CS4733 Class Notes

1 2-D Robot Motion Planning Algorithm Using Grown Obstacles

- Reference: *An Algorithm for Planning Collision Free Paths Among Poyhedral Obstacles* by T. Lozano-Perez and M. Wesley.
- This method of 2-D motion planning assumes a set of 2-D convex polygonal obstacles and a 2-D convex polygonal mobile robot.
- The general idea is grow the obstacles by the size of the mobile robot, thereby reducing the analysis of the robot's motion from a moving area to a single moving point. The point will always be a safe disatnce away form each obstacle due the growing step of each obstacle. Once we shrink the robot to a point, we can then find a safe path for the robot using a graph search technique.

2 Algorithm

- Method I: Grow each obstacle in the scene by the size of the mobile robot. This is done by finding a set of vertices that determine the grown obstacle (see figure 1). First, we reflect the robot about its X and Y axes. Placing this reflected object at each obstacle vertex, we can map the robot reference points when added to these vertices. This constitutes a grown set of vertices.
- Given the grown set of vertices, we can find its convex hull and form a grown polygonal obstacle. The obstacle is guaranteed to be the convex hull.
- We can now create a visibility graph (see figure 2). A visibility graph is an undirected graph $G = (V, E)$ where the V is the set of vertices of the grown obstacles plus the start and goal points, and E is a set of edges consisting of all polygonal obstacle boundary edges, or an edge between any 2 vertices in V that lies entirely in free space except for its endpoints. Intuitively, if you place yourself at a vertex, you create an edge to any other vertex you can see (i.e. is visible). A simple algorithm to compute G is the following. Assume all N vertices of the G are connected. This forms $\frac{N\cdot(N-1)}{2}$ edges. Now, check each edge to see if it intersects (excepting its endpoints) any of the grown obstacle edges in the graph If so, reject this edge. The remaining edges (including the grown obstacle edges) are the edges of the visibility graph. This algorithm is brute force and slow ($O(N^3)$ but simple to compute. Faster algorithms are known.
- The shortest path in distance can be found by searching the Graph G using a shortest path search (Dijkstra's Algorithm) or other heuristic search method.
- Method II: Every grown obstacle has edges from the original obstacle and edges from the robot. These edges occur in order of the obstacle edge's outward facing normals and the inward facing normals of the robot. By sorting these normals, you can construct the boundary of the grown obstacle(see figures 4.14. and 4.15 in this handout from *Planning Algorithms*, S. Lavalle, Cambridge U. Press, 2006. http://planning.cs.uiuc.edu/)

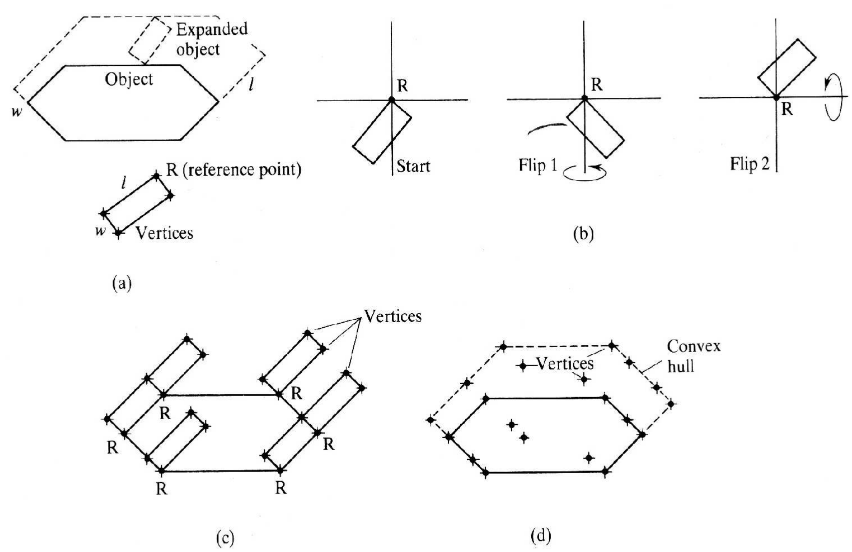


Figure 1: Reflection method for computing grown obstacles (from P. McKerrow, *Introduction to Robotics*).

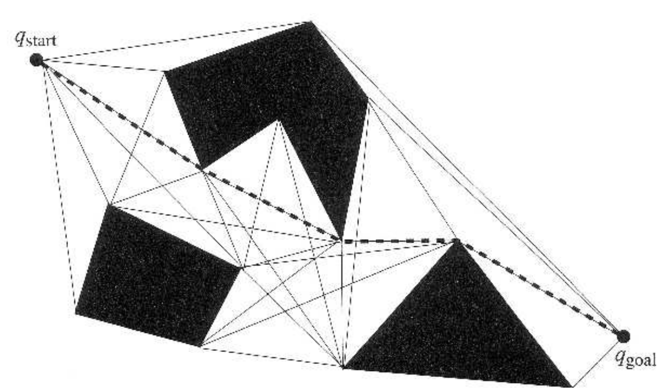


Figure 2: Visibility graph with edges.

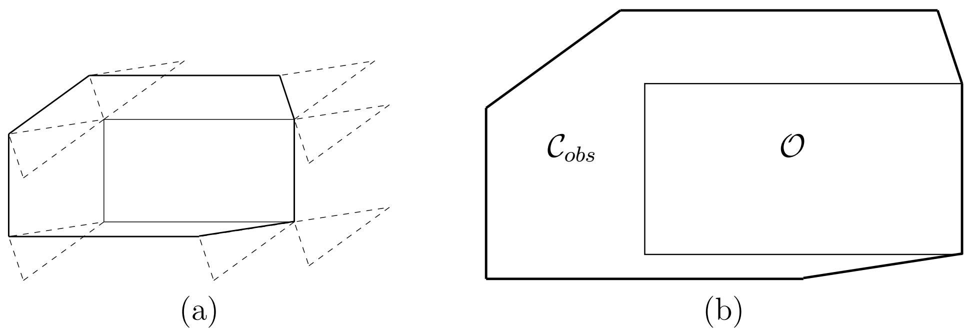


Figure 4.14: (a) Slide the robot around the obstacle while keeping them both in contact. (b) The edges traced out by the origin of $\mathcal{A}$ form $\mathcal{C}_{obs}$.

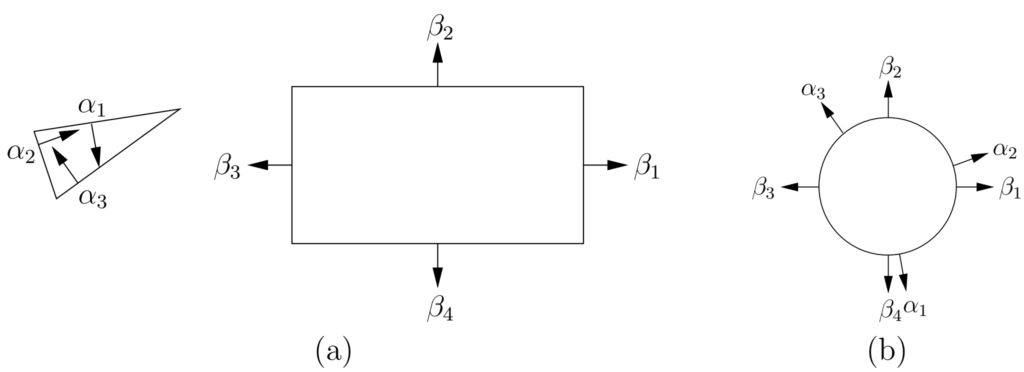


Figure 4.15: (a) Take the inward edge normals of $\mathcal{A}$ and the outward edge normals of $\mathcal{O}$. (b) Sort the edge normals around $\mathbb{S}^1$. This gives the order of edges in $\mathcal{C}_{obs}$.

3 Complications

- How do we add rotation to the problem. The above algorithm assumes that the robot only translates in space.
- Method I: Grow the obstacles by an amount that guarantees the robot will not intersect no matter what orientation it is in. Problem: Overly conservative method, may not find a path when one exists.
- Method II: Create *slices* that are separate solutions to the planning problem for different orientations. If you can find an overlap region of the slices, this means you can effectively change orientation at these positions without collision. Path planning now becomes a search between different configurations of the robot.
- How do we handle non-convex robots and obstacles? Any non-convex polygon can be brolen up into a set of convex polygons. Once we decompose the objects into convex regions, we can run the normal obstacle growing procedure and union the resulting obstacles (see figure 3).
- How do we extend this algorithm to 3-D objects? Assuming translational motion only, we can grow the polyhedral obstacles by polyhedral mobile robots. Unfortunately, the shortest path is now not constrained to be via the vertices of the grown objects, so a lengthier and more complicated search is necessary.
- We can use a higher dimensional, discretized configuration space approach (see the paper *A Simple Motion Planning Algorithm for General Purpose Robot Manipulators* by T. Lozano-Perez.

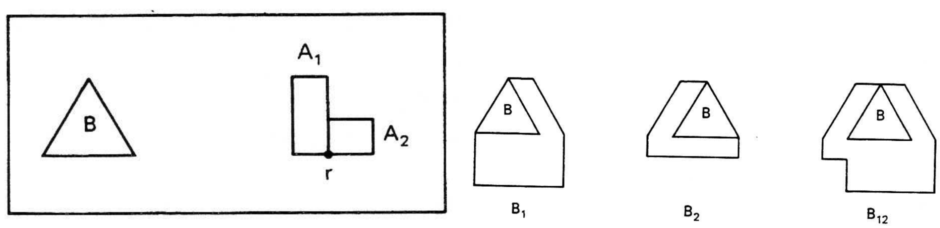


Figure 3: Concave objects. Decompose concave robot A into convex regions, Compute grown space of each convex region with obstacle B, union the resulting grown spaces (from R. Schilling, *Fundamentals of Robotics*).

4 Finding the Convex Hull of a 2-D Set of Points

- Reference: *Computational Geometry in C* by J. O'Rourke
- Given a set of points S in a plane, we can compute the convex hull of the point set. The convex hull is an enclosing polygon in which every point in S is in the interior or on the boundary of the polygon.
- An intuitve definition is to pound nails at every point in the set S and then stretch a rubber band around the outside of these nails - the resulting image of the rubber band forms a polygonal shape called the Convex Hull. In 3-D, we can think of "wrapping" the point set with plastic shrink wrap to form a convex polyhedron.
- A test for convexity: Given a line segment between any pair of points inside the Convex Hull, it will never contain any points exterior to the Convex Hull.
- Another definition is that the convex hull of a point set S is the intersection of all half-spaces that contain S. In 2-D, half spaces are half-planes, or planes on one side of a separating line.

5 Computing a 2-D Convex Hull: Grahams's Algorithm

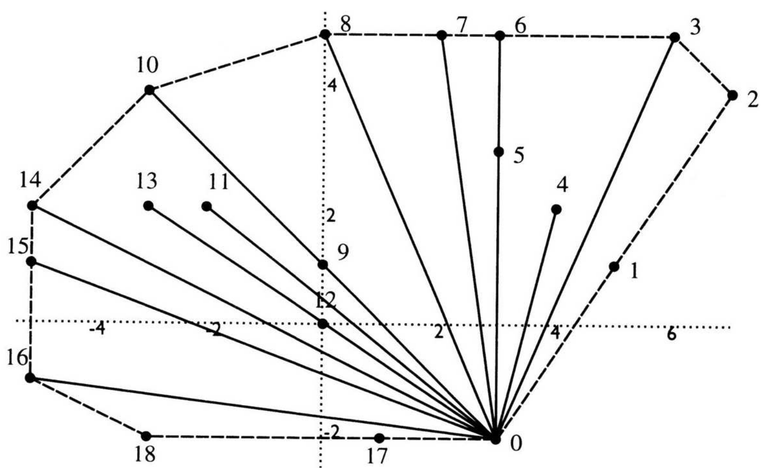
There are many algorithms for computing a 2-D convex hull. The algorithm we will use is Graham's Algorithm which is an $O(N\ Log\ N)$ algorithm (see figure 4).

1. Given N points, find the righmost, lowest point, label it P_0.
2. Sort all other points angularly about P_0. Break ties in favor of closeness to P_0. Label the sorted points $P_1 \cdots P_{N-1}$.
3. Push the points labeled P_{N-1} and P_0 onto a stack. These points are guaranteed to be on the Convex Hull (why?).
4. Set $i = 1$
5. While $i < N$ do

   If P_i is strictly $left$ of the line formed by top 2 stack entries (P_{top}, P_{top-1}), then Push P_i onto the stack and increment i; else Pop the stack (remove P_{top}).
6. Stack contains Convex Hull vertices.

6 Finding Shortest Paths: Dijkstra's Algorithm

1. We want to compute the shorterst path distance from a source node S to all other nodes. We associate lengths or costs on edges and find the shortest path.
2. We can't use edges with a negative cost. Otherwise, we can take take endless loops to reduce the cost.
3. Finding a path from vertex S to vertex T has the same cost as finding a path from vertex S to all other vertices in the graph (within a constant factor).
4. If all edge lengths are equal, then the Shortest Path algorithm is equivalent to the breadth-first search algorithm. Breadth first search will exapnd the nodes of a graph in the minimum cost order from a specified starting vertex (assuming equal edge weights everywhere in the graph).
5. **Dijkstra's Algorithm**: This is a greedy algorithm to find the minimum distance from a node to all other nodes. At each iteration of the algorithm, we choose the minimum distance vertex from all unvisited vertices in the graph,
   - There are two kinds of nodes: **settled** or closed nodes are nodes whose minimum distance from the source node S is known. **Unsettled** or open nodes are nodes where we don't know the minimum distance from S.
   - At each iteration we choose the unsetteld node V of minimum distance from source S. This settles (closes) the node since we know its distance from S. All we have to do now is to update the distance to any unsettled node reachable by an arc from V. At each iteration we close off another node, and eventually we have all the minimum distances from source node S.



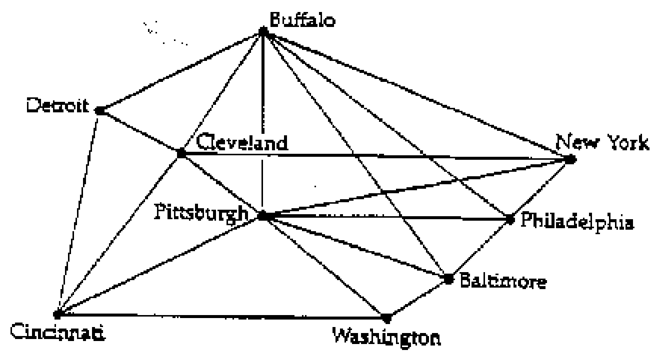
Below is shown the stack (point indices only) and the value of `i` at the top of the `while` loop. The stack is initialized to (0, 18), where the top is shown leftmost (the opposite of our earlier convention). Point p_1 is added to form (1, 0, 18), but then p_2 causes p_1 to be deleted, and so on. Note that p_{18} causes the deletion of p_{17} when $i = 18$, as it should. For this example, the total number of iterations is $29 < 2 \cdot n = 2 \cdot 19 = 38$.

```
i= 1:    0, 18
i= 2:    1, 0, 18
i= 2:    0, 18
i= 3:    2, 0, 18
i= 4:    3, 2, 0, 18
i= 5:    4, 3, 2, 0, 18
i= 5:    3, 2, 0, 18
i= 6:    5, 3, 2, 0, 18
i= 6:    3, 2, 0, 18
i= 7:    6, 3, 2, 0, 18
i= 7:    3, 2, 0, 18
i= 8:    7, 3, 2, 0, 18
i= 8:    3, 2, 0, 18
i= 9:    8, 3, 2, 0, 18
i=10:    9, 8, 3, 2, 0, 18

i=10:    8, 3, 2, 0, 18
i=11:    10, 8, 3, 2, 0, 18
i=12:    11, 10, 8, 3, 2, 0, 18
i=13:    12, 11, 10, 8, 3, 2, 0, 18
i=13:    11, 10, 8, 3, 2, 0, 18
i=13:    10, 8, 3, 2, 0, 18
i=14:    13, 10, 8, 3, 2, 0, 18
i=14:    10, 8, 3, 2, 0, 18
i=15:    14, 10, 8, 3, 2, 0, 18
i=16:    15, 14, 10, 8, 3, 2, 0, 18
i=16:    14, 10, 8, 3, 2, 0, 18
i=17:    16, 14, 10, 8, 3, 2, 0, 18
i=18:    17, 16, 14, 10, 8, 3, 2, 0, 18
i=18:    16, 14, 10, 8, 3, 2, 0, 18,
i=19:    18, 16, 14, 10, 8, 3, 2, 0, 18
```

After popping off the redundant copy of p_{18}, we have the precise hull we seek: (0, 2, 3, 8, 10, 14, 16, 18).

Figure 4: Graham Convex Hull Algorithm example from *J. O'Rourke, Computational Geometry in C*



(a) The graph

	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PITT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	265	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington

EXAMPLE OF DIJKSTRA'S ALGORITHM

			DISTANCES								
			1	2	3	4	5	6	7	8	9
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phl	Pitt	Wash
Initial			0	345	inf	inf	inf	inf	97	230	39
1	1	9	0	345	531	inf	inf	inf	97	230	39
2	1,9	7	0	345	531	inf	inf	189	97	230	39
3	1,9,7	6	0	345	531	696	inf	189	97	230	39
4	1,9,7,6	8	0	345	514	355	inf	189	97	230	39
5	1,9,7,6,8,	2	0	345	514	355	597	189	97	230	39
6	1,9,7,6,8,2	4	0	345	514	355	522	189	97	230	39
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39

Figure 5: Example of Dijkstra's algorithm for finding shortest path

6. Pseudo Code for Dijkstra's Algorithm (see figure 5)

```
Note: initialize all distances from Start vertex S
to each visible vertex. All unknown distances assumed
infinite.  Mark Start Vertex S as VISITED, DIST=0

Dijkstra(Graph G, Source_Vertex S)
{
  While Vertices in G remain UNVISITED
  {
        Find closest Vertex V that is UNVISITED
        Mark V as VISITED
        For each UNVISITED vertex W visible from V
        {
            If (DIST(S,V) + DIST(V,W)) < DIST(S,W)
               then DIST(S,W) =  DIST(S,V) + DIST(V,W)
        }
  }
}
```

7. Sketch of Proof that Dijkstra's Algorithm Produces Min Cost Path

   (a) At each stage of the algorithm, we settle a new node V and that will be the minimum distance from the source node S to V. To prove this, assume the algorithm *does not* report the minimum distance to a node, and let V be the first such node reported as settled yet whose distance reported by Dijkstra, $Dist(V)$, is not a minimum.

   (b) If $Dist(V)$ is not the minimum cost, then there must be an unsettled node X such that $Dist(X)+Edge(X,V) < Dist(V)$. However, this implies that $Dist(X) < Dist(V)$, and if this were so, Dijkstra's algorithm would have chosen to settle node X before we settled node V since it has a smaller distance value from S. Therefore, $Dist(X)$ cannot be $< Dist(V)$, and $Dist(V)$ is the minium cost path from S to V.

8. Improving Dijkstra: A* Algorithm – Heuristic Search

The A* algorithm searches a graph efficiently, with respect to a chosen heuristic. If the heuristic is "good," then the search is efficient; if the heuristic is "bad," although a path will be found, its search will take more time than probably required and possibly return a suboptimal path. The path cost at a node is F=G+H, where G is the minimum distance to the current node from the start node, and H is the heuristic cost of traveling from the current node to the goal. A* will produce an optimal path if it's heuristic is optimistic. An optimistic, or admissible, heuristic always returns a value less than or equal to the actual cost of the shortest path from the current node to the goal node within the graph.

The A* search has a priority queue which contains a list of nodes sorted by priority. This priority is determined by the sum of the distance from the start node to the current node and the heuristic at the current node. The first node to be put into the priority queue is naturally the start node. Next, we expand the start node by popping the start node and putting all adjacent nodes to the start node into the priority queue sorted by their corresponding priorities (path costs). Note that only unvisited nodes are added to the priority queue. At each step, the highest priority node (i.e. least cost node) is dequeued and expanded until the goal is reached. A* is greedy in that it tries to skew the search towards the goal. Breadth first search can be thought of as search with heuristic function H=0 (i.e. no heuristic).

A* Search on 4- neighbor Grid

i Breadth First search expands more nodes than A*

ii A* with a heuristic function =0 becomes Breadth First Search

iii A* is admissible if heuristic cost is an UNDERESTIMATE of the true cost

	0	1	2	3	4	5
0	S					
1						
2						
3						
4						G

Example 1

	0	1	2	3	4	5
0	0					
1	1		12			
2	2		9	13		
3	3		7	10	14	
4	4	5	6	8	11	15

Breadth First Search Node Expansion

	0	1	2	3	4	5
0	0					
1	1					
2	2					
3	3					
4	4	5	6	7	8	9

A* Node Expansion (Example 1)

	0	1	2	3	4	5
0	9	8	7	6	5	4
1	8	7	6	5	4	3
2	7	6	5	4	3	2
3	6	5	4	3	2	1
4	5	4	3	2	1	0

Heuristic (L1 dist to Goal)

	0	1	2	3	4	5
0	S					
1	1					
2	2					
3	3		8	9	10	11
4	4	5	6	7		G

A* Node Expansion (Example 2)

	0	1	2	3	4	5
0	0					
1	1					
2	2					
3	3			8	9	10
4	4	5	6	7		11

A* Final Path
(follow goal node back via
Opener node to compute path)

OPEN LIST - Example 2

Opener	Node	f	g	h	f	expand	order
	[0,0]	0+9	0	9	9	0	
0,0	[1,0]	1+8	1	8	9	1	
1,0	[2,0]	2+7	2	7	9	2	
2,0	[3,0]	3+6	3	6	9	3	
3,0	[4,0]	4+5	4	5	9	4	
4,0	[4,1]	5+4	5	4	9	5	
4,1	[4,2]	6+3	6	3	9	6	
4,2	[4,3]	7+2	7	2	9	7	
4,2	[3,2]	7+4	7	4	11	8	
4,3	[3,3]	8+3	8	3	11	9	
3,2	[2,2]	8+5	8	5	13		
3,3	[2,3]	8+4	8	4	12		
3,3	[3,4]	8+2	8	2	10	10	
3,4	[3,5]	9+1	9	1	10	11	
3,4	[2,4]	9+3	9	3	12		
3,5	[4,5]	goal					

Path Cost= f = g + h

g= min distance traveled to this node

h= heuristic cost to goal from this node

(we are using L1 metric cost on 4-neighbor grid)

A* is admissible if heuristic cost is an UNDERESTIMATE of the true cost: h <= C(i,j)