CS 6301.008.18S Lecture—March 8, 2017

Main topics are #robot_motion_planning, #shortest_paths, and #visibility_graphs.

Shortest Paths

- Tuesday, we looked at robot motion planning.
- We started with a special case of moving a point robot through a set of polygonal obstacles in the plane.
- Today, we're going to focus on that case again, but instead of computing just any path from s to t, we're going to search for the shortest path.
- In other words, given a set of disjoint polygonal obstacles with n vertices total, and two points s and t outside the obstacles, we want to compute the shortest path through the plane that avoids the interior of any of the obstacles.

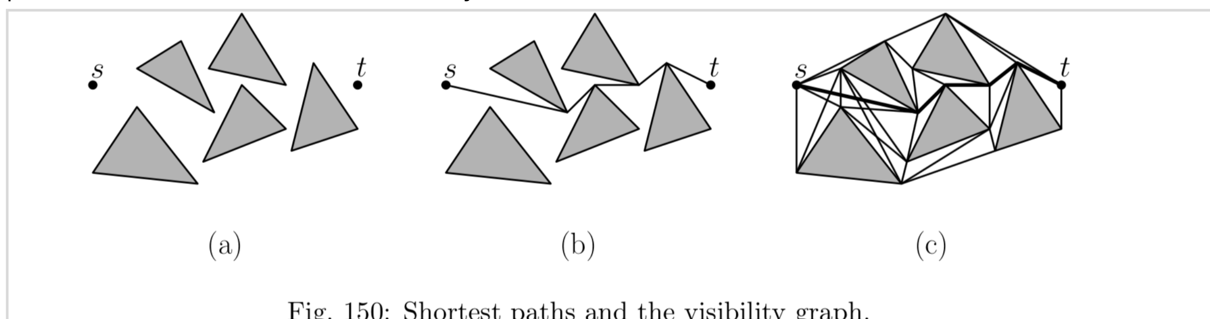


Fig. 150: Shortest paths and the visibility graph.

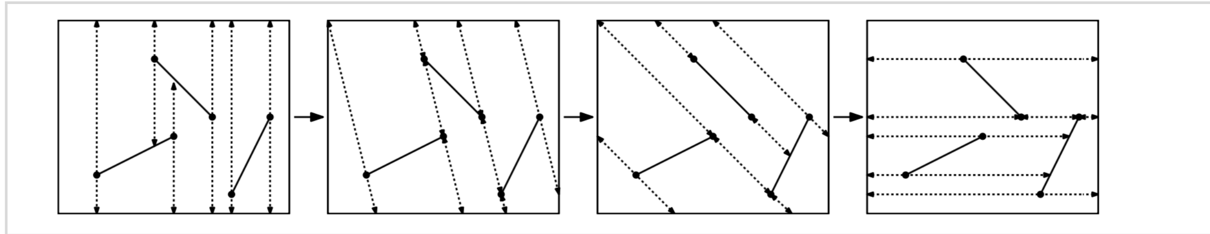
- In other words, we want the shortest s,t-path in the free space.
- Today, we'll look at an O(n^2) time algorithm for solving the problem. There's an O(n log n) time algorithm also, but it's much more complicated.
- So first thing we need to figure out: unlike in a graph, there are *a lot* of paths in the plane and a lot of places our path could turn. Or what if… it actually had curved sections? Fortunately:
- Claim: The shortest path from s to t avoiding polygonal obstacles is a polygonal curve whose vertices are s, t, or vertices of the obstacles.
- Proof:
  - Let pi be the shortest path.
  - If it is not a polygonal curve, then there is some point p in the interior of the free space through which the path passing through p is not locally a line segment.
  - If we look at a tiny neighborhood/ball around p, it contains no other obstacles, s, or t. We can replace the portion of pi through the neighborhood by a single line segment which is shorter and still does not go through any obstacles.
  - So pi is a polygonal curve. But what if there's a vertex v that is not s, t, or a vertex of an obstacle?
  - Again, look at a tiny neighborhood around v and replace the portion of pi through the

neighborhood with a shorter line segment.

- So the shortest path is a bunch of line segments between s, t, and vertices of the obstacles.
- These line segments do not pass through the interior of the obstacles.
- Call two points p and q *mutually visible* if the line segment pq does not intersect the interior of any obstacle.
- The *visibility graph* is the graph with vertices s, t, and vertices of the obstacles. Two vertices u and v are connected by an edge if u and v are mutually visible.
- The graph is *not* necessarily planar and may have Omega(n^2) edges.
- But if we have the visibility graph, we can easily compute the shortest path from s to t: weight each edge by the length of its line segment. Now compute the shortest path from s to t using Dijkstra's algorithm with a Fibonacci heap in O(n log n + n^2) = O(n^2) time.
- So how do we find that visibility graph?

Computing the Visibility Graph

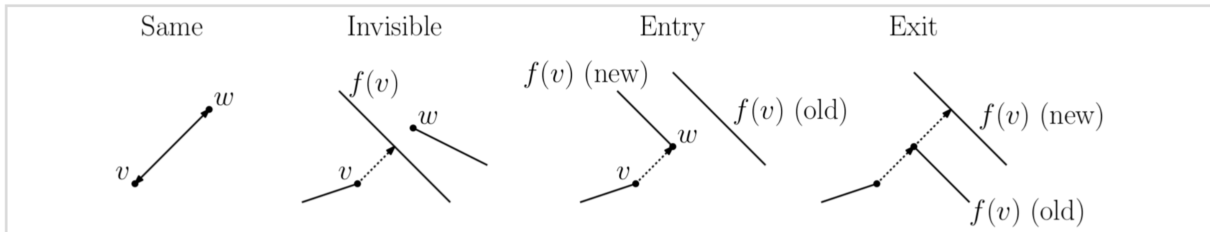
- The textbook presents an algorithm that computes the edges incident to each vertex one vertex at a time. It takes O(n log n) time per vertex for O(n^2 log n) time total. We're solving the problems completely independently for each vertex, so it seems unlikely we can speed up this strategy.
- I'm going to present an algorithm from Mount's notes based on dual line arrangements where we'll try to find edges from all vertices simultaneously. The way I'll present the algorithm suggests it should run in O(n^2 log n) time also, but there are ways to speed it up to O(n^2) since we're working with a dual arrangement and all vertices at once.
- Also to keep things simple, I'll focus on a slightly easier problem: Given n line segments, compute their visibility graph. Two endpoints p and q are mutually visible if they are on the same input segment or pq does not pass through the interior of any other segment.
- Here's a high level way of thinking about this algorithm. Remember how we shot rays up and down from each endpoint to compute the trapezoid map? In a sense, each endpoint is looking straight up and down, and vision is blocked when the ray hits a line segment.
- Let theta be the slope of these rays. You could say theta = -infinity.
- Now imagine we continuously increase theta from -infinity to infinity, performing an angular sweep. The rays along which the endpoints are looking start to rotate counterclockwise.
- We end up with these *events* where some ray would passes through a second endpoint.
- If the ray doesn't pass through another line segment before hitting the second endpoint, then that second endpoint is visible from the first.



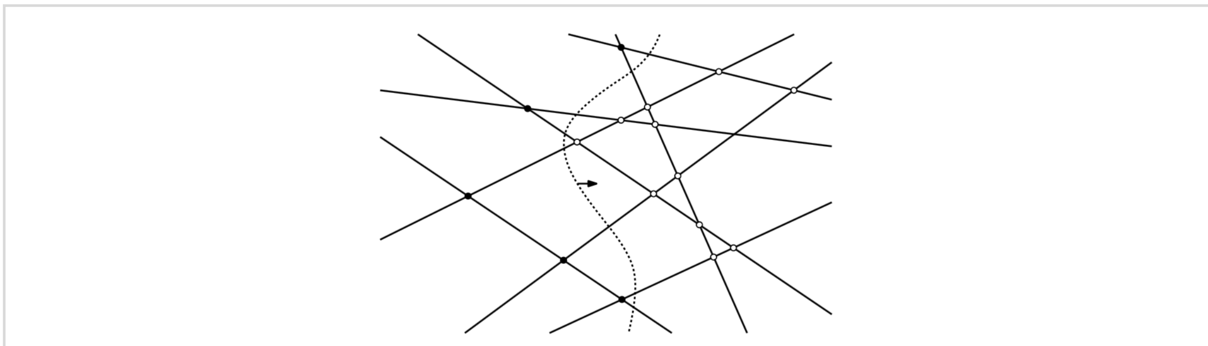
- This figure shows just the visible points along each ray.
- Our goal is to recognize these events and determine whether the visible portion of the ray actually reaches the second endpoint when the occur.

Dual Interpretation and Events

- It may be easier to think about this process in the dual graph.
- A point v = (v_a, v_b) is dual to line v^* : y = v_a x - v_b.
- So if a ray of slope theta goes from u to v, then u^* and v^* intersect at dual x-coordinate theta.
- So an angular sweep in the primal plane is the same as a left-to-right sweep in the dual plane. Events occur when the dual sweep lines passes over a vertex of the dual line arrangement. To figure out the correct order of events, compute the dual line arrangement and sort the vertices in left to right order.
- What will we use as our sweep line status? We mostly care about how far each ray goes before it hits its first line segment. So, for each endpoint v we store two line segments f(v) and b(v) standing for the endpoints of the *forward and backward bullet paths*.
- If you read ahead, this is the main difference from the textbook algorithm. That algorithm essentially stores *all* the line segments intersected by each ray in some data structure. Here, though, we are working with all endpoints simultaneously instead of one at a time. We only need to store the first segment hit by each ray.
- So let's look at an event. The ray from vertex v passes through vertex w and vice versa. We can tell which case we're handling in O(1) time.
  - Same segment: If v and w are endpoints of the same segment, they are visible and we add vw to the visibility graph.
  - Invisible: Suppose the same ray from v hits w and f(v), but f(v) is closer to v than w. We just move on to the next event since f(v) stays the same and v and w are not mutually visible.
  - Segment entry: If w is visible from v, then we add vw to the visibility graph. If the ray is about to *start* crossing w's segment, then we set f(v) to be that segment.
  - Segment exit: But if w is visible from v and the ray is about to leave f(v), we need to update f(v). However, the ray from v passes through w and then f(w). So we set f(v) to be f(w).



- So running time. All we need for data structures is f(v) and b(v) for each endpoint and the event queue for the O(n^2) events. We can lookup next events in O(log n) time each. Or, we could instead sort all event dual points in left-to-right order after building the arrangement in O(n^2 log n) time and then find events in O(1) time each by using a for loop. Actually processing each event takes only O(1) time. The total running time is O(n^2 log n).
- That log n in the running time comes from insisting we handle events in left-to-right order. We have to sort or use a priority queue.
- However, we could instead use the "topological plane sweep" I briefly mentioned last week for line arrangements.
- Essentially, you relax the order in which you process events. The vertices on each dual line are processed in left to right order, but vertices on different lines may be swept in different orders.
- Instead of the dual sweep line being a real vertical line, it's now more accurate to think of it as a *pseudoline* that intersects each line of the arrangement exactly once.



- With topological plane sweep, we can lookup next events in only O(1) time each. The whole algorithm runs in O(n^2) time.

Simple Polygon Robots Again

- Let's finish up by going back to Tuesday's final problem. We're given a convex polygonal robot with O(1) vertices, polygonal obstacles with n vertices total, and points s and t in the free space.
- Say we want to translate the robot from configuration s to configuration t while moving it as little as possible.
- We compute the free space in O(n log^2 n) time using Minkowski sums and the divide and

conquer algorithm.

- The free space is bounded by a bunch of polygons with O(n) total complexity. So we can run today's shortest path algorithm in the configuration space in O(n^2) time.