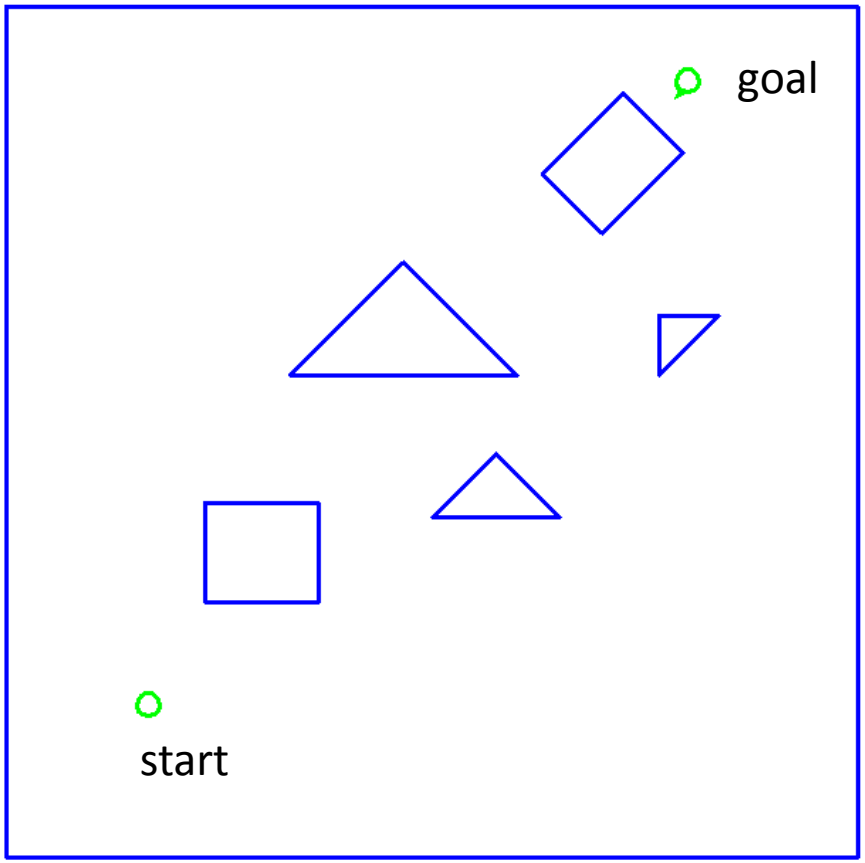
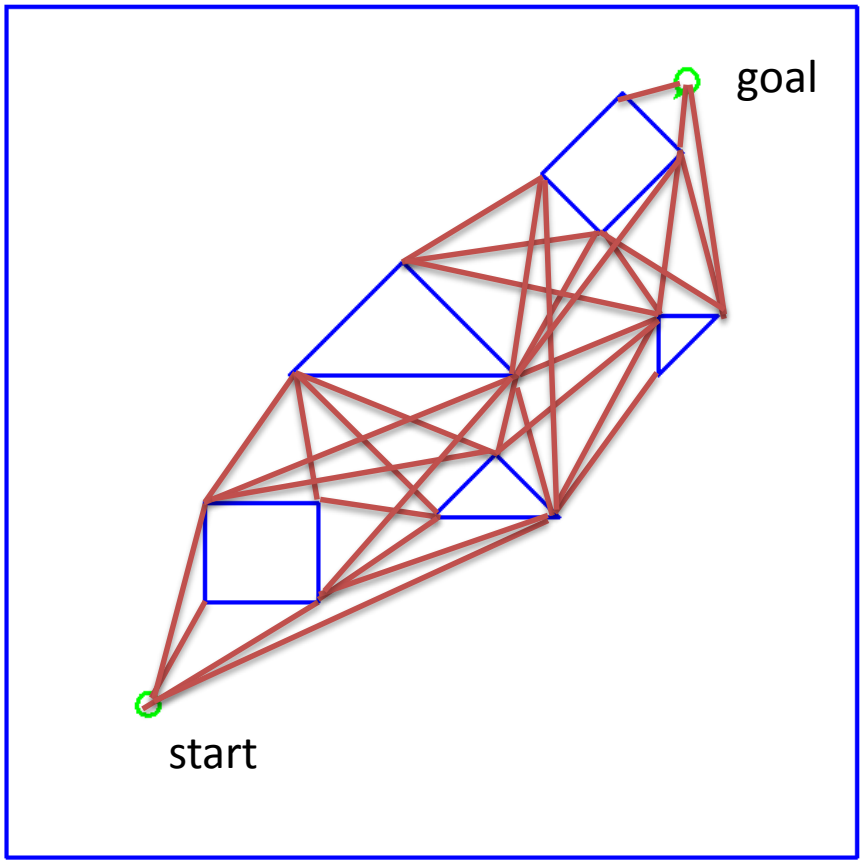
Visibility Graph

How does a Mobile Robot get from A to B?

- Assume robot is a point in 2-D planar space
- Assume obstacles are 2-D polygons
- Create a Visibility Graph:
  - Nodes are start point, goal point, vertices of obstacles
  - Connect all nodes which are “visible” – straight line un-obstructed path between any 2 nodes
  - Includes all edges of polygonal obstacles
- Use A* to search for path from start to goal

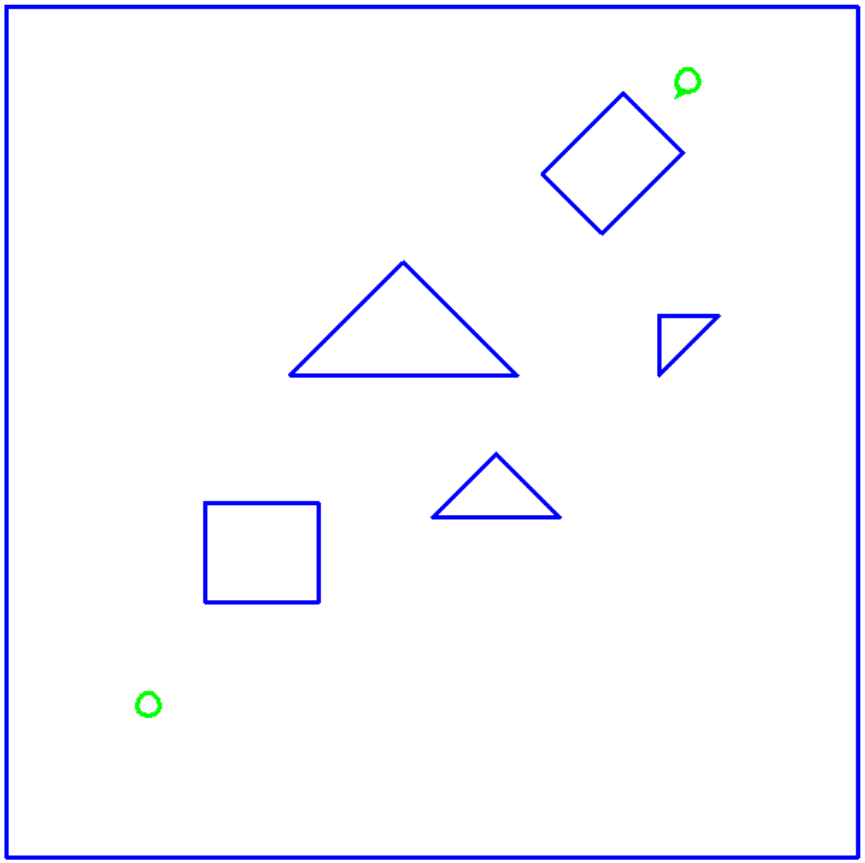
Visibility Graph - VGRAPH

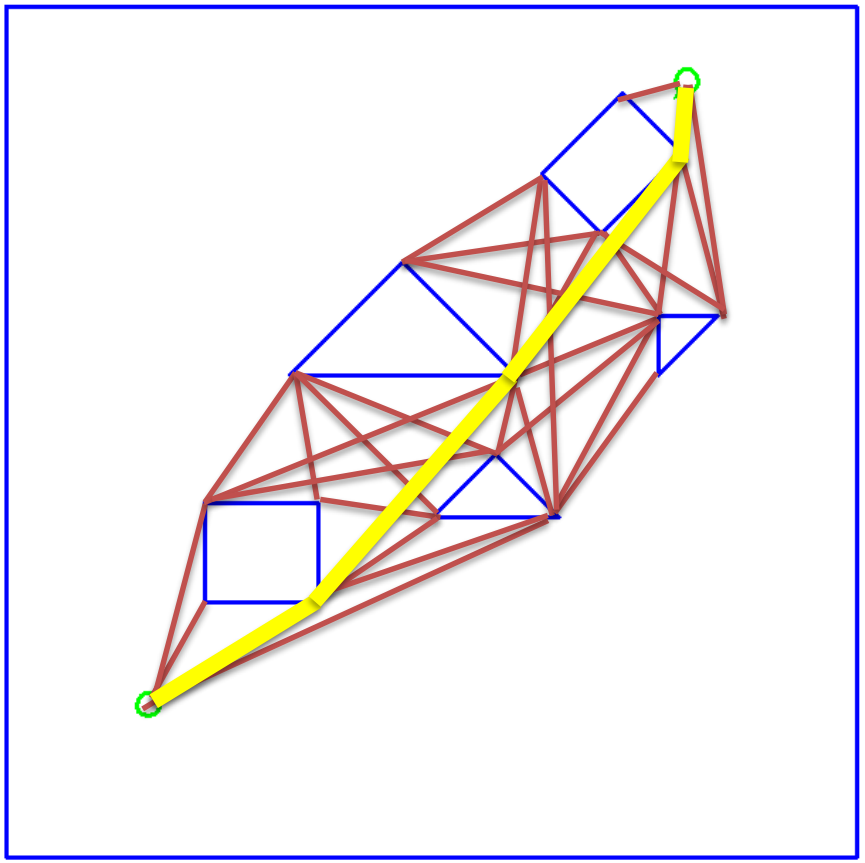




- Start, goal, vertices of obstacles are graph nodes
- Edges are “visible” connections between nodes, including obstacle edges

Visibility Graph - VGRAPH

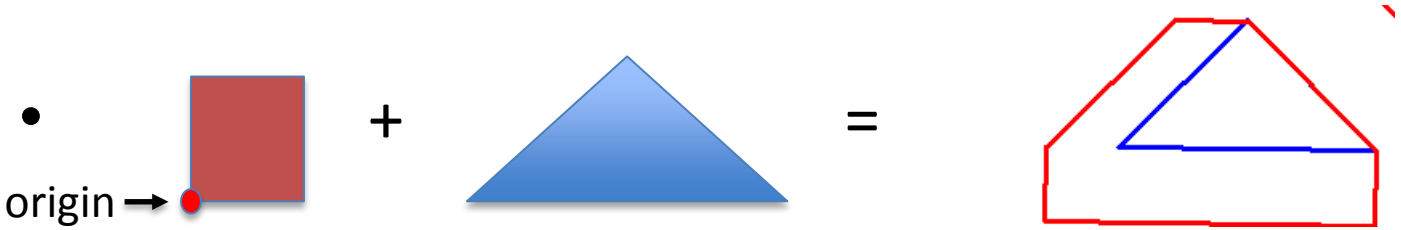




- A* search for shortest path via visible vertices

VGRAPH: Grown Obstacles

- VGRAPH algorithm assumes point robot
- What if robot has mass, size?
- Solution: expand each obstacle by size of the robot – create Grown Obstacle Set



robot obstacle grown obstacle

- This effectively "shrinks" the robot back to a point
- Graph search of the VGRAPH will now find shortest path if one exists using grown obstacle set

CS4733 Class Notes

1 2-D Robot Motion Planning Algorithm Using Grown Obstacles

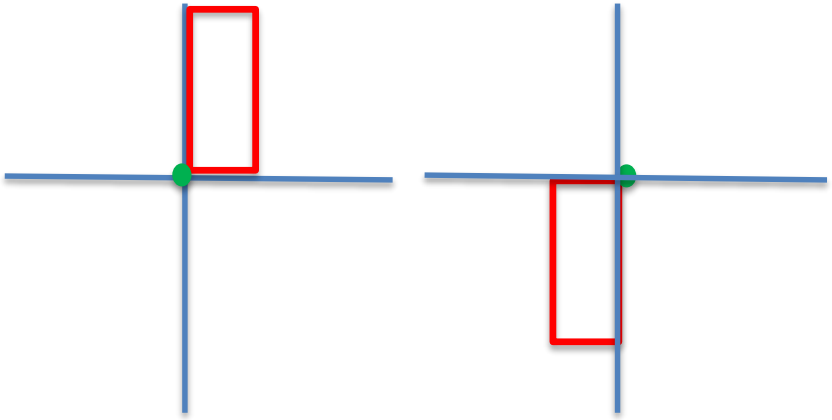
- Reference: *An Algorithm for Planning Collision Free Paths Among Poyhedral Obstacles* by T. Lozano-Perez and M. Wesley.
- This method of 2-D motion planning assumes a set of 2-D convex polygonal obstacles and a 2-D convex polygonal mobile robot.
- The general idea is grow the obstacles by the size 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

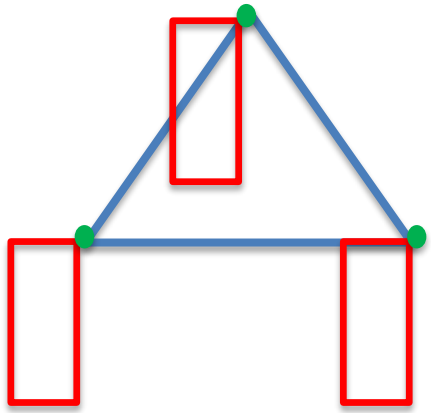
- Method I: Grow each obstacle in the scene by the size of the mobile robot. This is done by finding a set of vertices that determine the grown obstacle (see figure 1). First, we reflect the robot about its X and Y axes. Placing this reflected object at each obstacle vertex, we can map the robot reference 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/)

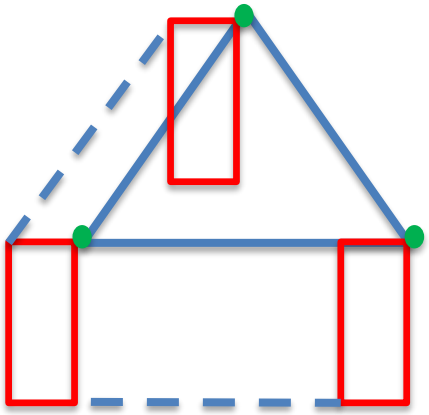
VGRAPH: Growing Obstacles

Reflect robot about X, Y axes

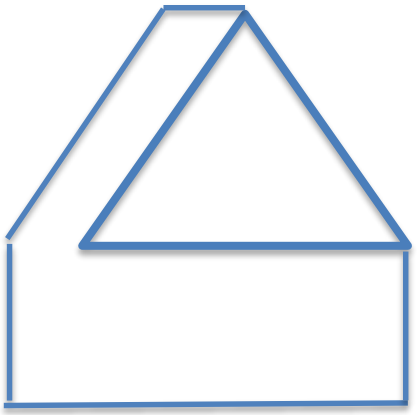


Add reflected robot vertices to each obstacle vertex



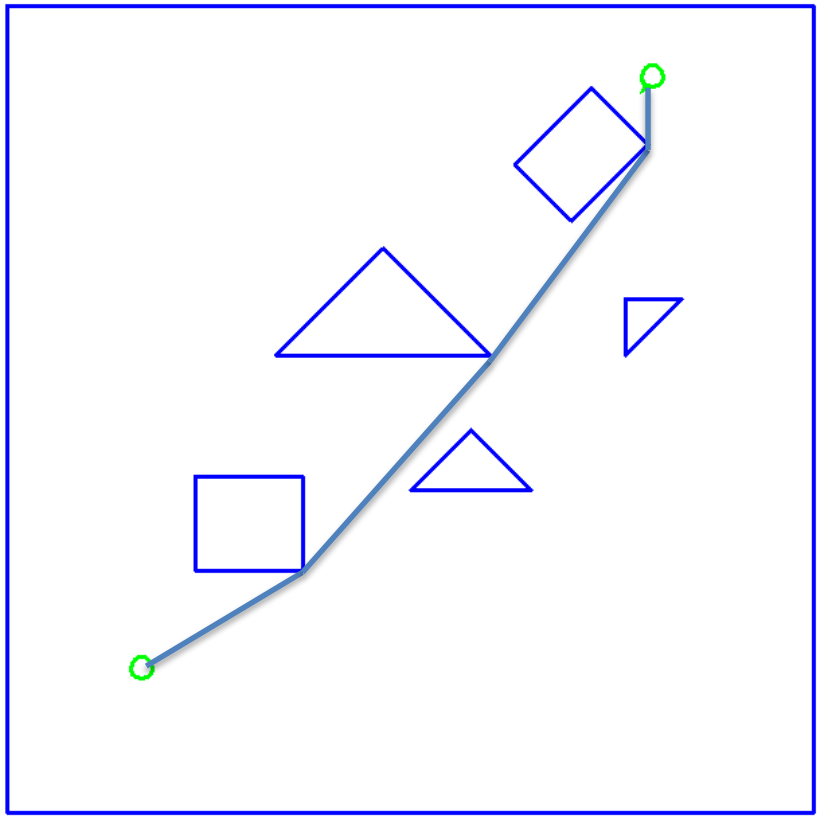


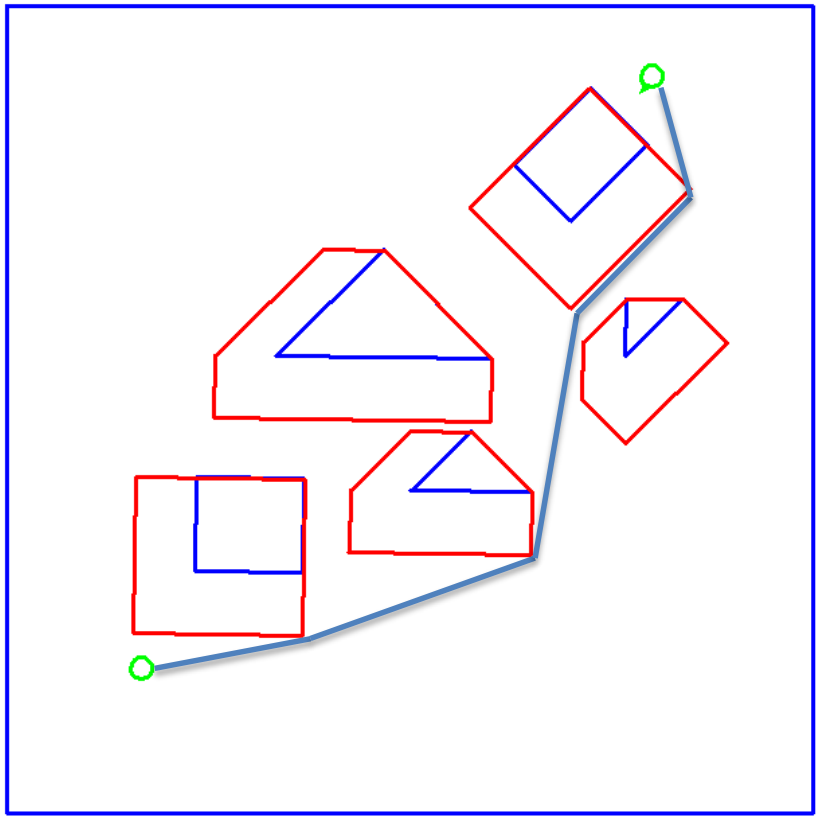
Compute convex hull of vertices



Convex hull is grown obstacle

VGRAPH: Grown Obstacles





Point Robot Path

Path after growing obstacles with square robot

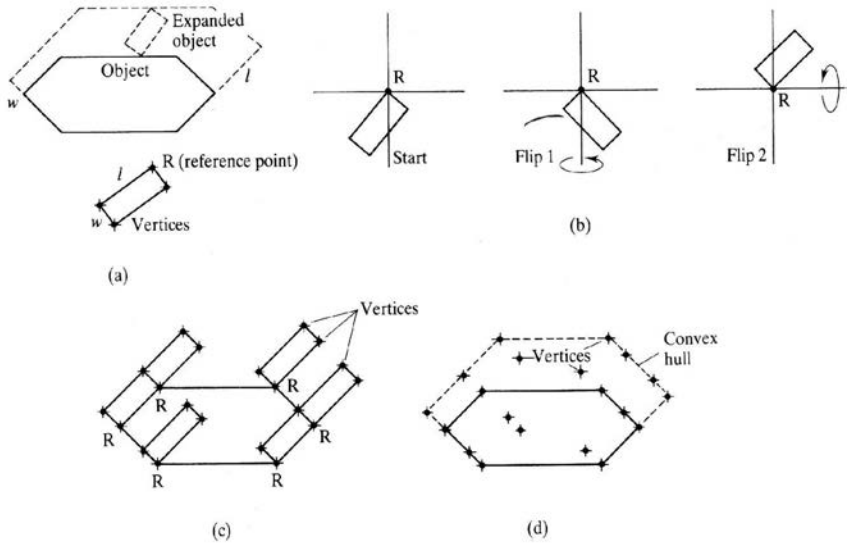


Figure 1: Reflection method for computing grown obstacles (from P. McKerrow, *Introduction to Robotics*).

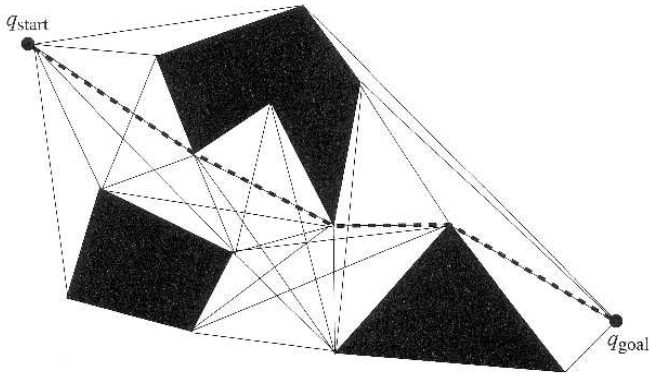


Figure 2: Visibility graph with edges.

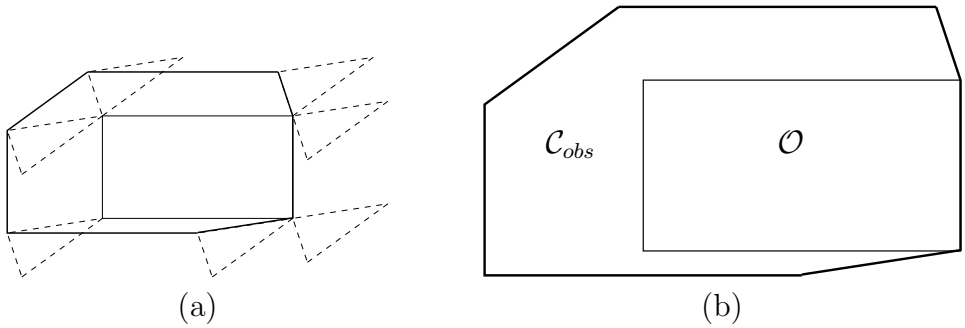


Figure 4.14: (a) Slide the robot around the obstacle while keeping them both in contact. (b) The edges traced out by the origin of $\mathcal{A}$ form $\mathcal{C}_{obs}$.

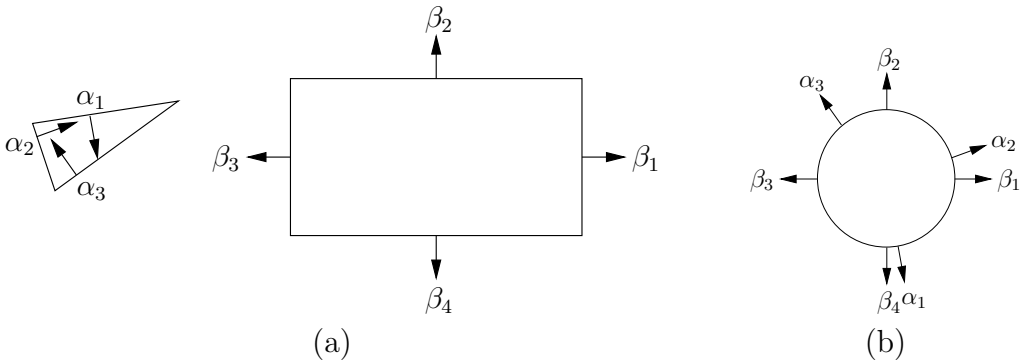
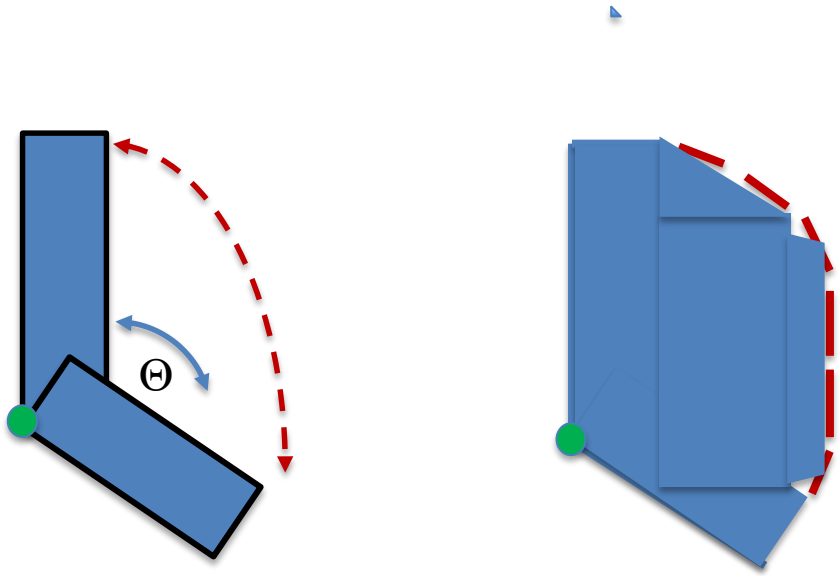


Figure 4.15: (a) Take the inward edge normals of $\mathcal{A}$ and the outward edge normals of $\mathcal{O}$. (b) Sort the edge normals around $\mathbb{S}^1$. This gives the order of edges in $\mathcal{C}_{obs}$.

VGRAPH Extensions

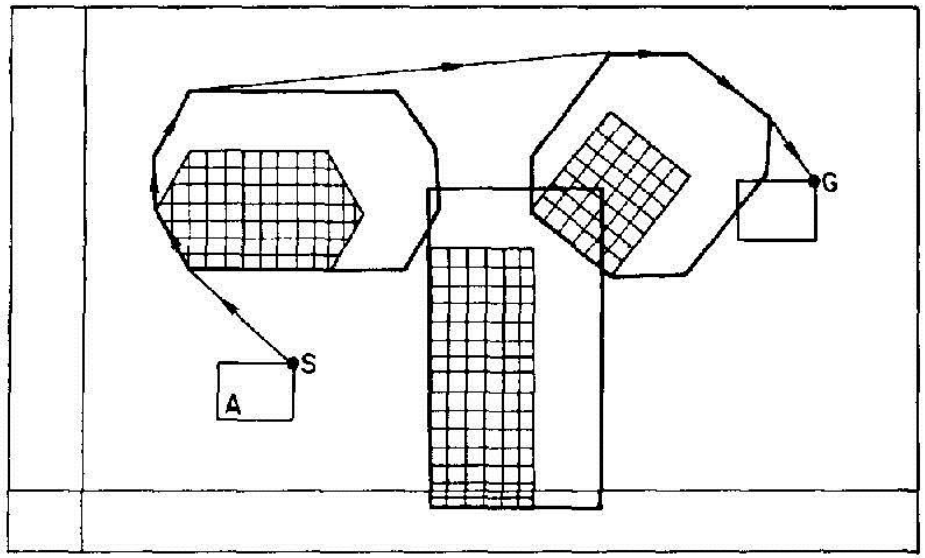
- Rotation: Mobile Robot can rotate
- Solution:
  - Grow obstacles by size that includes all rotations
  - Over-conservative. Some paths will be missed
  - Create multiple VGRAPHS for different rotations
  - Find regions in graphs where rotation is safe, then move from one VGRAPH mapping to another
- Non-convex obstacles/robots: any concave polygon can be modeled as set of convex polygons

VGRAPH: Rotations



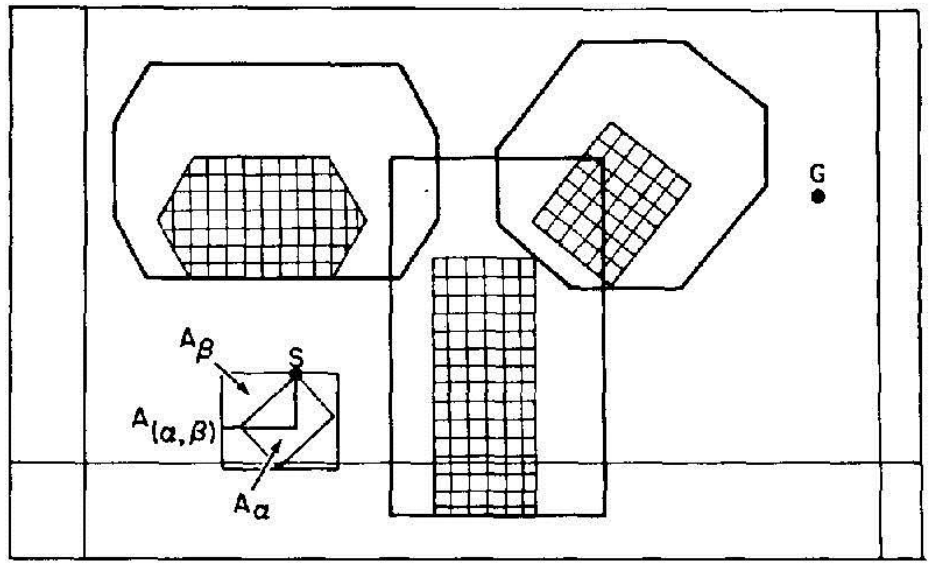
Left: Rectangular robot that can rotate
Right: Polygon that approximates all rotations
Polygon is over-conservative, will miss legal paths

Fig. 5.



Path for grown obstacles with fixed robot orientation

Fig. 6.



Path for grown obstacles with robot rotation

Growing Non-Convex robots

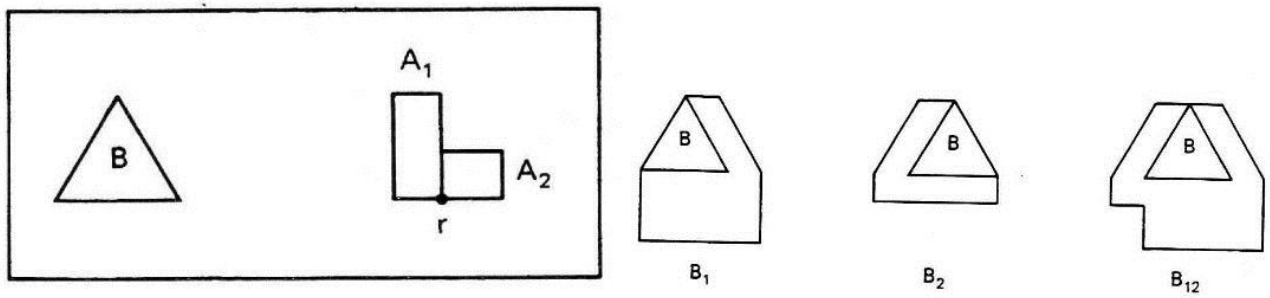
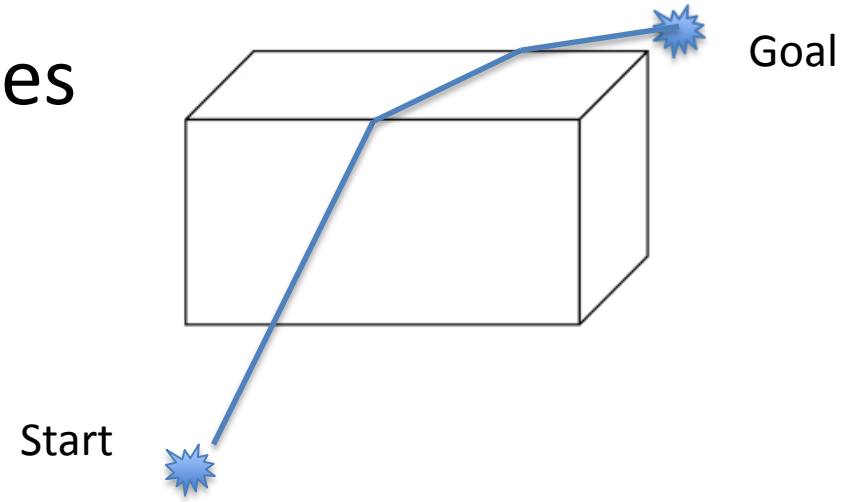


Figure 3: Concave objects. Decompose concave robot A into convex regions, Compute grown space of each convex region with obstacle B, union the resulting grown spaces (from R. Schilling, *Fundamentals of Robotics*).

VGRAPH Summary

- Guaranteed to give shortest path in 2D
- Path is dangerously close to obstacles – no room for error
- Does not scale well to 3D. Shortest path in 3D is not via vertices:
- Growing obstacles is difficult in 3D



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

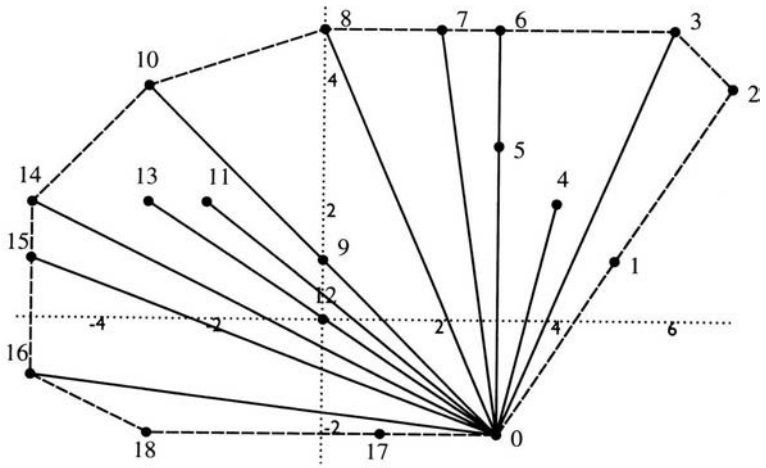
- Reference: *Computational Geometry in C* by J. O'Rourke
- Given a set of points S in a plane, we can compute the convex hull of the point set. The convex hull is an enclosing polygon in which every point in S is in the interior or on the boundary of the polygon.
- An intuitve definition is to pound 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.



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*