Spanner visibility graph

Lev Nachmanson

## The task

We are following . The difference is that we do not build the Voronoi diagram but build the spanner graph directly. Let *C* be a cone with the bisector parallel to the axis y and looking upwards as in Figure 1.

Figure : Cone *C*

For points *a, b* in plane for the purposes of our task we set *d(a,b)=Math.Abs(a.y-b.y)* and will call it the cone distance between *a* and *b*. For point a on the plane by *C­a* we denote cone *C* translated such that its apex coincides with point *a*. We are given a set of obstacles *P* where each obstacle is a convex polygon. Let *V* be the set of vertices of polygons from *P*. We call vertex *u* in *V C*-closest to *v* if *u* is visible from *v, u* belongs to *Cv* and *d(u,v)* is smallest for such vertices. Our task is to build graph *G* such that for every *v* in *V* with *Cv* intersected with *V* not empty there will be an edge *(v,u)* in *G* such that *u* is *C*-closest to v. When we rotate cone *C* to cover the whole circle of directions we will build the graph from (Clarkson, 1987).

## Data structures

Our algorithm is a sweepline algorithm where we sweep with a horizontal line going bottom top. During the sweep the line will pass through *EventSites*. Each *EventSite* is associated with a point on the plane denoted by *EventSite.p*. We consider lexicographical order of the plane points where *a < b* iff *a.y < b.y* ||*(a.y == b.y && a.x < b.x)*. We maintain a priority queue *Q* of *EventSites* where the smallest according to this lexicographical order elements will be at the top of the queue. During the sweep we construct *Segments*. Each *Segment* has the start point denoted by *Segment.start* and the direction denoted by *Segment.dir*. For each Segment we will have *Segment.dir.y > 0*. We will sort *Segments* along a sweep line. For a real number z we denote by Horz the line which points have *z* as the *y* coordinate. Let *s0* and *s1*be *Segments* with starts not above the line, that is *s0.start.y* ≤ *z* and *s1.start.y* ≤ *z*. *Horz*defines the order on *s0* and *s1* in the following way: *s0* < *s1* if and only if the point of intersection of the *s0* with *Horz* is to the left of the point of intersection of *Horz* with *s1*, or *s0.start*==*s1.start* and the direction of *s0* is to the left of *s1* when looking in the direction of *y*-axis, see Figure 2.

*s1*

*s0*

*Horz*

Horz

*s0*

*s1*

Figure : *s0*is less than *s­1* in the order defined by *Horz*

### EventSites

*Event* is the base class for event classes. *Event.site* is the point on the plane associated with the event. There are two kinds of events: *VertexEvent* and *IntersectionEvent*. *VertexEvent* happens when the sweepline passes through a vertex and an *IntersectionEvent* happens when the sweepline passes through an intersection of two *Segments*.

### Segments

There are two kinds of segments: *ConeSides* and *EdgeSegments,* as Figure 3 shows*.* In turn there are two types of of *EdgeSegments*: *LeftEdgeSegment* and *RightEdgeSegment,* see Figure 4. Also there is *LeftConeSide* and *RightConeSide*. *ConeSide* points to *Segment.apex* and to the other *ConeSide* of the cone.

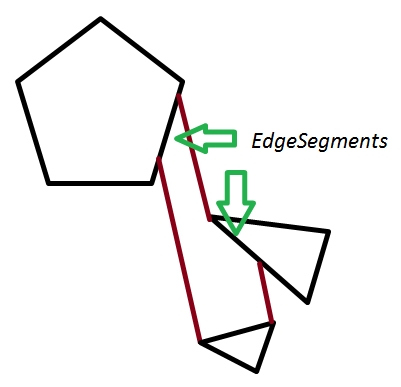


Figure : *EdgeSegments* and ConeSides

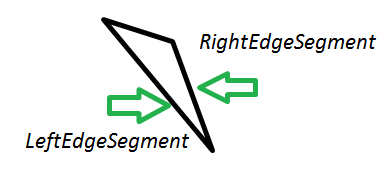


Figure : LeftEdgeSegment and RightEdgeSegment

*EdgeSegment* keeps a list of *ConeSides* called *EdgeSegment.ObscuredConeSides* of some active cones; intuitevely a segment points to a *ConeSide* if for the current sweepline the segment represents the side of the cone. More precisely the side belongs to the list when there is a point *t* on the sweepline which is outside of all obstacles, belongs to the side cone, and if we look to the direction of the cone side along the sweepline we see the *EdgeSegment* first, as in Figure 5.

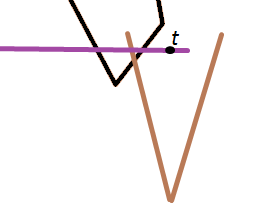


Figure : *ObscuredConeSides*

### ActiveSegmentsTree

During the sweep we will keep S*egments* that are active in a tree *T* ordered by the order defined by the current Horz.

# Algorithm in pseudocode

BuildGraph(){

Init();

Sweep();

}

Init(){

EnqueueAllObstacleVertices();

CreateEmptySegmentTree();

}

Sweep(){  
 while( Q is not empty )  
 ProcessEvent (Q.Dequeue());  
}

ProcessEvent(EventSite event){  
 z=event.site.y;  
 if(event is VertextEvent) ProcessVertexEvent(event);  
 else ProcessIntersectionEvent(event);  
}

ProcessVertexEvent(VertexEvent vertexEvent){  
 ProcessActiveConesContainingVertexEvent(vertexEvent.site);  
 TryToCreateConeAtVertex(vertexEvent.site);  
}  
ProcessActiveConesContainingVertexEvent(Point site){  
The version that we present here is a bit simplified. For a Segment s we denote by s(z) the x-coordinate of the intersection of s with Horz. Let s be the first right cone side in T with s(z) ≥ site.x and s.leftSide(z)<=site.x is a such s exists, otherwise set s to null. If s is not null for each left cone side k with   
 k(z)>= s.leftSide(z) and k(z)<=site.x create a new edge of the spanner graph (k.apex, site) and delete k and k.rightSide. See Figure 6.

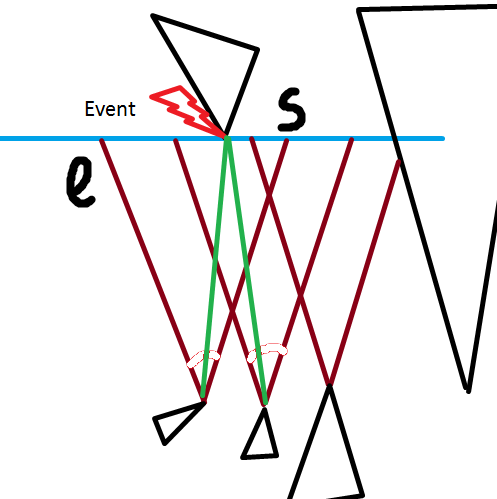


Figure : green edges are created and cones containing them are deleted

}

TryToCreateConeAtVertex((Point site){  
}

Clarkson, K. L. (1987). Approximation Algorithms for Shortest Path Motion Planning. *Annual ACM Symposium on Theory of Computing* , 56-65.