*

Proof. We construct all geodesic paths as described above, according to Lemma 18.2 in time linear in the length of the path, that is in the number of triangles from T that are traversed by it. By Lemma 18.1 any single ray, in particular, any single edge of T intersects at most a logarithmic number of geodesic paths. Therefore, the total number of triangles that appear in the course of all geodesic path computations is bounded by $O(n \log n)$. □

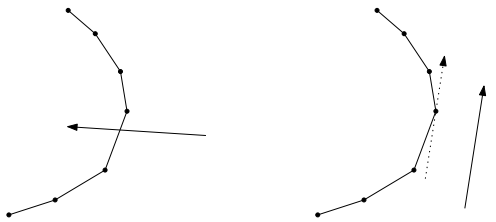
Ray shooting. Once we have a balanced pseudotriangulation $\mathcal{B}$, a data structure for ray shooting queries is straightforward: Build a point location data structure on top of $\mathcal{B}$ to find the pseudotriangle t that contains a query point q. Starting from t, traverse $\mathcal{B}$ to find the edge hit by the query ray r. In each step of the traversal, we have to find the edge of a pseudotriangle hit by r. If the edges of each concave chain are stored in an array, we can find the (first) intersection, if any, of r with each chain using binary search. Then select the leaving edge from the constant number of candidates found.

The runtime needed to process a query in this way is dominated by the traversal. Due to Lemma 18.1 at most a logarithmic number of pseudotriangles is traversed, in each of which we spend $O(\log n)$ time for the binary searches. In total, this yields $O(\log^2 n)$ query time.

In the following, we will briefly sketch how to get the query time down to $O(\log n)$. First, for every pseudotriangle t in $\mathcal{B}$, maintain its three concave chains as a *weight-balanced* binary search tree, where the weight $w(e)$ of an edge e is the size (number of edges) of the *bay* of P behind e. (Imagine P as a lake. Cutting P along e separates P into two parts. On one side is t, the part on the other side is what we refer to as "the bay behind e".)

If we denote the total weight of all bays attached to a chain C by W, then the cost of discovering e in a weight-balanced search tree for C is $O(1 + \log(W/w(e)))$. Consider a traversal of $\mathcal{B}$ and denote the size of the bays of P entered by $n_1, \ldots, n_k$. Then the cost of traversing the corresponding weight-balanced search trees is $\sum_{i=1}^{k} O(1 + \log(n_i/n_{i+1})) = O(k + \log n)$.

Still, we cannot afford to search all three chains in this way. Let us have a closer look at possible interactions between the concave chains of a pseudotriangle and a query ray. In a *fly-by* situation (Figure 18.4(a)) the ray is parallel to a tangent to a chain, whereas in a *home-in* situation (Figure 18.4(b)) the ray is directed towards a chain and its slope falls outside the range of slopes along the chain.



(a) home-in. (b) fly-by.

Figure 18.4: Relation between a ray and a concave chain.

In order to detect the fly-by chain (if any), concatenate the edges of all three chains bounding a pseudotriangle into a single array (break up the circular order arbitrarily). By definition, this array is sorted according to slope. Thus, using a single binary search on this array we can find the possible point of tangency and thereby detect the fly-by chain, if any. To test whether the ray flies by, indeed, use the weighted search tree for that chain and observe that the possible point of tangency is known already, and this point determines where to go in the search tree.

Finally, in order to not spend too much time in fly-by searches, take the three weight-balanced search trees of a pseudotriangle and join them by a fictitious root vertex. Also join each leaf of a search tree with the leaf in the search tree "on the other side", that is, join the two leaves corresponding to the same edge in $\mathcal{B}$. To each fictitious root vertex associate the fly-by array of the pseudotriangle. To the resulting graph apply fractional cascading with the fly-by arrays as catalog graphs and letting the non-root vertices start with an empty catalog. In this way, using linear time preprocessing we reduce the cost of k successive fly-by searches from $O(k \log n)$ to $O(k + \log n)$ time. Also note that after changing from one pseudotriangle to an adjacent one we can afford to traverse the path

to the fictitious root because altogether that amounts to nothing else but tracing the ray in reverse direction.

Theorem 18.4 *For a simple polygon* P *on* n *vertices, there is a data structure to answer ray shooting queries (which edge of* P *is hit by a query ray* r *starting from a point inside* P*) in* $O(\log n)$ *time. The data structure can be built in* $O(n \log n)$ *time and using* $O(n)$ *space.*

Remarks. The idea to use pseudotriangulations (back then called geodesic triangulations) for ray shooting queries is by Chazelle et al. [**?**]. They also give a linear time algorithm to construct a balanced pseudotriangulations, used more elaborate data structures. The geodesic path computation as described above is due to Lee and Preparata [**?**].