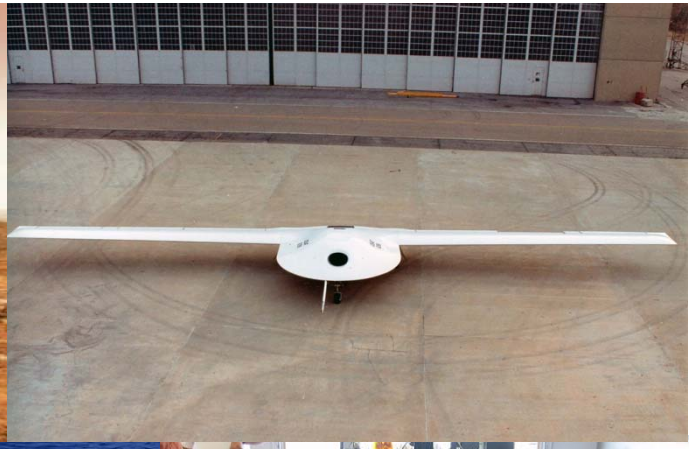
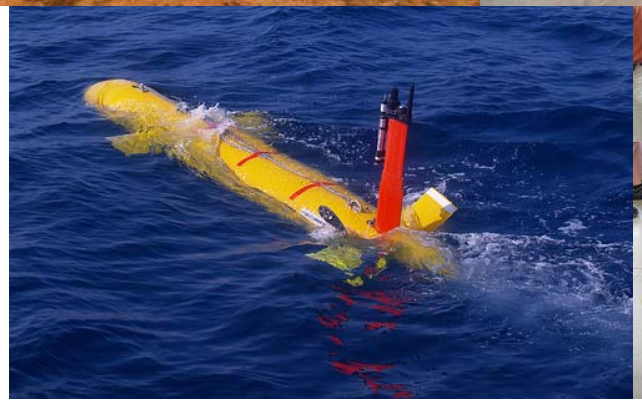
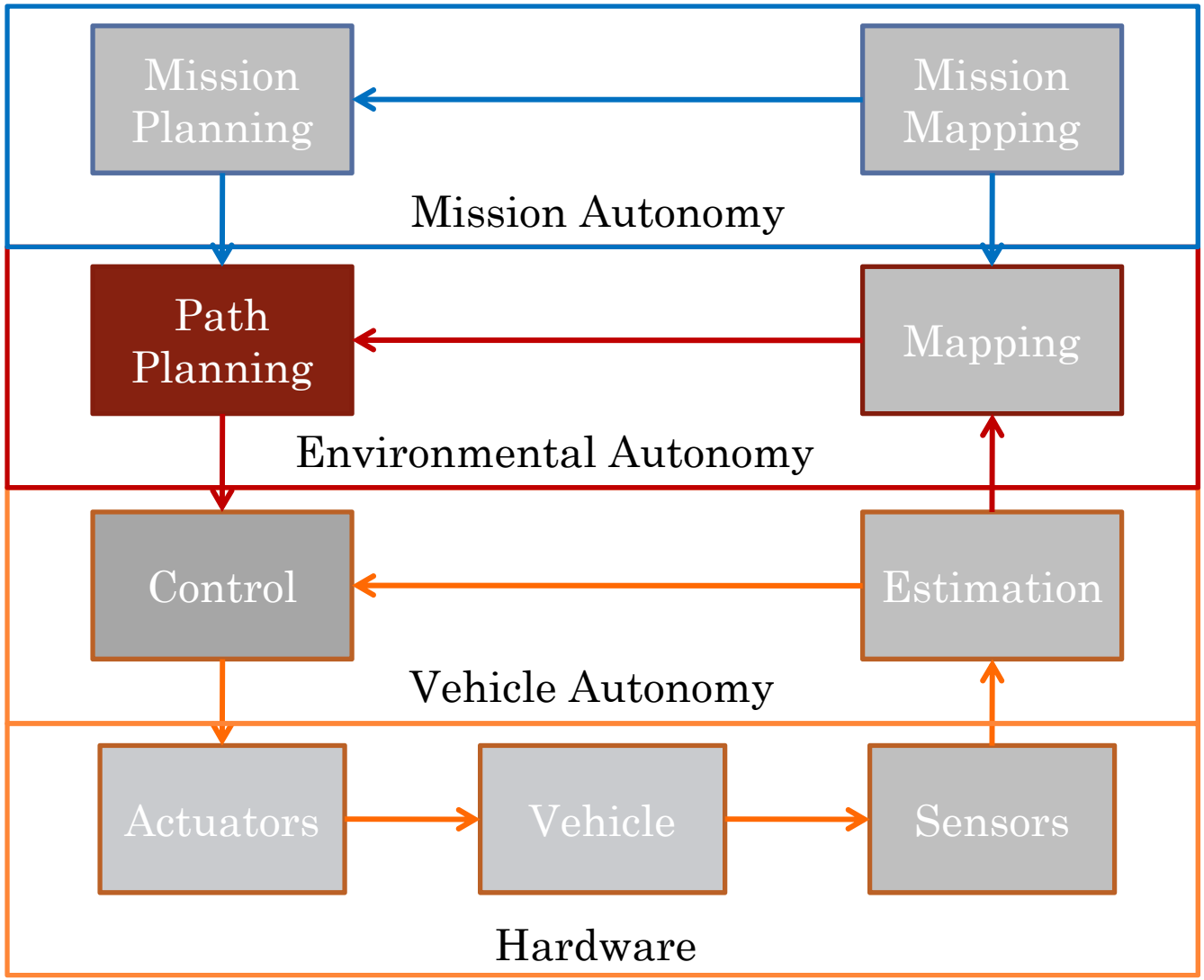



ME 597: Autonomous Mobile Robotics
Section 8 – Planning I

Prof. Steven Waslander

Components


Mission Planning
Mission Mapping
Mission Autonomy
Path Planning
Mapping
Environmental Autonomy
Control
Estimation
Vehicle Autonomy
Actuators
Vehicle
Sensors
Hardware

Outline

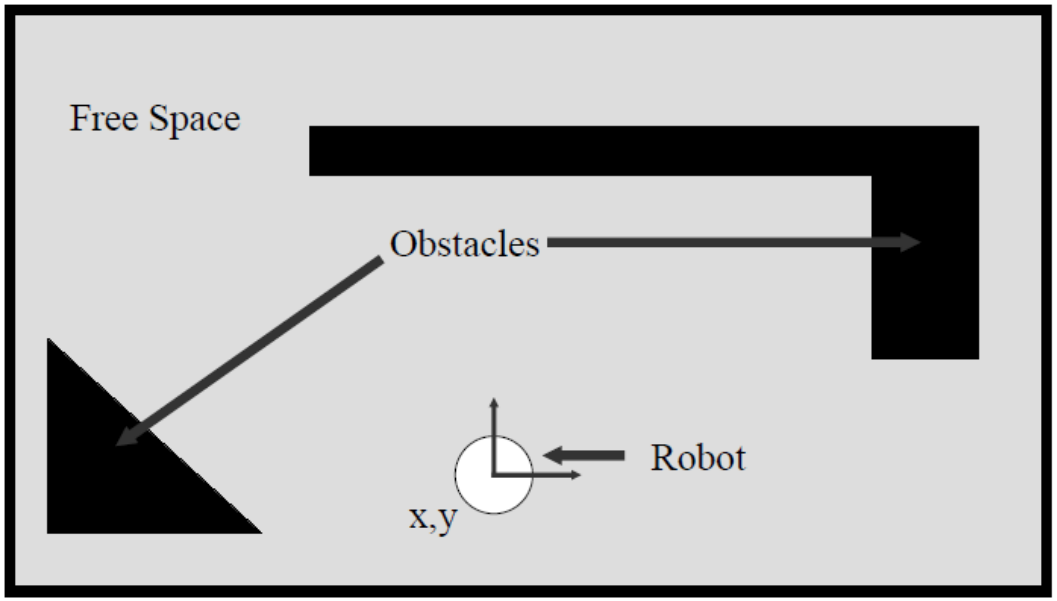
- Planning Concepts
- Reactive Motion Planning Algorithms
  - Bug
  - Potential Fields
  - Trajectory Rollout
- Graph Based Motion Planning
  - Finding paths on graphs
    - Depth First, Breadth First, Wavefront
    - Dijkstra, A*
  - Generating Graphs from environments
    - Visibility Graphs
    - Decompositions

Outline

- Probabilistic Graph Based Planning
  - Complex Planning Examples
  - Probabilistic Roadmaps
  - PRM Algorithm
  - Collision Detection
  - Sampling Strategies
  - RRT Algorithm
- Optimization Based Planning
  - Linear Programming
  - Nonlinear Programming

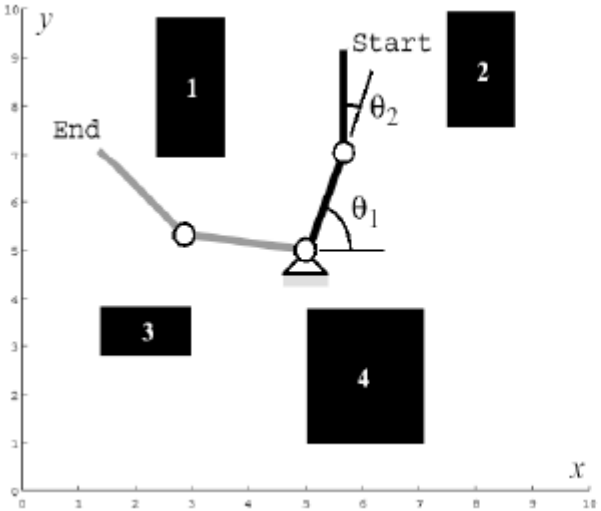
Planning

- Motion Planning Terminology
  - Work space
    - The environment the vehicle finds itself in
    - Comes from industrial robotics
    - 2-3D physical world
    - Can be defined in a number of ways
      - Polygons, Surfaces, Occupancy grids

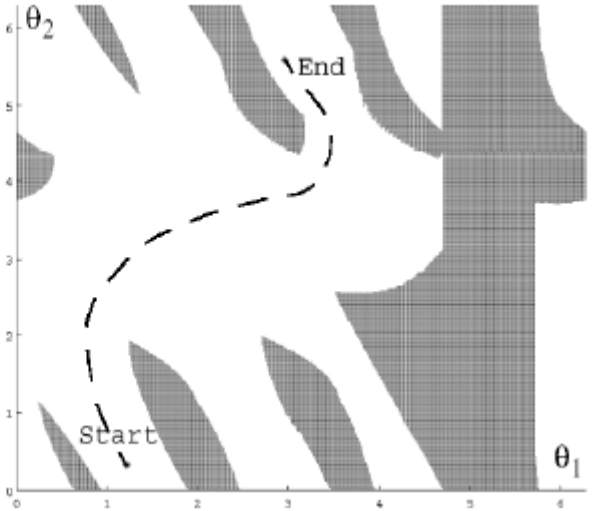


Planning

- Motion Planning Terminology
  - Configuration Space
    - Complete planning space of robot
    - For two linkage robot, workspace is 2D space of joint angles, minus black areas which are positions blocked by obstacles
    - Configuration space is much different, defined by allowable states in white, unallowable in grey



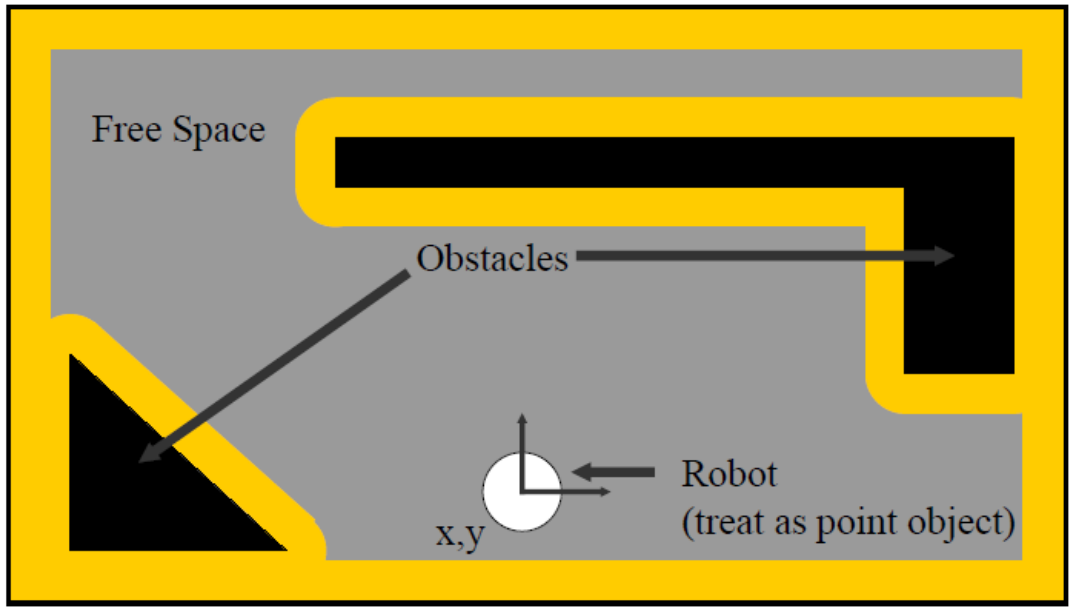
Work space



Configuration space

Planning

- Motion Planning Terminology
  - Configuration space for a two wheeled non-point robot



  - Can be insufficient to simply expand the obstacle
    - Can find x,y path but must also identify heading to travel in
    - Constraints on velocity not represented here

Planning

- Objectives
  - Predefined target configuration
    - Guaranteed to find a path
    - Minimum distance
    - Minimum time
    - Minimum cost (drivability, risk)
  - Coverage/Search
    - Explore/monitor an area by visiting all locations
      - At least once
      - Exactly once
      - Minimizing time between visits etc.

Planning

- Constraints
  - Occupancy
    - Obstacles defined by geometric representation
    - State of vehicle cannot violate obstacle regions
    - Included in definition of work space, configuration space
  - Dynamics
    - Holonomic vs Nonholonomic
      - When motion constraints involve vehicle velocities, the system is considered nonholonomic
        - Much harder planning problem
        - Two wheeled robot a classic example

Planning

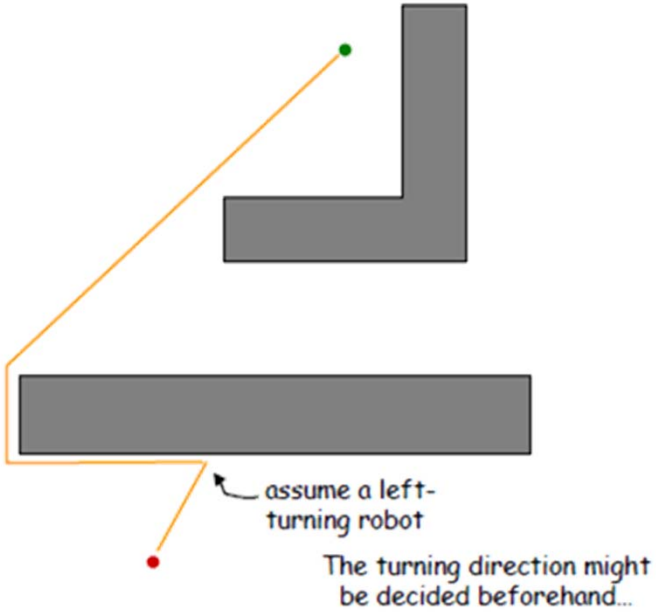
- Approaches
  - Reactive – local approach
    - Decide a direction to go in based on goal and obstacles
    - Ignores vehicle dynamics
    - Usually deterministic formulation
  - Graph-based – global approach
    - Graph extracted from workspace definition
    - Graph generated by random sampling of nodes and random connections between nodes
  - Optimal – global approach
    - Find complete path to goal
    - Incorporate constraints
      - May need to model a certain way
        - Graph representation of environment
        - Linear, nonlinear, mixed integer-linear
        - Probabilistic representation of configuration space (soft constraints)

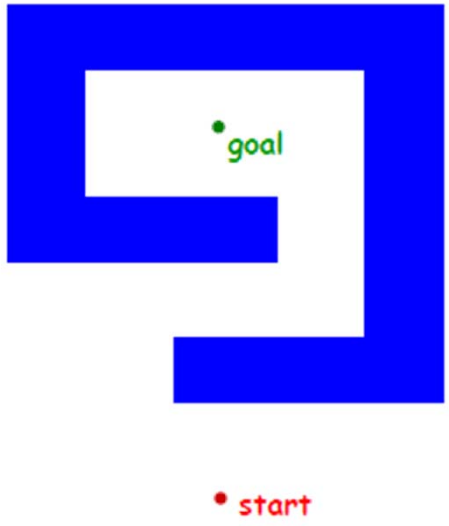
Bug Algorithms

- Reactive - Bug Algorithms
  - Simplest form of path planning from implementation point of view
    - Assume very little knowledge of environment or robot state
  - Define a set of rules, prove reachability of goal
- Bug 0, 1, 2, Tangent Bug ….
  - Demonstrate how hard it is to find way around 2D environment even if optimality is of no concern
    - Require as little storage and sensing as possible

Bug Algorithms

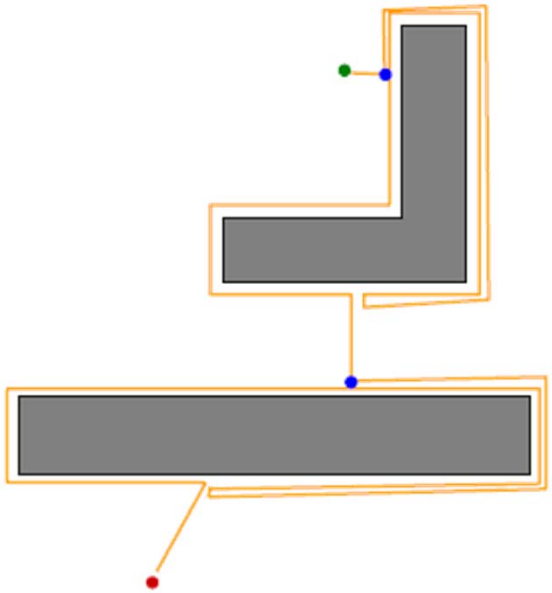
- Bug 0: Known goal and robot locations, can follow obstacle boundary
  - Always head directly to goal
  - If blocked, turn and follow obstacle until you can head directly to goal again
  - Doesn't always work





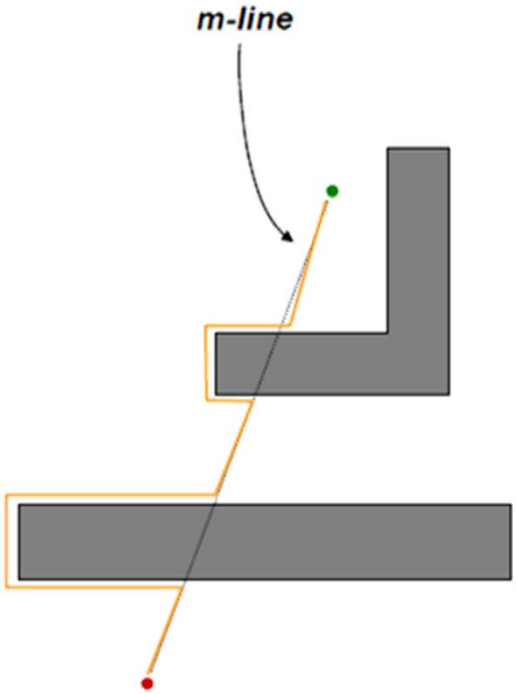
Bug Algorithms

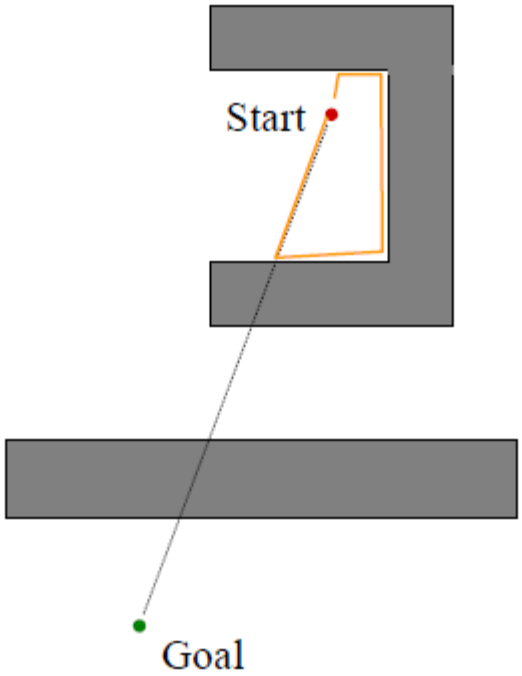
- Bug 1: Known location or robot and goal, can follow obstacle boundary
  - Head directly toward goal
  - When blocked, circumvent obstacle, remember closest point
  - Return to closest point and continue to goal
  - Guaranteed arrival
  - Can be slow



Bug Algorithms

- Bug 2: Known location and goal, can follow obstacle boundary
  - Head toward goal, track start-goal line (m-line)
  - When blocked, circumvent obstacle until m-line
    - Try both directions if necessary
  - Continue to goal

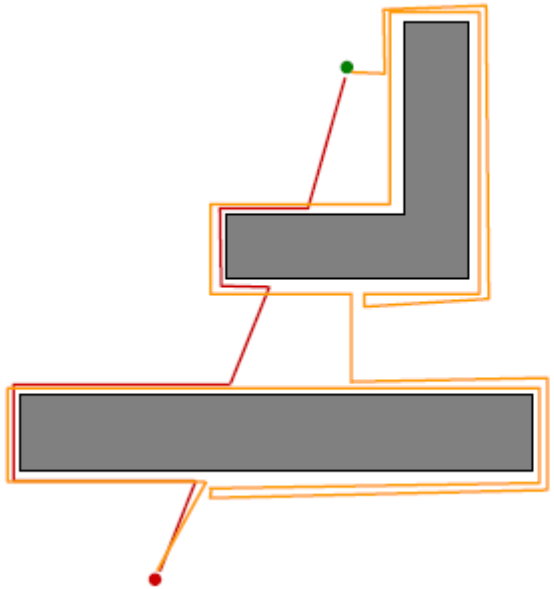




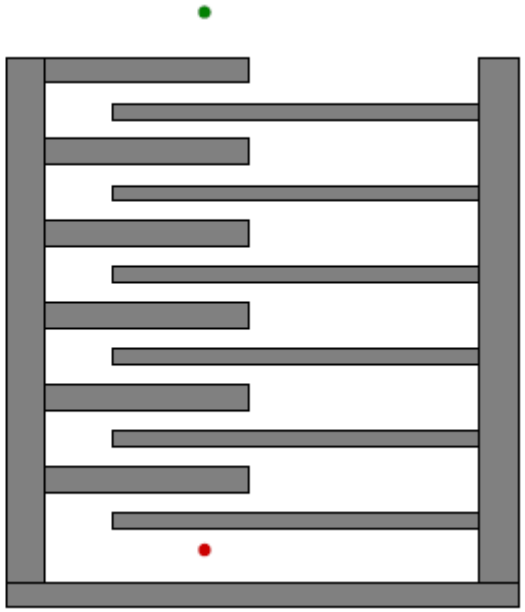
Bug Algorithms

- Bugs Comparison

Bug 2 beats Bug 1



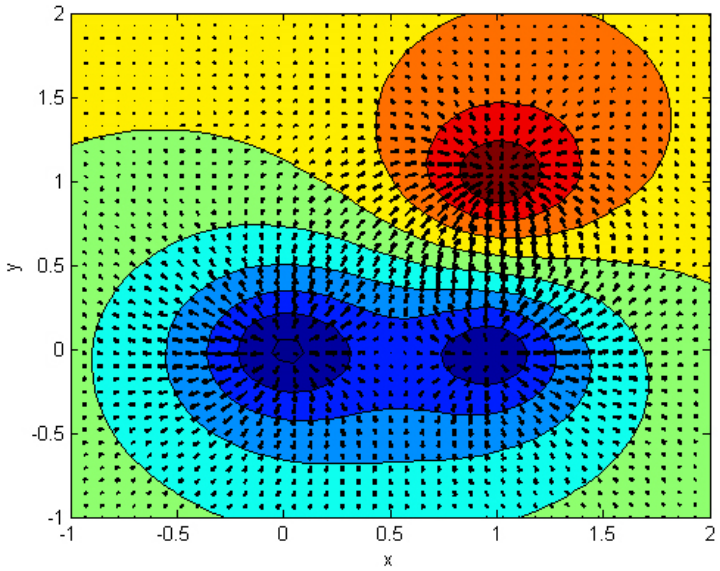
Bug 1 beats Bug 2



No clear winner, we need something more sophisticated

Potential Fields

- Potential Fields [Khatib, 1986]
  - A simple type of navigation function
    - A function that describes a direction of travel everywhere in the environment
  - Defines a potential field at every point in map
  - Robot descends potential field by moving in direction of negative gradient

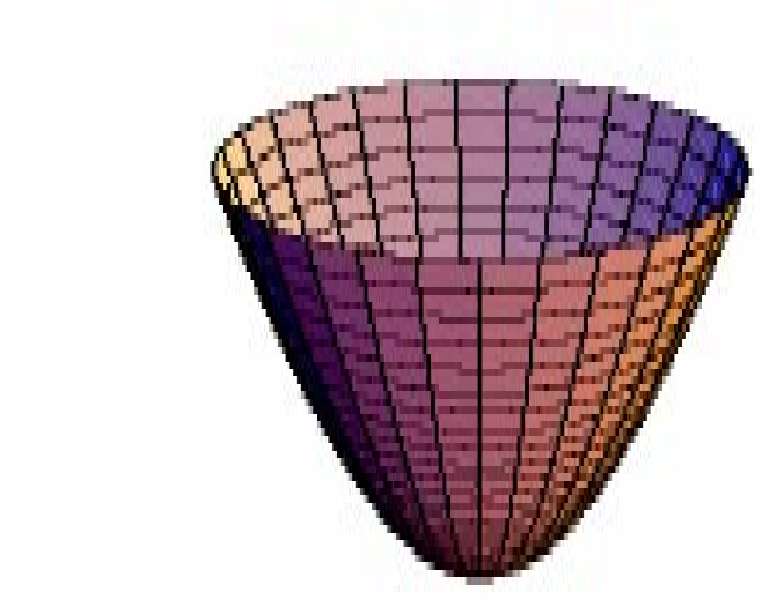


Potential Fields

- Potential Field Target function
  - Target attracts the vehicle
    - Distance (ρ) between vehicle, q, and target, q^g

$$V_{att}(q) = K_{att}\rho(q, q^g)^2$$

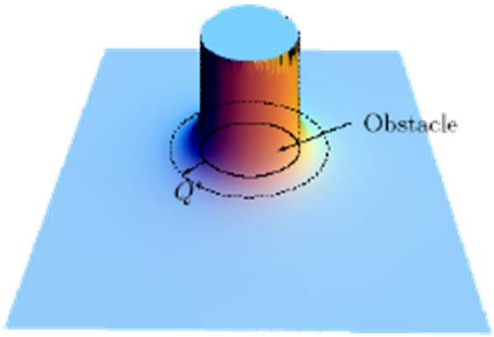
  - Usually quadratic, can be anything



Potential Fields

- Potential Fields
  - Obstacles repel the vehicle
    - Strength based on shortest distance to obstacle O^i

$$V_{rep}(q) = K_{rep} \sum_{i=1}^{n} \frac{1}{\rho(q, O^i)^2}$$



    - Often a maximum distance of influence is included

$$V_{rep}(q) = K_{rep} \sum_{i=1}^{n} \begin{cases} \left(\frac{1}{\rho(q, O^i)} - \frac{1}{\bar{\rho}} \right)^2 & \rho(q, O^i) < \bar{\rho} \\ 0 & \text{otherwise} \end{cases}$$

Potential Fields

- Distance to obstacle function
  - Minimum of the distances to every point on the boundary of the obstacle

$$\rho(q, O^i) = \min_{c \in \delta O^i} \rho(q, c) = \min_{c \in \delta O^i} \left((q-c)^T (q-c) \right)^{1/2}$$

  - Gradient for distance to obstacle

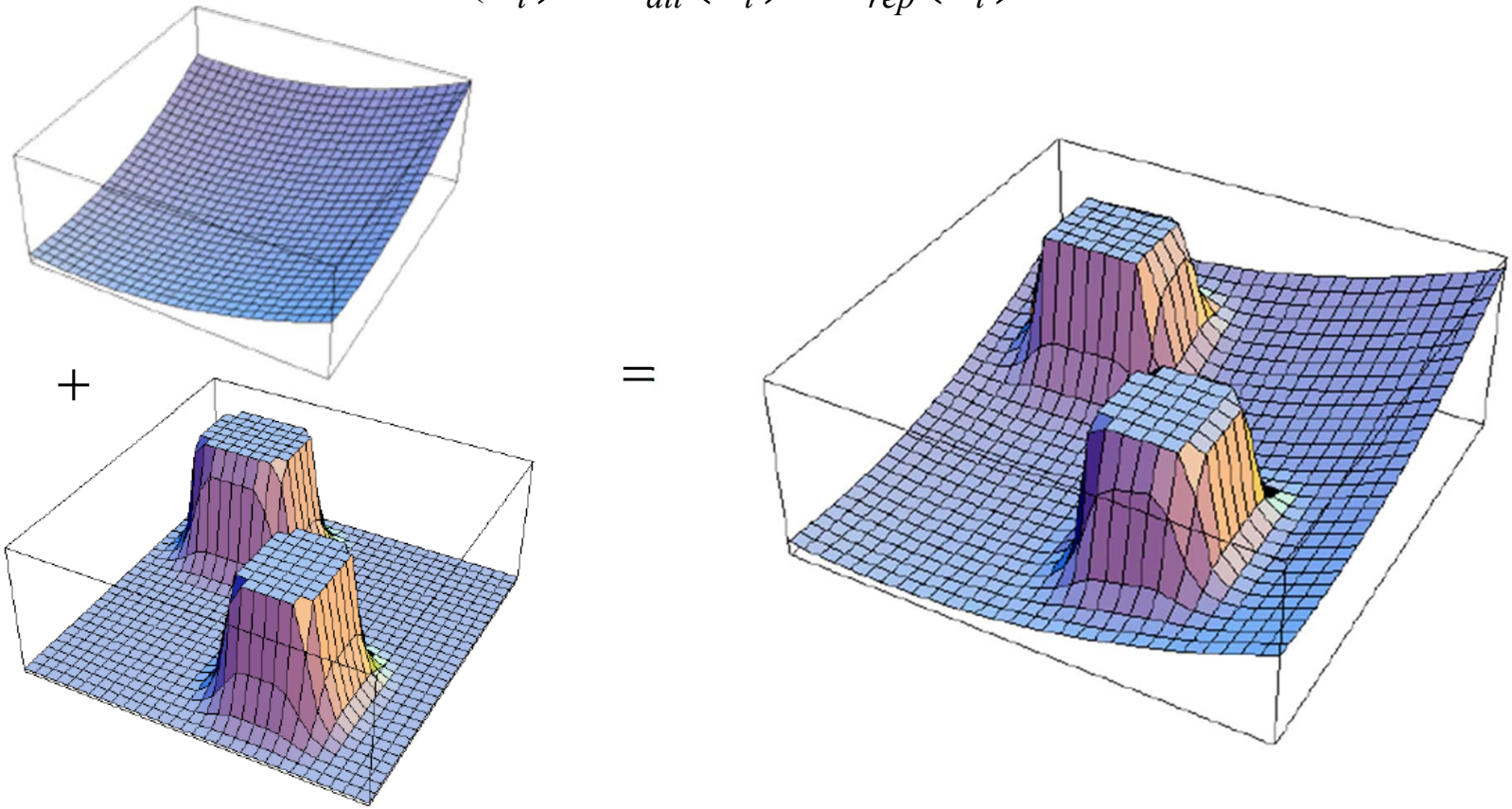
$$\nabla \rho(q, c^*) = \frac{\left(q - c^*\right)}{\rho(q, c^*)}$$

    - Must find closest point to evaluate either

Potential Fields

- Potential Fields
  - Potential field is combination of the two fields

$$V(x_t) = V_{att}(x_t) + V_{rep}(x_t)$$

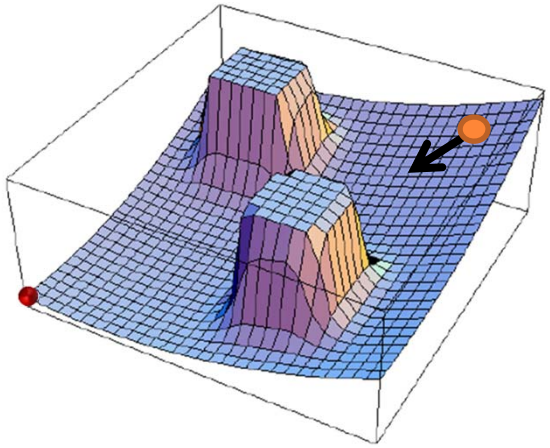


Potential Fields

- Potential Fields
  - Motion should then proceed in the direction of steepest descent of the potential

$$-\nabla V$$

$$= -\left(2K_{att}(q - q^g) - \sum_{i=1}^{n} \max\left(0, 2K_{rep}\left(\frac{1}{\rho(q,c^*)} - \frac{1}{\bar{\rho}} \right) \frac{\left(\rho - c^*\right)}{\rho(q,c^*)^3} \right) \right)$$



Potential Fields

- Potential fields
  - Pros
    - Easy to implement
    - Fast to compute online
    - Intuitive
    - Can tailor how close to go to obstacles
  - Cons
    - Not optimal
    - No dynamic constraints considered
    - Local minima
    - Stability

Potential Fields

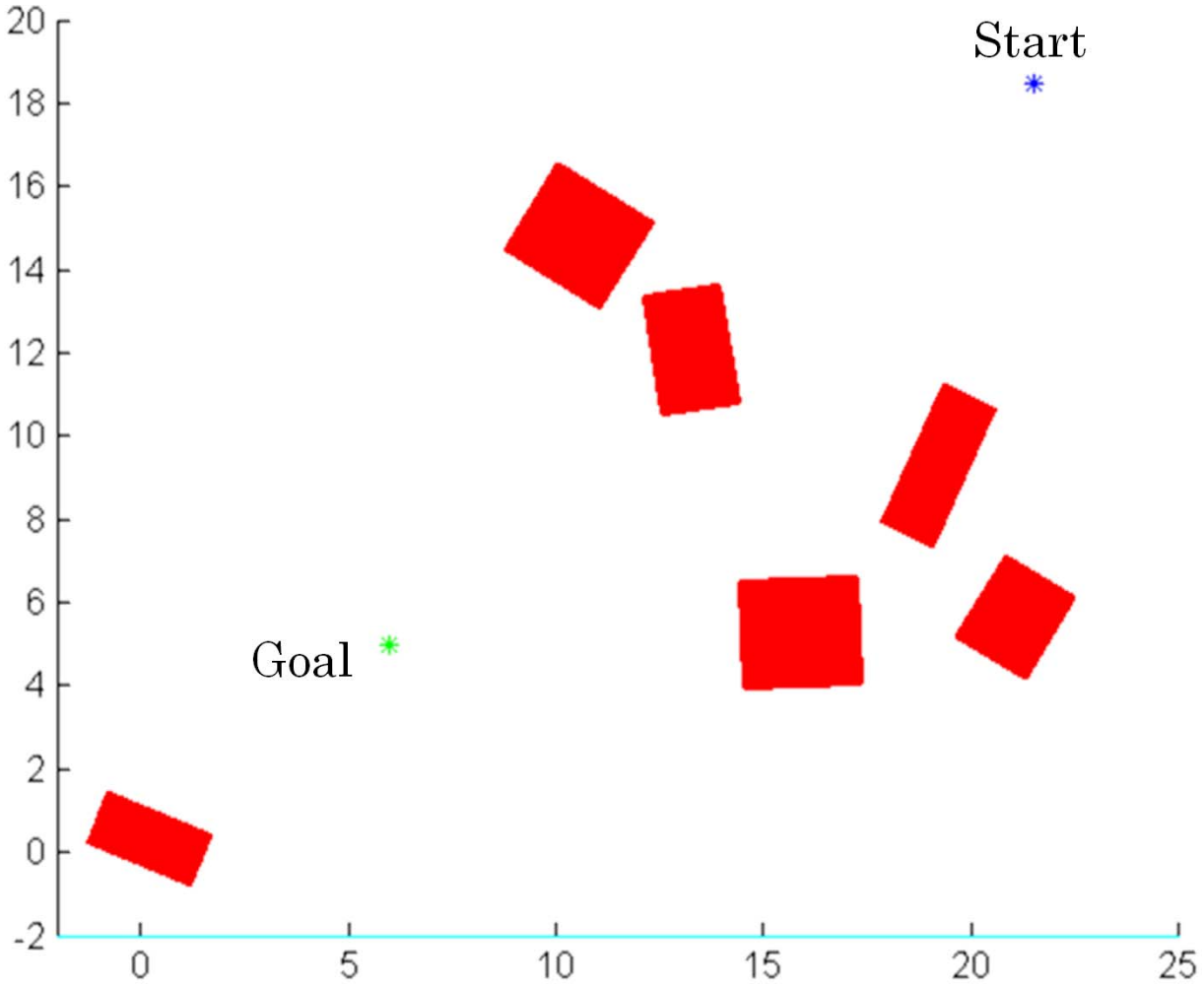
- Potential fields example
  - Hardest part is defining the environment
    - Non overlapping obstacles
  - Define potential field only for plotting
  - Gradient at current location is needed for motion

Potential Fields

- Potential Field Example
  - Robot is assumed to move in direction of steepest descent with speed equal to magnitude of gradient
  - Potential is created from three elements
    - Attractive potential to goal
    - Repulsive potential from closest point on obstacle, up to a range of 0.5 meters
    - Repulsive potential from center of obstacle, up to a range of 4 meters

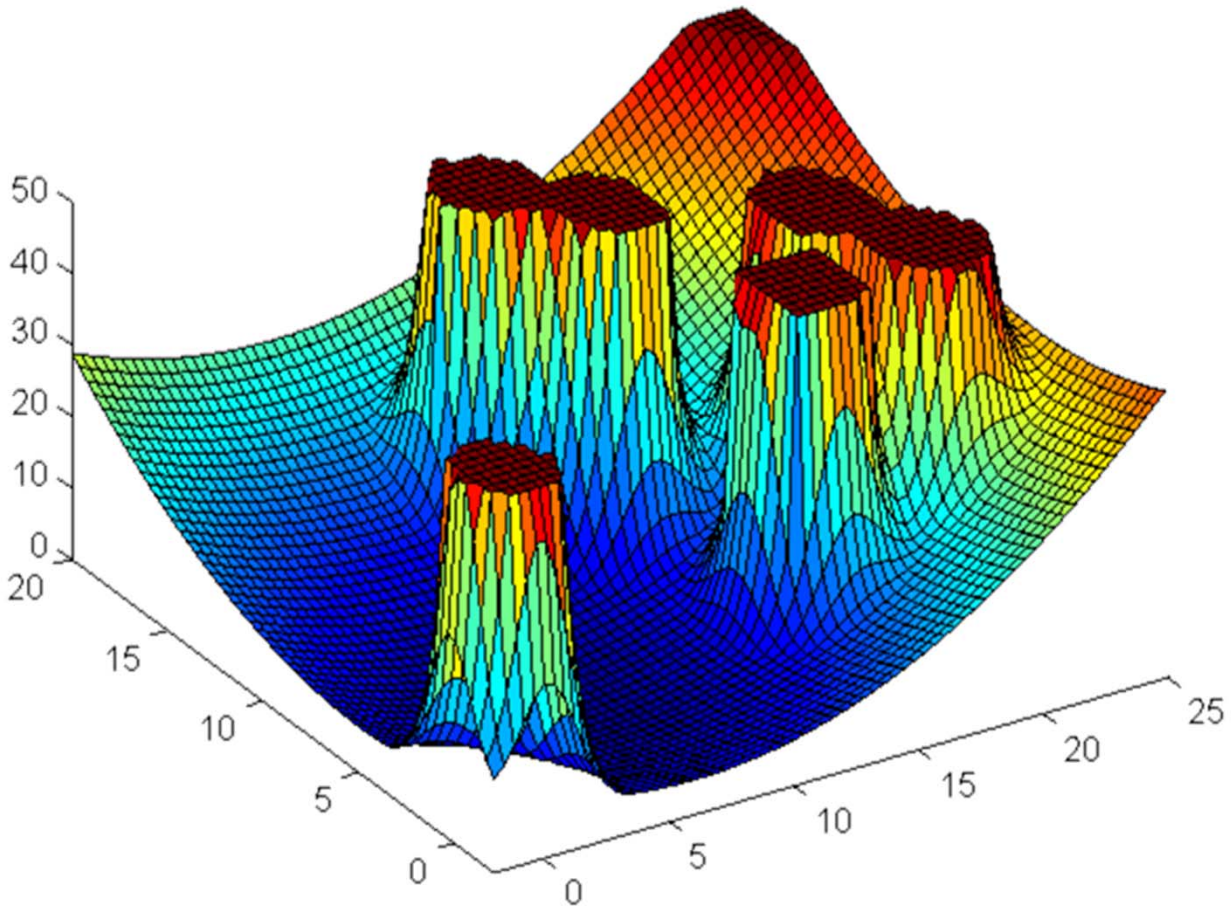
Potential Fields

- The obstacle field



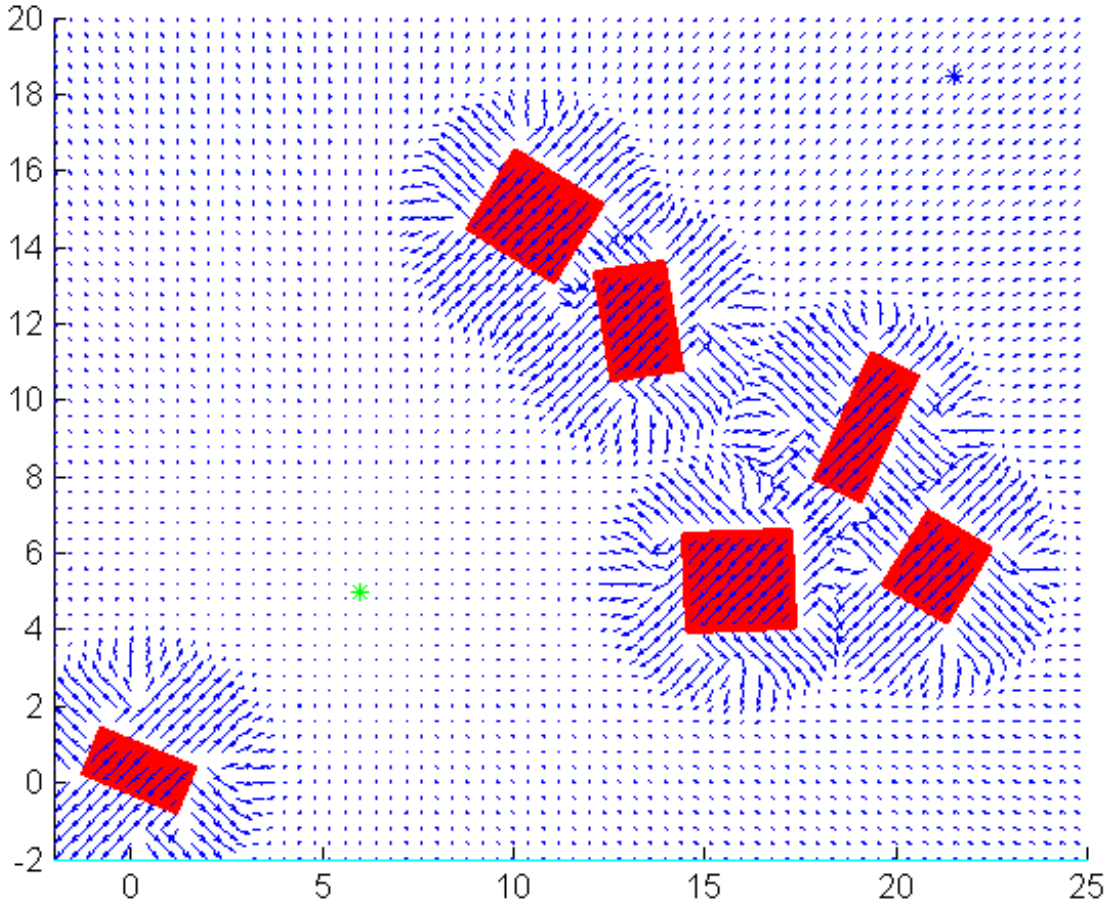
Potential Fields

- Potential fields example
  - The potential field



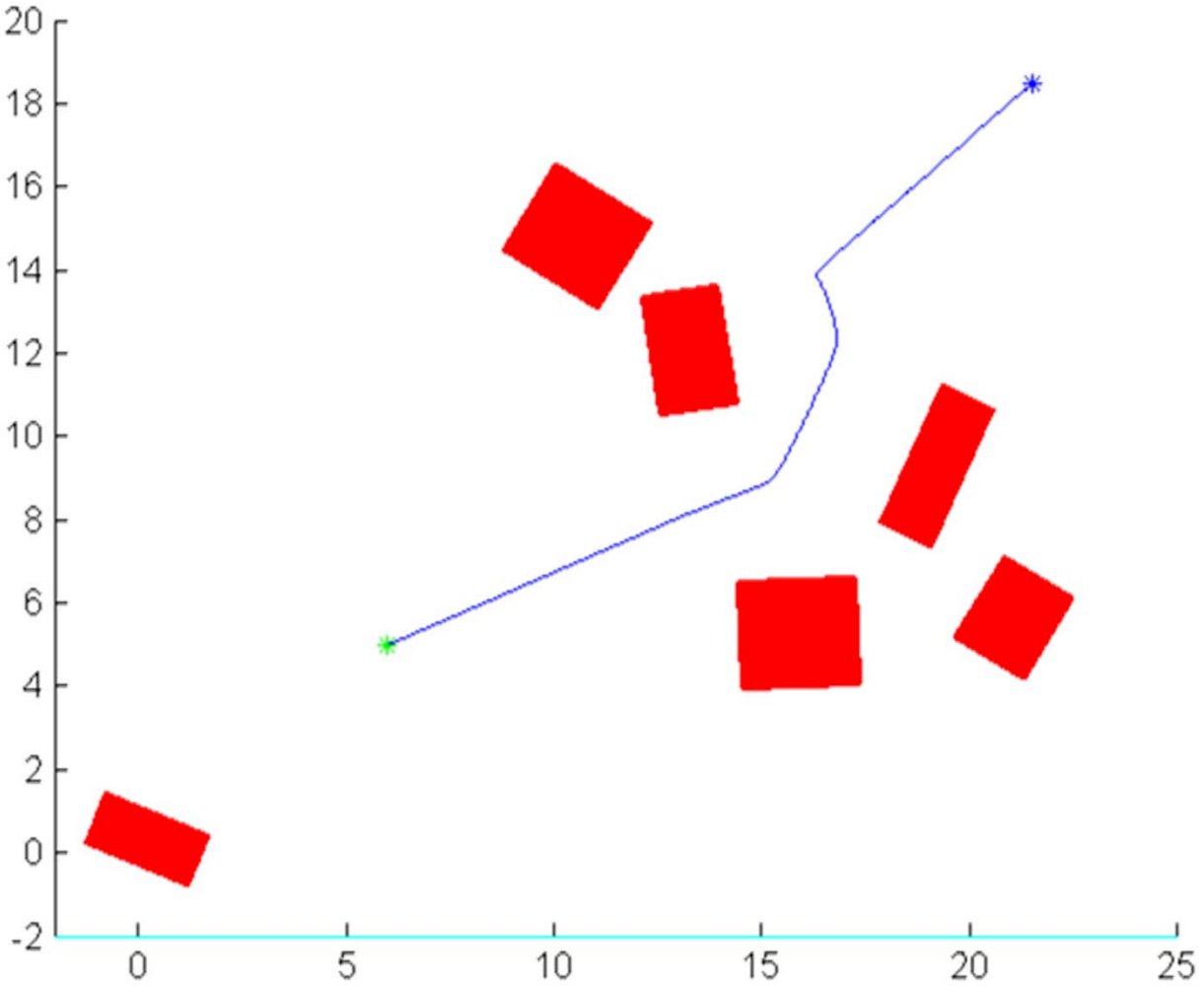
Potential Fields

- Potential fields example
  - Gradient field



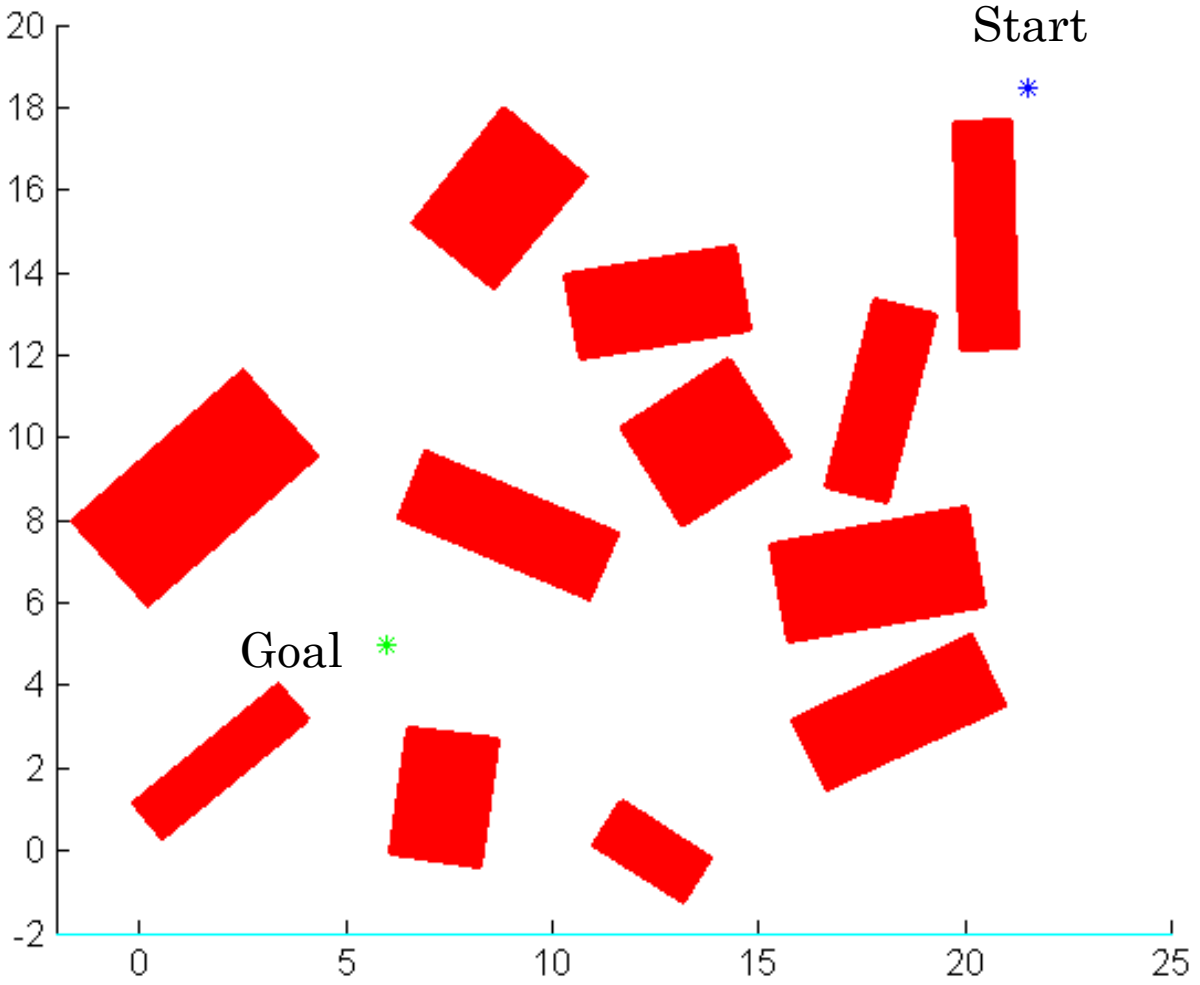
Potential Fields

- Potential fields example
  - The trajectory



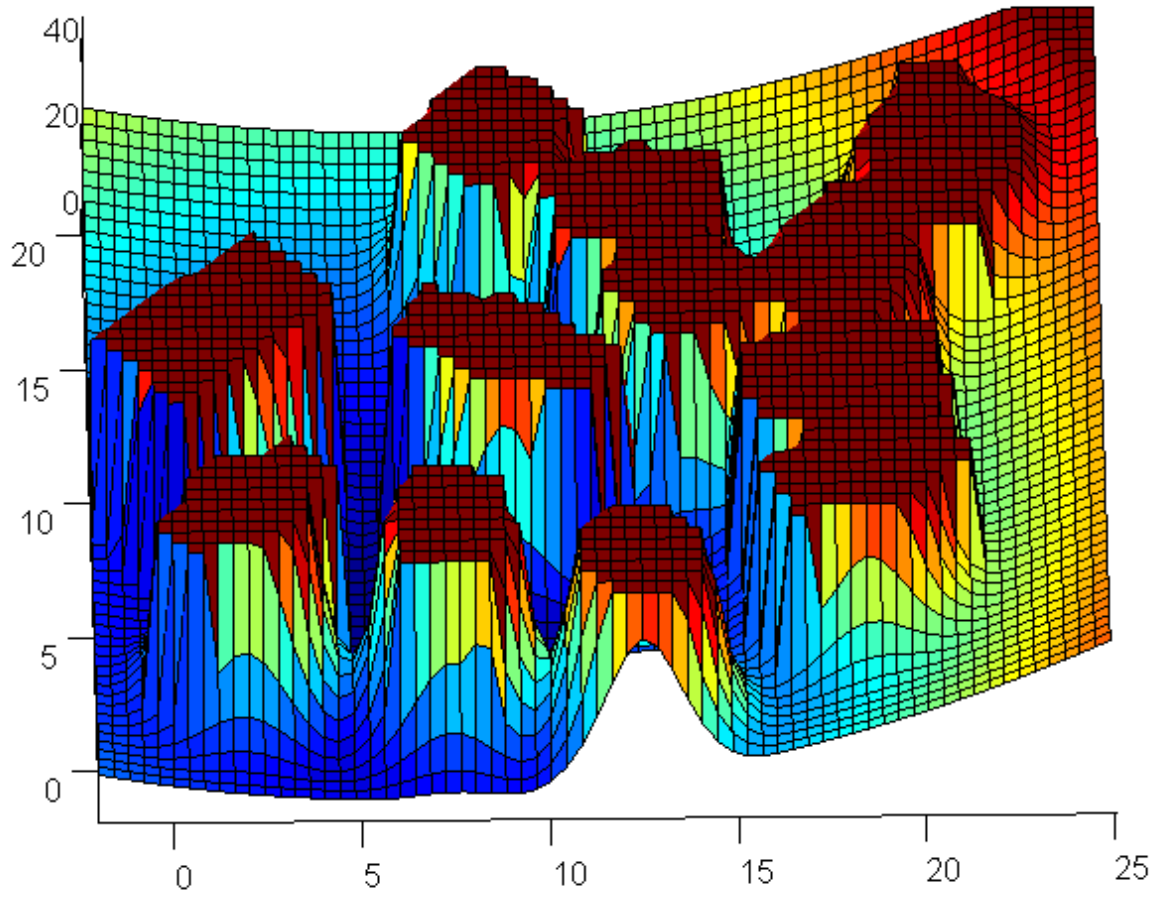
Potential Fields

- The obstacle field



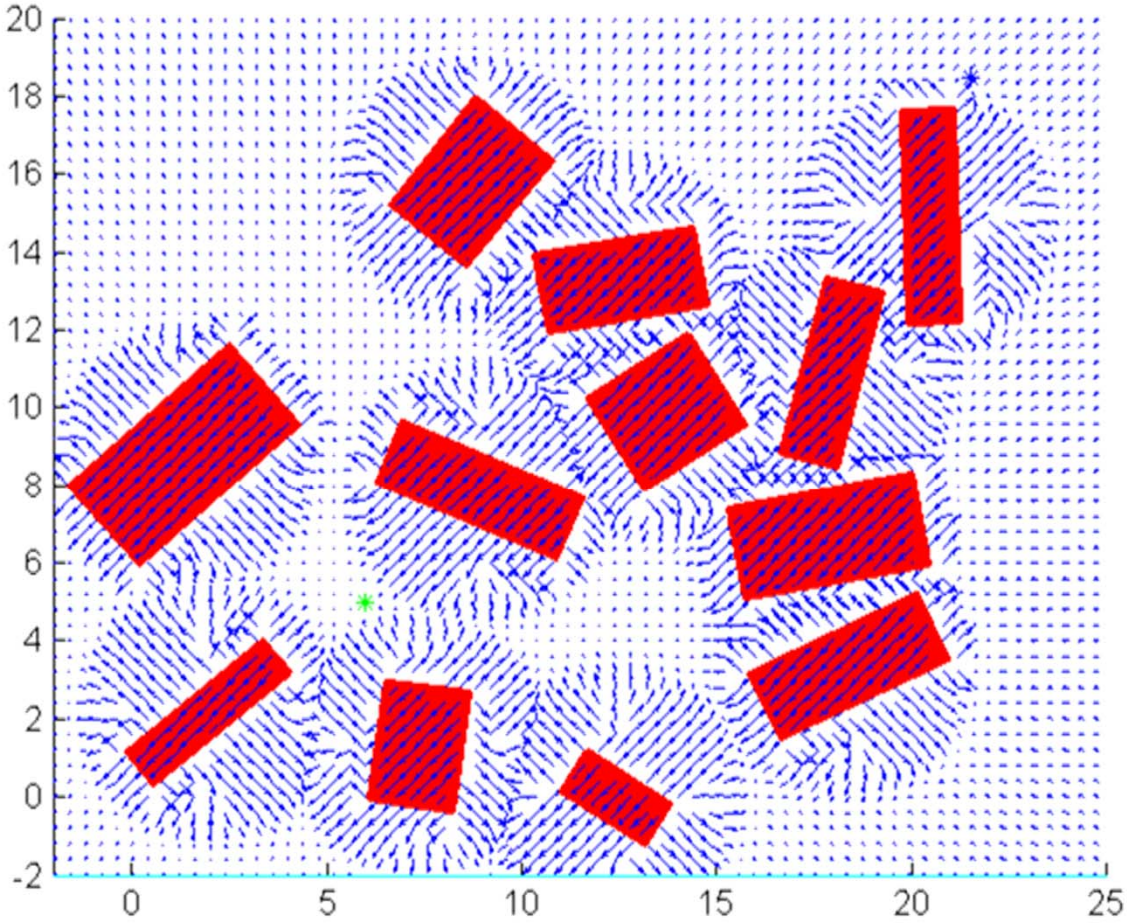
Potential Fields

- Potential fields example
  - The potential field



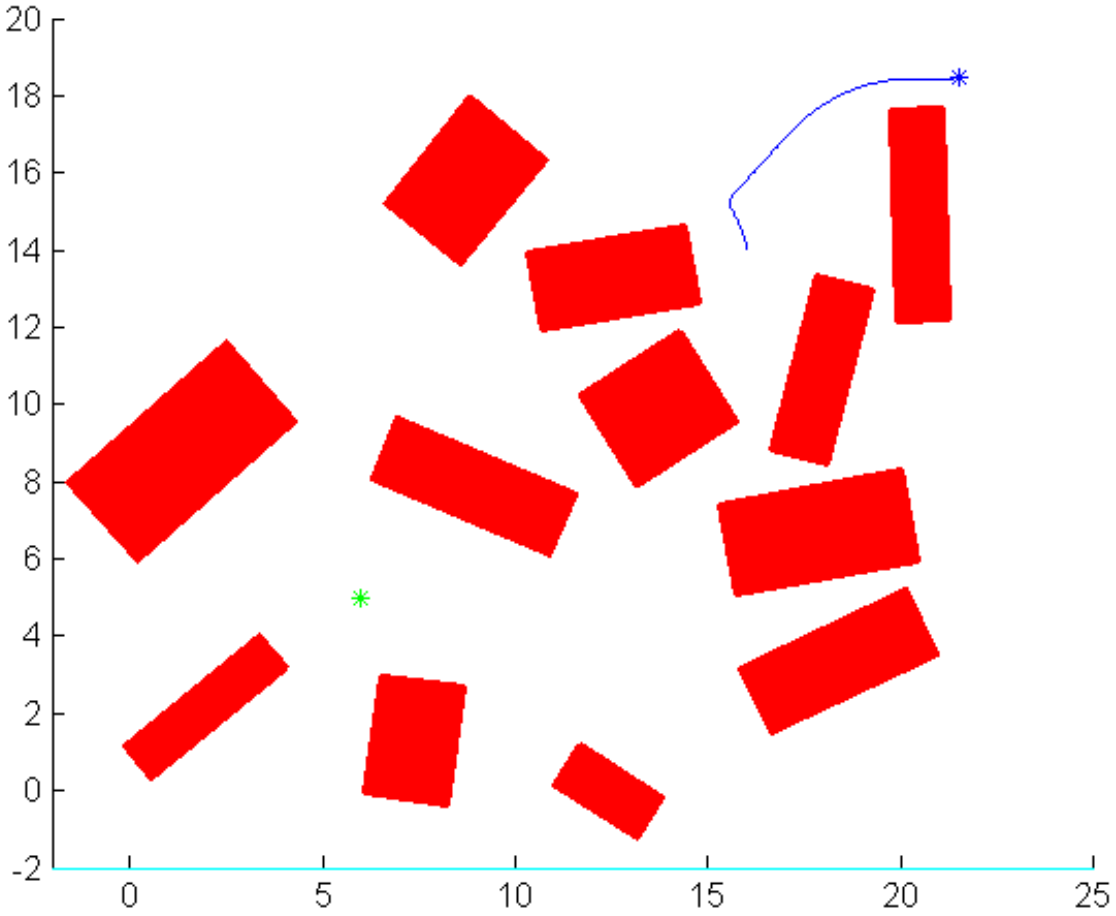
Potential Fields

- Potential fields example
  - Gradient field



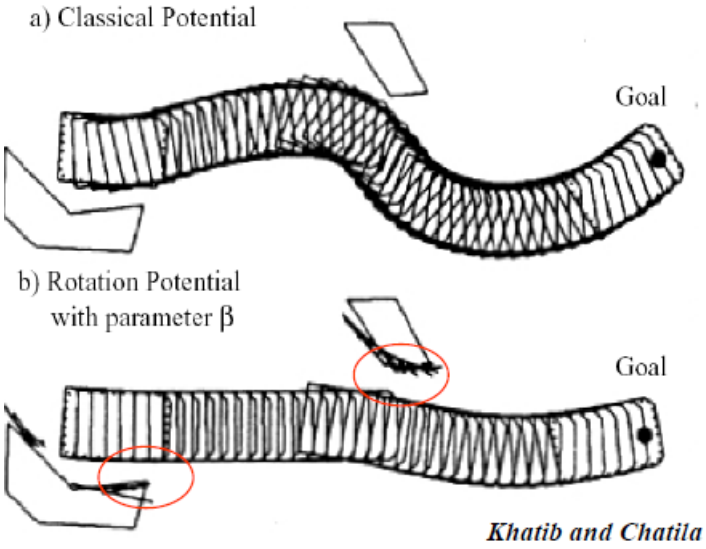
Potential Fields

- Potential fields example
  - The trajectory



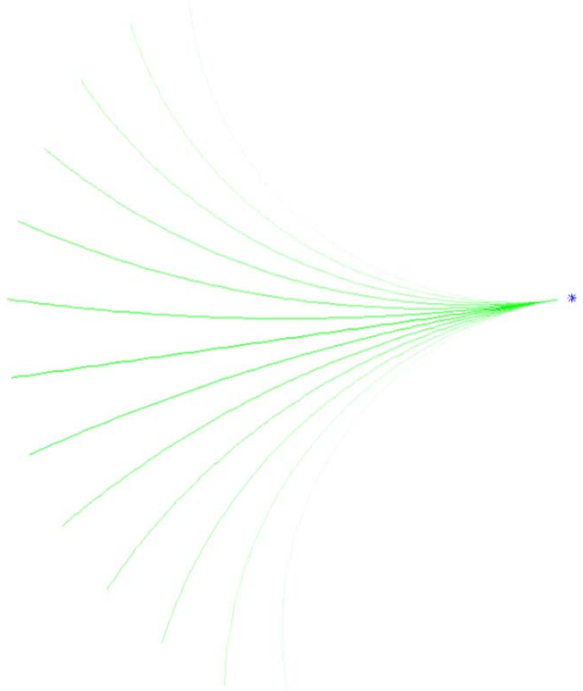
Potential Fields

- Extended Potential Field
  - Can add effect to manage vehicle heading
    - A specific adaptation for driving robots
      - Rotation potential
        - Add a dependence on bearing to obstacle,
        - As bearing increases, reduce potential
        - No point worrying about what's behind you



Khatib and Chatila

Trajectory Rollout



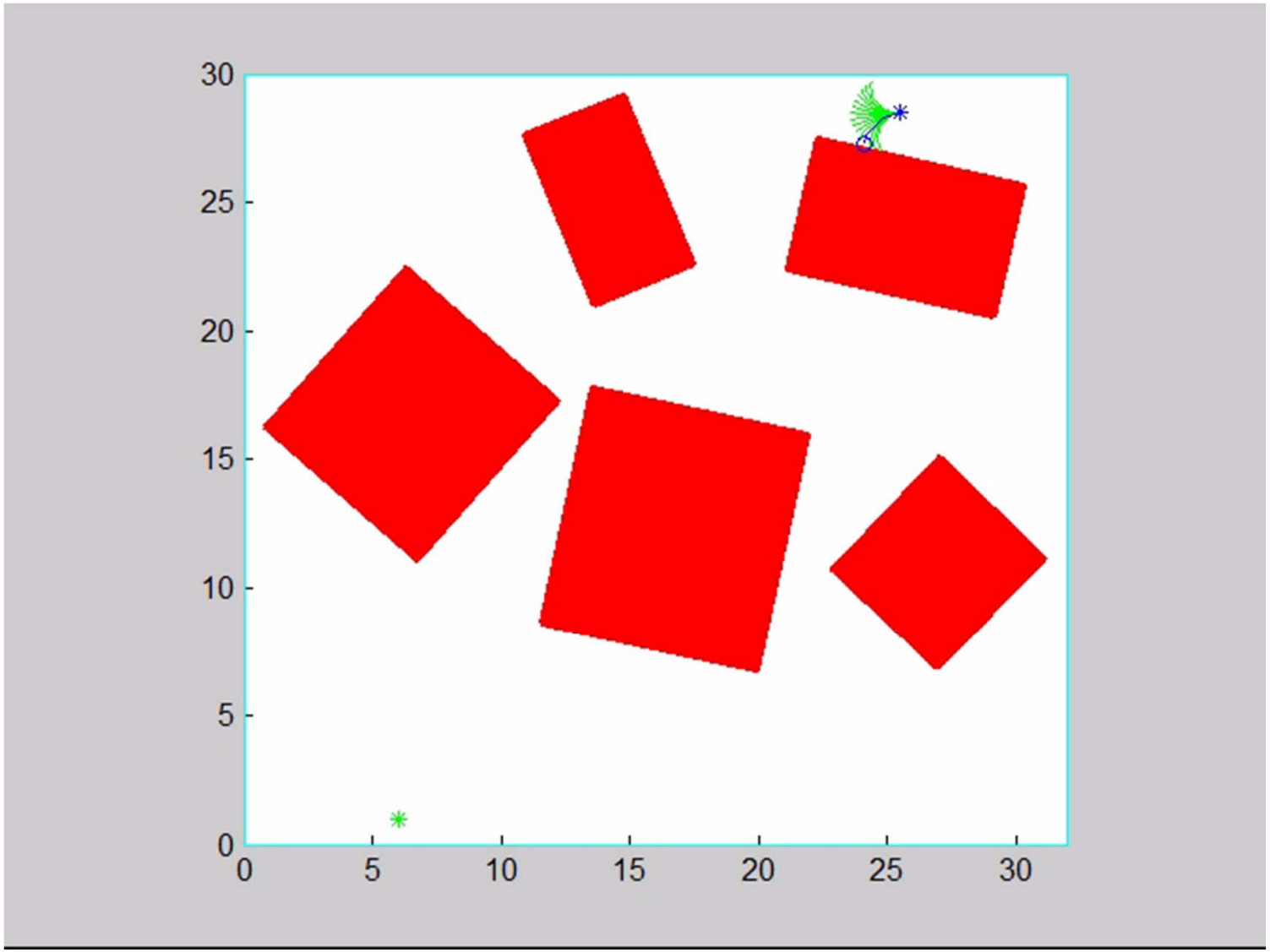
- Select n inputs to apply
  - Eg. Const velocity, 10 different rotation rates
- Propagate trajectory forward to time $t+T$
- Check each trajectory for collisions
- Score each trajectory based on
  - Progress to goal
  - Distance from obstacles
  - Similarity to previous choice
  - Preference between input choices
  - Etc…
- Pick best option and apply input
- Repeat as quickly as possible

Trajectory Rollout

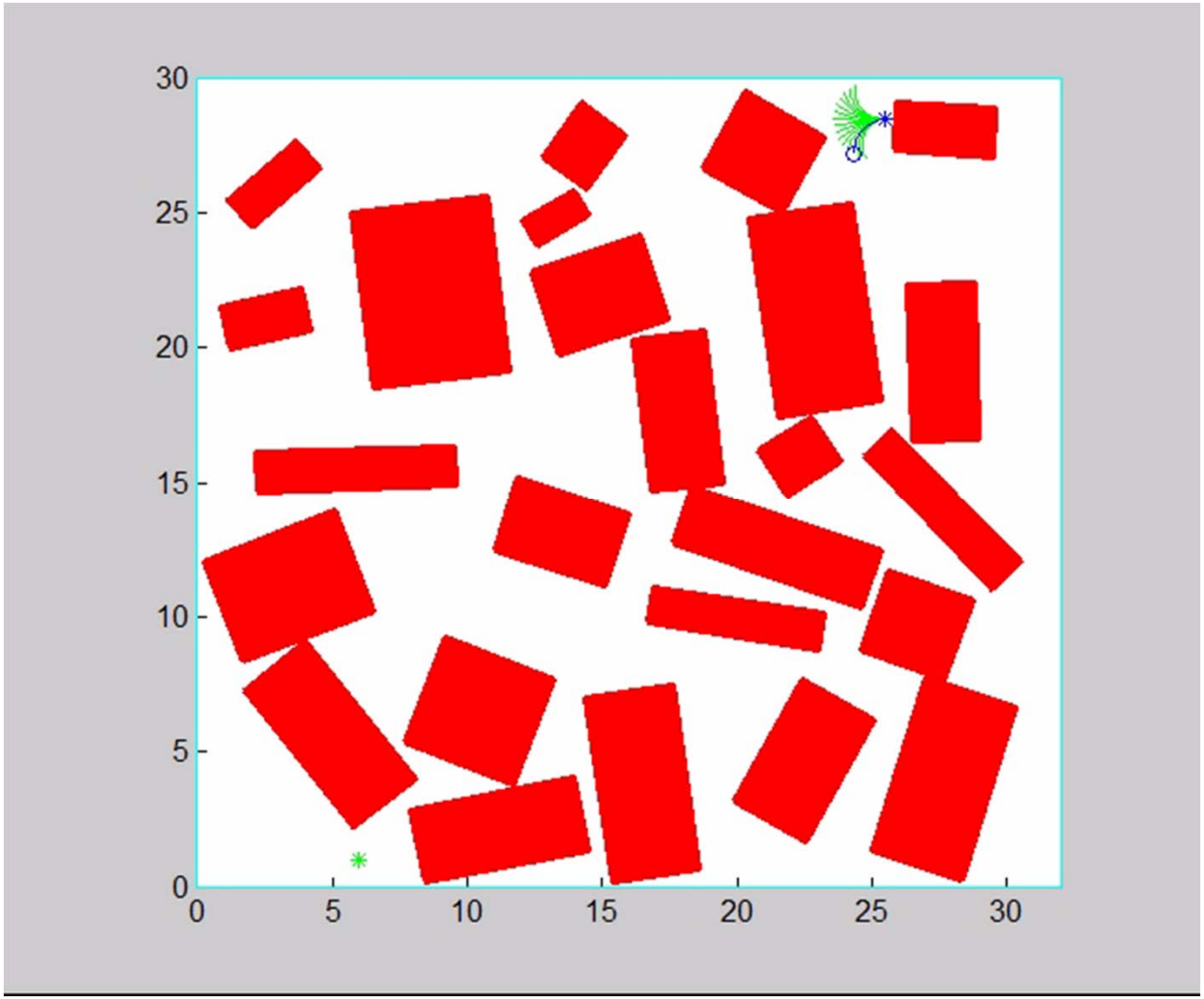
- Example
  - Two-wheeled robot
  - n = 11 trajectories
  - T = 1 second
  - v = 2 m/s
  - ω = [-2, 2] rad/s
  - Update rate = 5 Hz

- Environments with 5 well spaced and 25 not-so-well spaced obstacles

Trajectory Rollout – 5 Obstacles



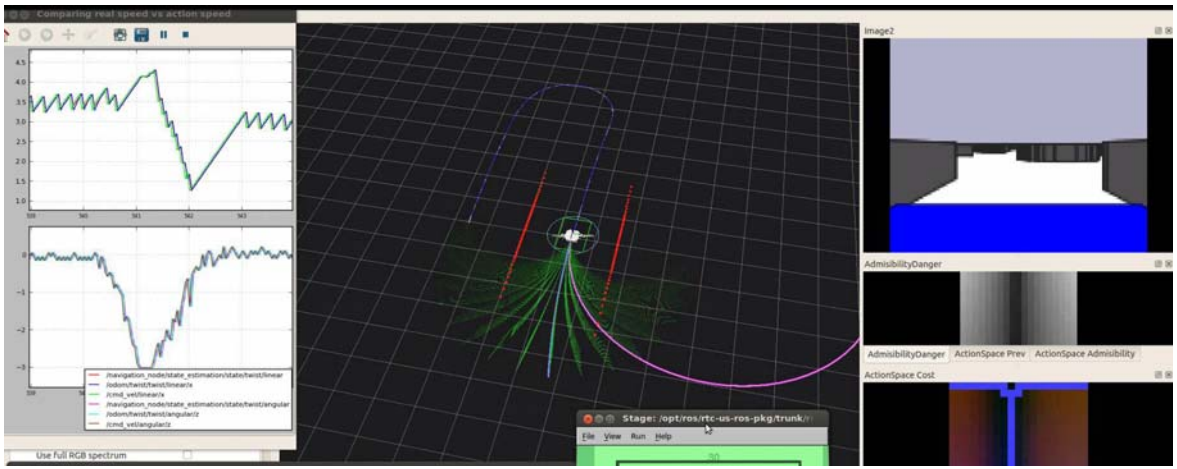
Trajectory Rollout – 25 Obstacles



Dynamic Window Approach

- Identical to Trajectory Rollout except:
  - Add dynamic constraint on input choices
    - Max angular acceleration limits rotation rate options at each timestep
    - Same for max translational acceleration if varying velocity

- Both are implemented in ROS navigation stack
  - You've already used these



PLANNING

- Summary - Reactive Planners
  - Fast computationally
    - Unless entire potential field must be computed (wavefront)
  - Simple control laws
    - Low computation requirements
    - Great for microcontroller based robots
  - Difficult to find globally optimal solutions
  - Do not consider dynamic constraints
  - Great for 2D, and for maneuverable robots

Outline

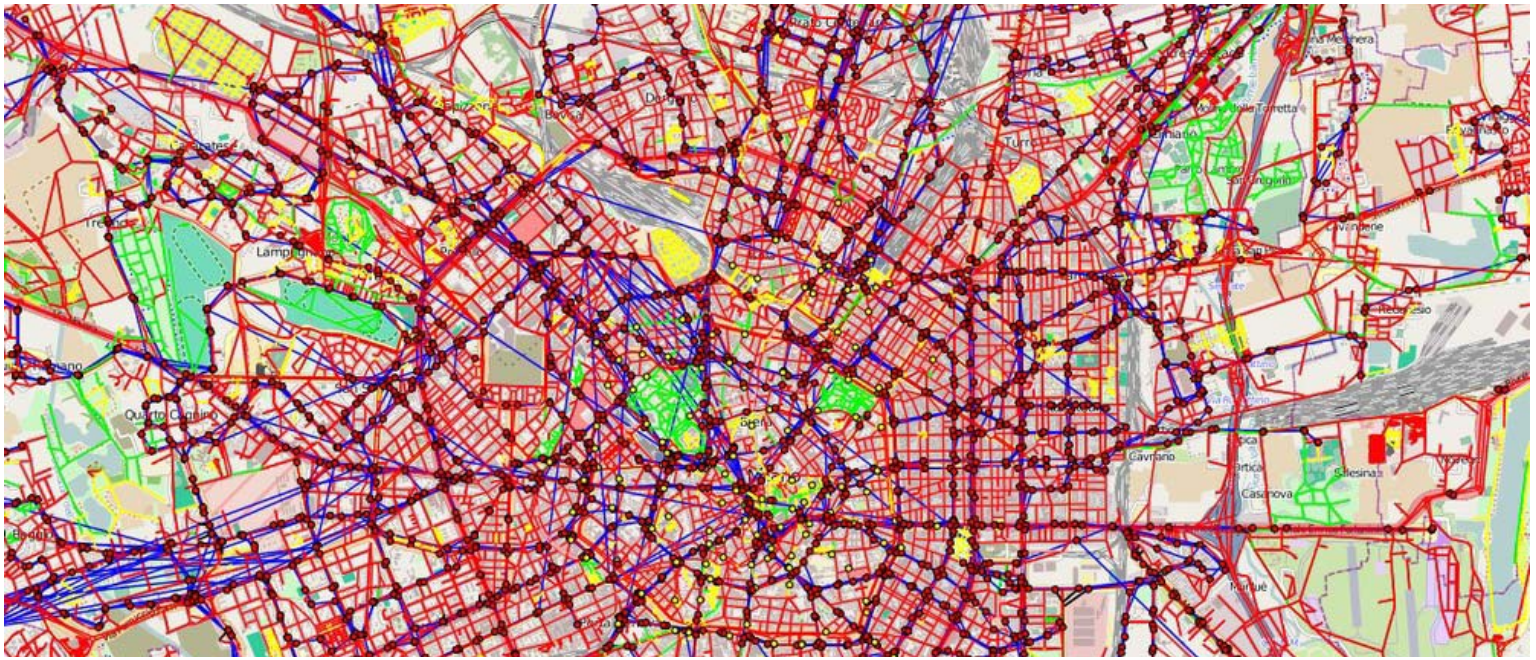
- Planning Concepts

- Reactive Motion Planning Algorithms
  - Bug
  - Potential Fields
  - Trajectory Rollout

- Graph Based Motion Planning
  - Finding paths on graphs
    - Wavefront
    - Dijkstra, A*, D*
  - Generating Graphs from environments
    - Visibility Graphs
    - Decompositions

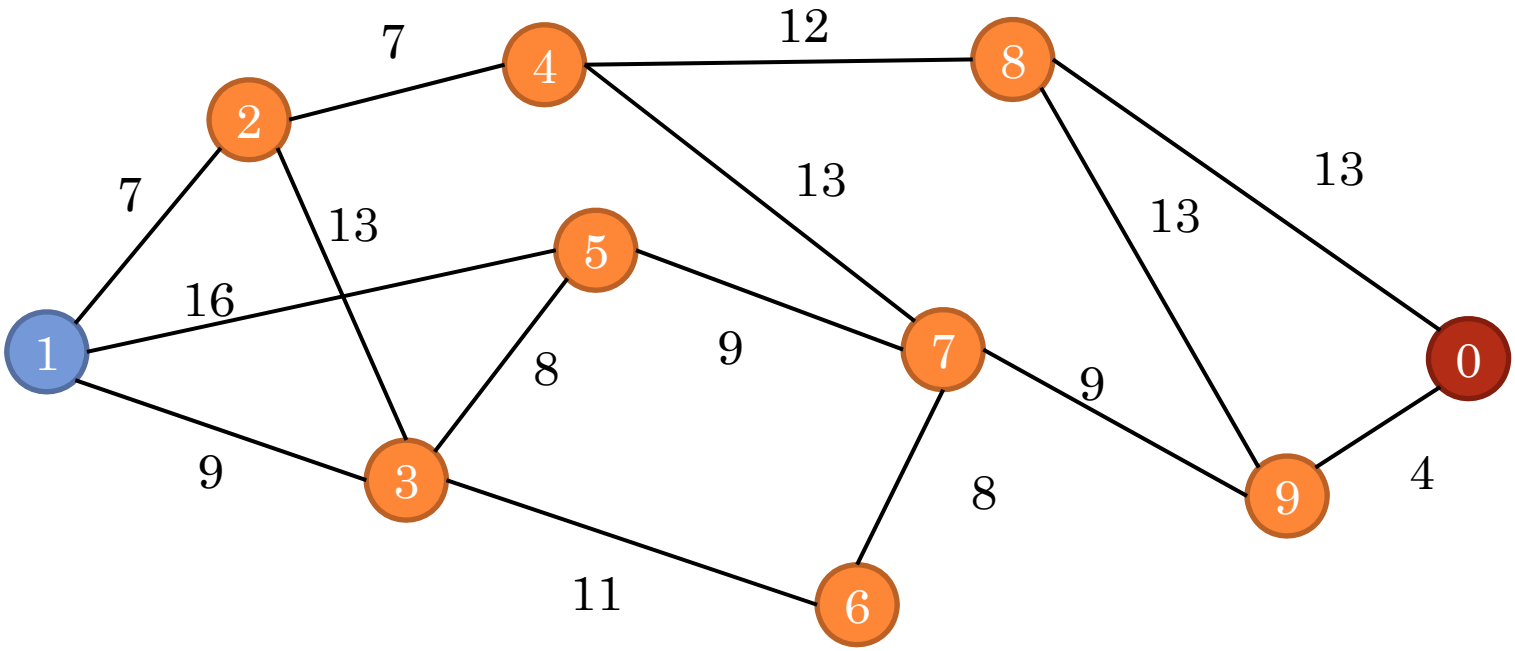
Planning

- Graph-Based Planning
  - Suppose map can be represented by a set of nodes and edges along which the vehicle can travel
  - Can apply graph based shortest path solutions to find a path quickly
    - Optimal over graph
  - Ignore dynamics



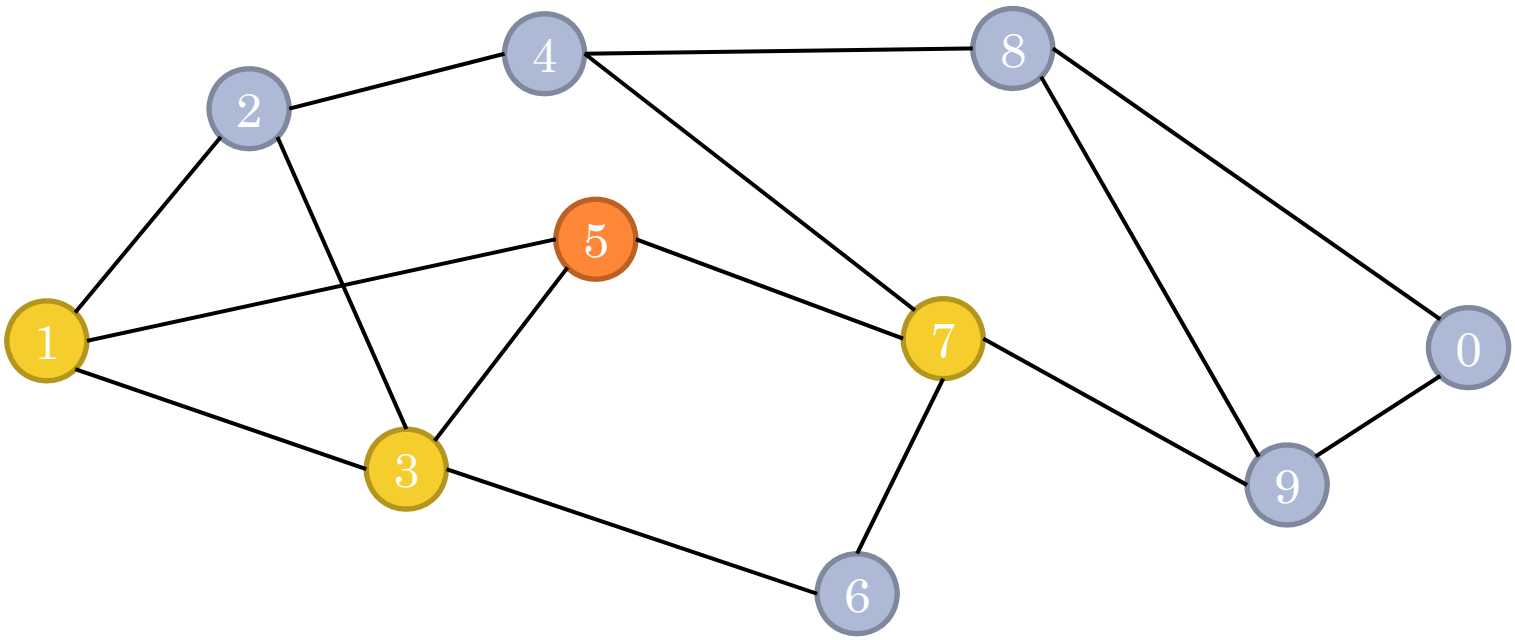
PLANNING

- Definition of graph
  - Graph G of nodes N with edges E: $G(N,E)$
  - Cost of traveling from n_i to n_j: $c(n_i,n_j)$
    - $c(n_1,n_3) = 9$



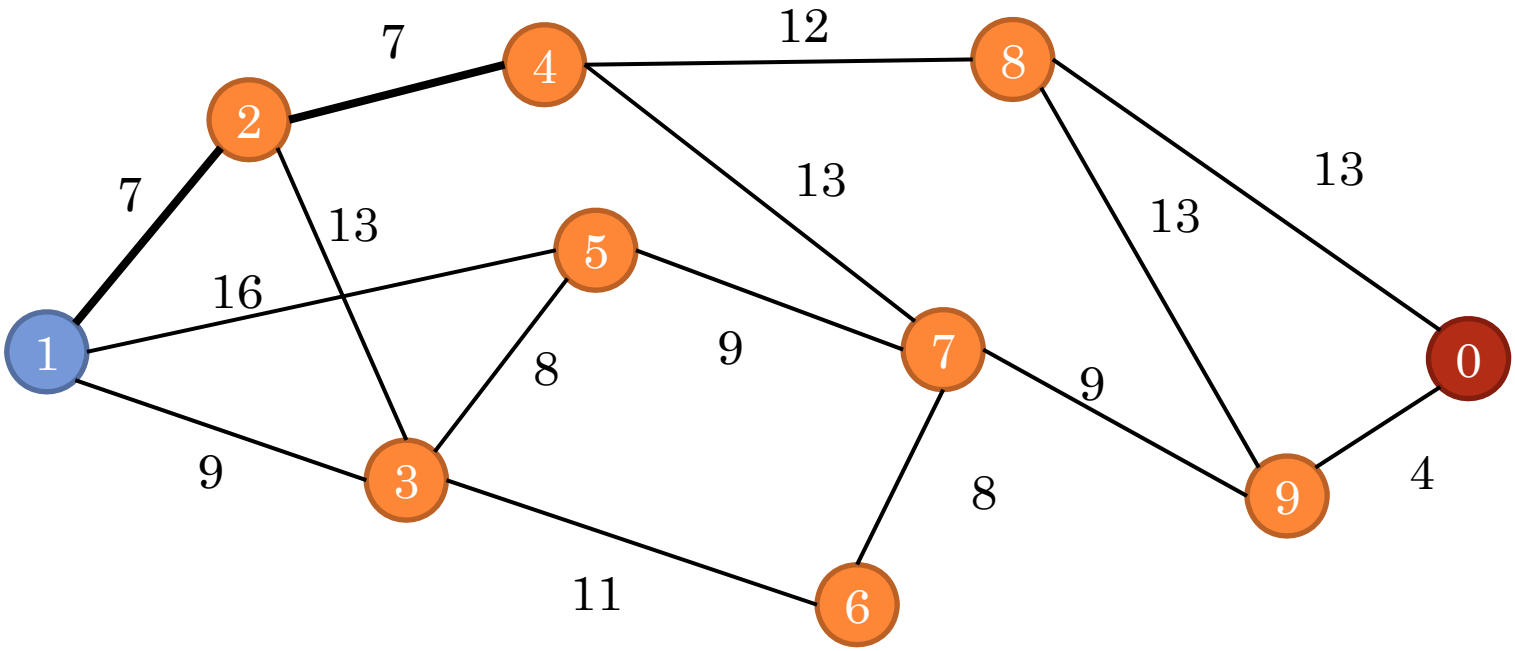
Planning

- Neighbouring nodes
  - Set of nodes adjacent to n: $A(n)$
    - $A(n_5) = \{n_1, n_3, n_7\}$



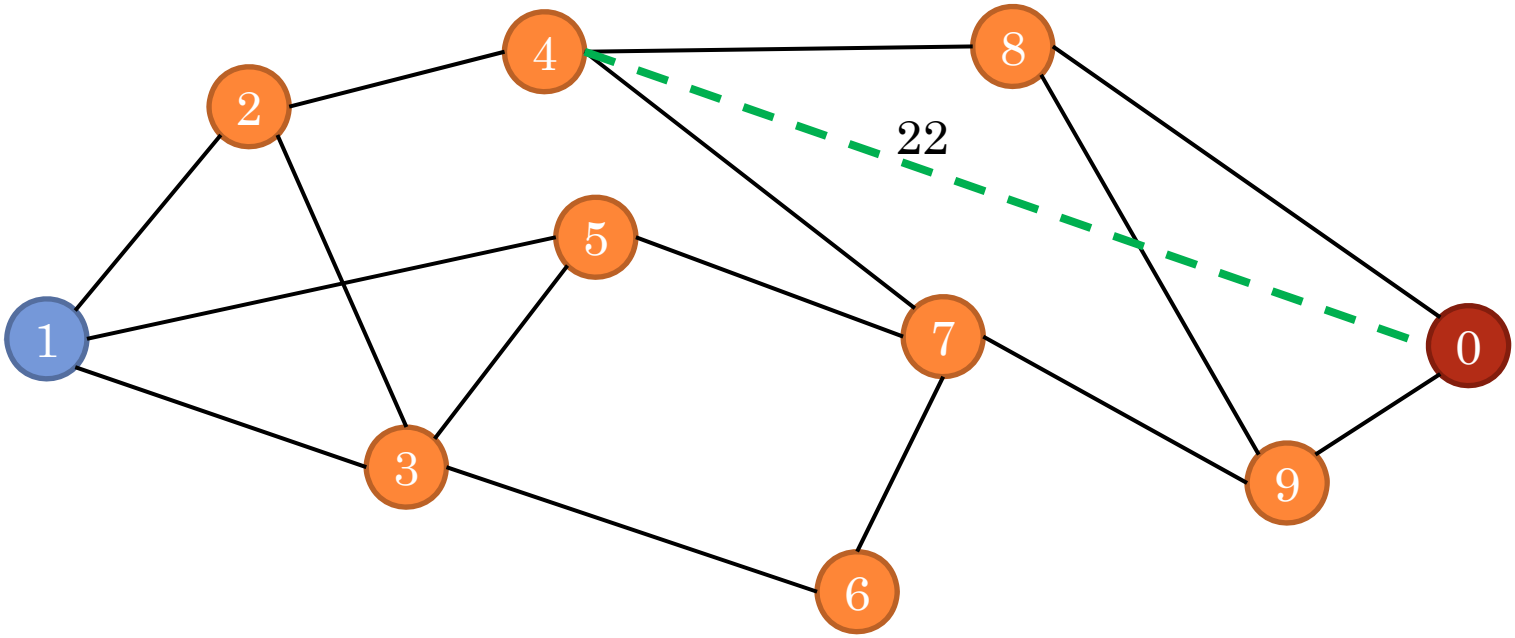
Planning

- Current cost
  - Minimum cost of getting to node n: $g(n)$
    - $g(n_4) = 14$



Planning

- Cost to go
  - Cost to go heuristic from node n to the end: $h(n)$
    - $h(n_4) = 22$ for straight line distance metric
    - Must always be less than or equal to true cost to go

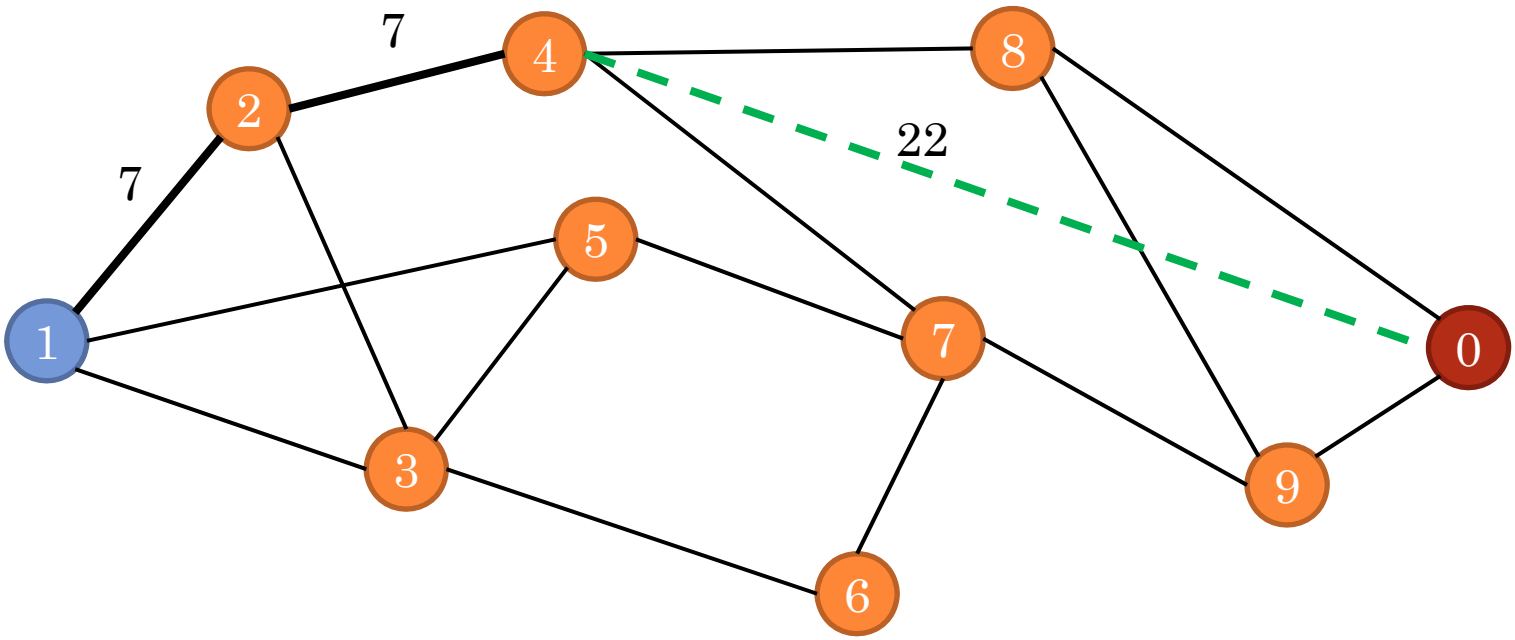


Planning

- Cost lower bound
  - Estimated cost of shortest path through node *n*:

$$f(n) = g(n) + h(n)$$

  - $f(n_4) = 14 + 22 = 36$



Planning

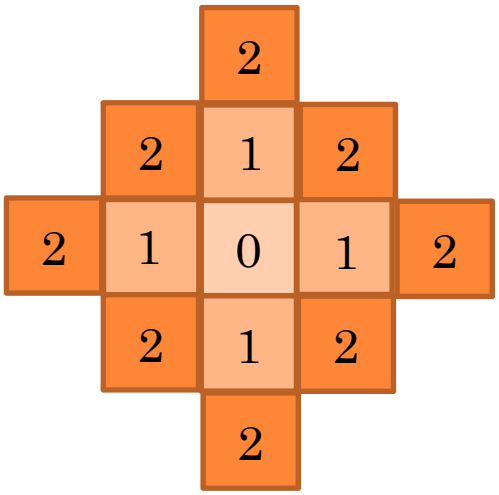
- Finding the shortest path over a graph
  - Breadth first search
    - Start at starting node
    - Find all nodes that can be reached in one step (neighbours)
    - For each neighbour in previous step, find all of its neighbours, and repeat until all nodes (or end node) has been reached
    - **Only works for edges of equal length**
  - Depth first search
    - Start at starting node
    - Pick an available node based on some criteria (longest, closest to goal)
    - Proceed as far as possible, then backtrack
    - Continue until all nodes have been visited
    - **Only works for edges of equal length**

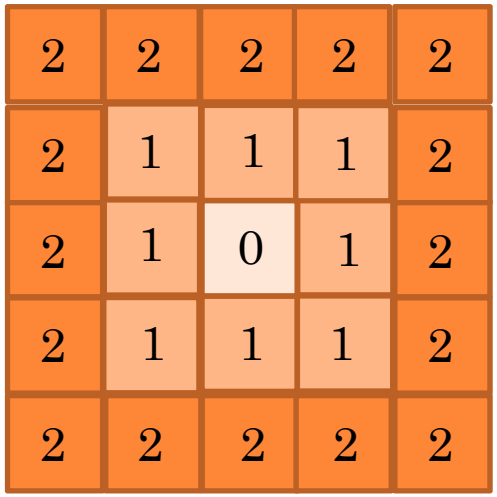
PLANNING

- Wavefront
  - If the graph produced has unit cost edges, breadth first search can be used
  - Resembles the propagation of a wave through graph
    - Works well in 2D, 3D for reasonable discretizations
  - Resulting cost map is monotonic
    - Leads to shortest path from *any point* in the occupancy grid to the final position
    - Or from current position to every point in the graph

PLANNING

- Underlying graph structure for wavefront
  - Add edges of unit cost by discretizing free space with an occupancy grid





PLANNING

- Define two sets
  - Open Set: O
    - Set of nodes currently under consideration
    - Initialize with start node n_0
    - Implemented as a queue, stack or priority queue
      - Queue – breadth first search
      - Stack – depth first search
      - Priority queue – Dijkstra's and A*
    - Top node is first node in queue or stack form of open set
    - Best node is first node in priority queue open set
  - Closed Set: C
    - All nodes for which processing is complete

PLANNING

- Breadth first search algorithm
  - While top node is not goal
    - Move top node from open set to closed set
      - Store node, back pointer to previous node and current cost

$$f(n_{top}) \leq f(n), \forall n \in O$$

    - Add all neighbouring nodes of top node not currently in either set to the bottom of the open set
      - Store node, current cost and back pointer to top node

$$O = \{O, A(n_{top}) \setminus (O \cup C)\}$$

    - For each node already in the open set, update current cost and back pointer if new path is shorter

$$\text{for all } n \in O \cap A(n_{top})$$

$$\text{if } (g(n_{top}) + 1 < g(n))$$

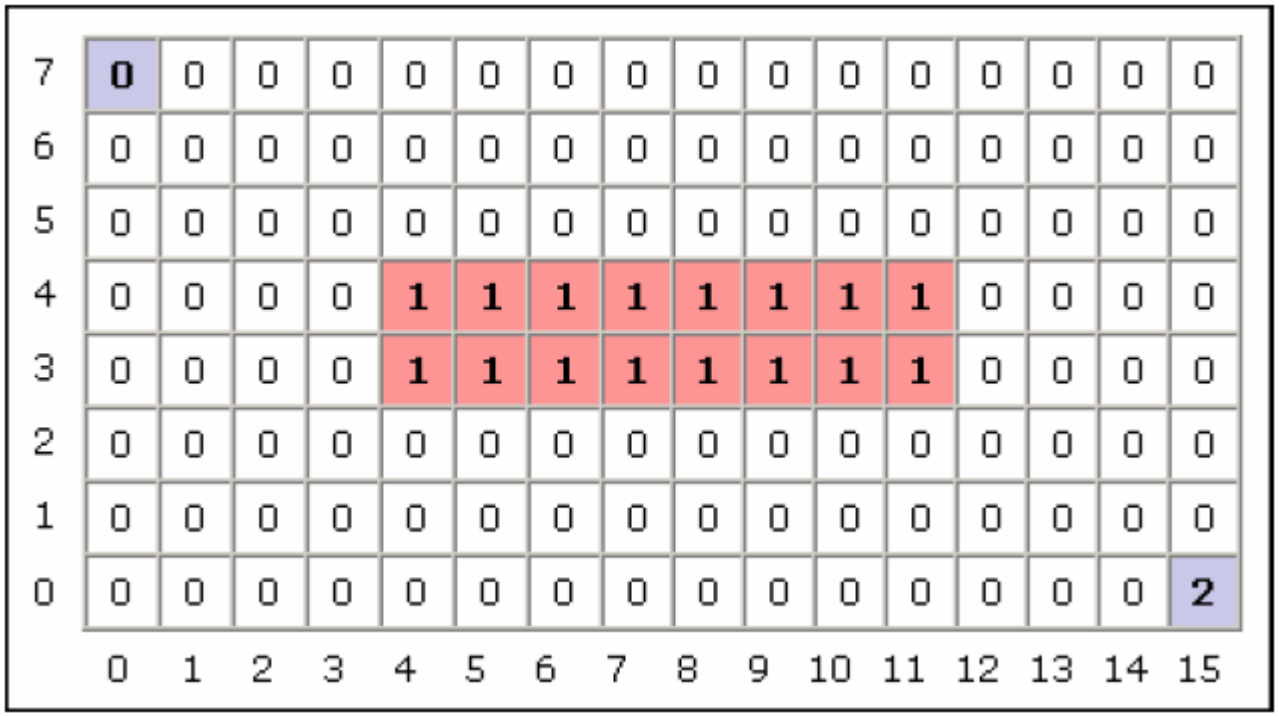
$$\text{backpoint to } n_{top}, \text{ update } g(n)$$

PLANNING

- Wavefront Algorithm
  - Initialization
    - Create open set of positions, which includes only the end point, assign a cost of 0
    - Create a closed set of position, which includes all obstacles, assign a cost of infinity
  - Main loop
    - First position of open set becomes active
      - Move to closed set
      - Identify all neighbours that can be reached and are not already in open or closed sets
      - Update each neighbour in open set with lower of the cost through current node or previous best cost
      - Assign each new neighbour a cost of the active position +1
      - Add all new neighbours to the end of the open set
    - Until open set is empty

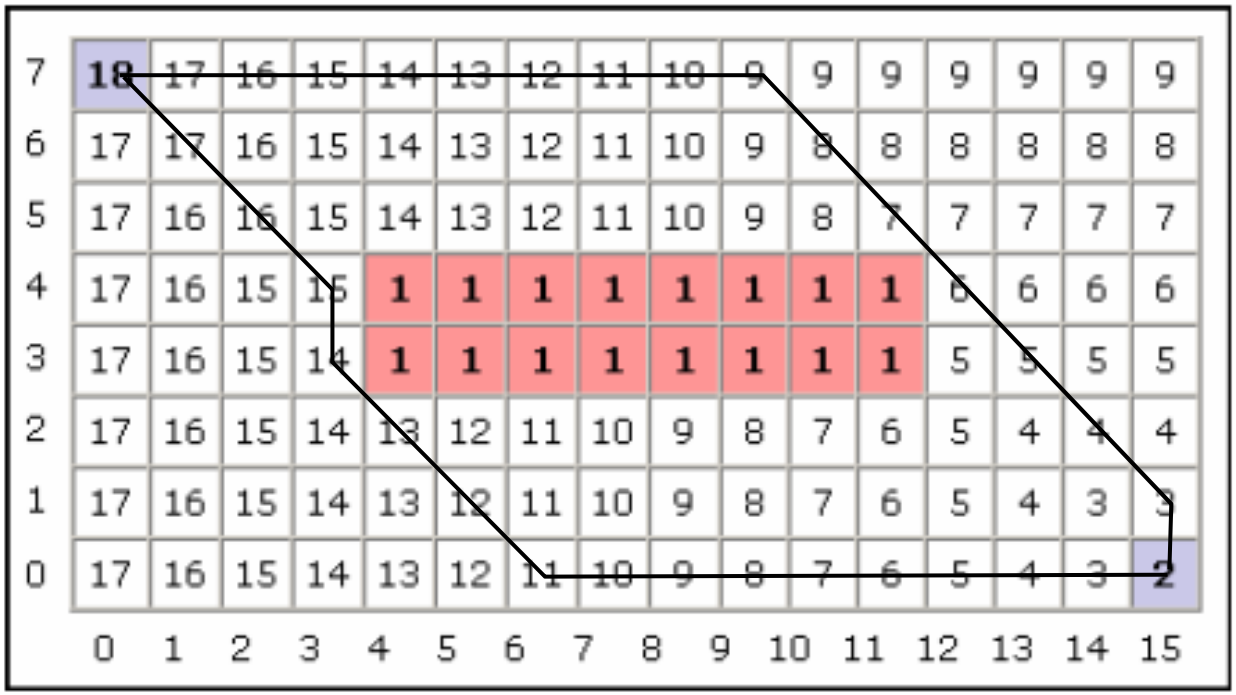
Planning

- Wavefront
  - Example



PLANNING

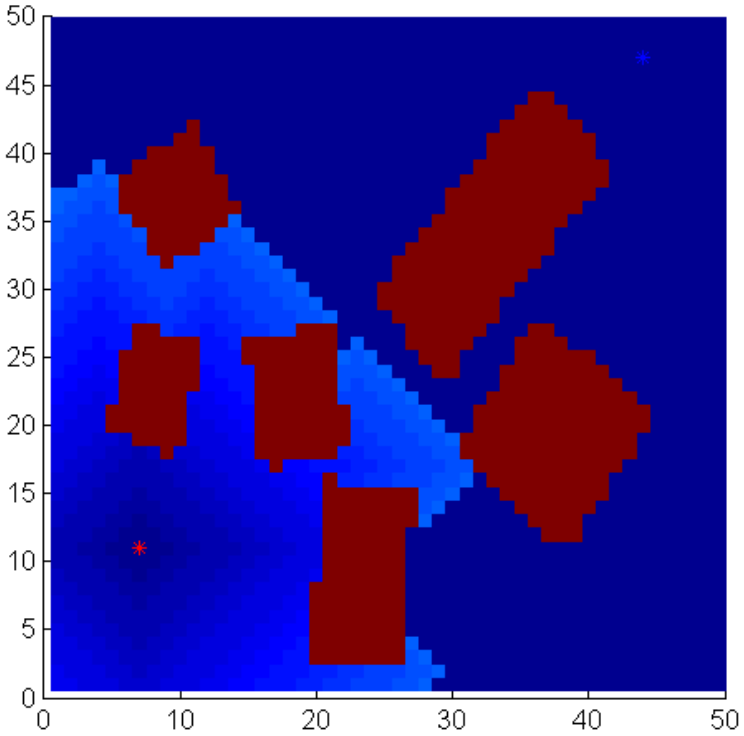
- Wavefront
  - Example



16-735, Howie Choset, with slides from Ji Yeong Lee, G.D. Hager and Z. Dodds

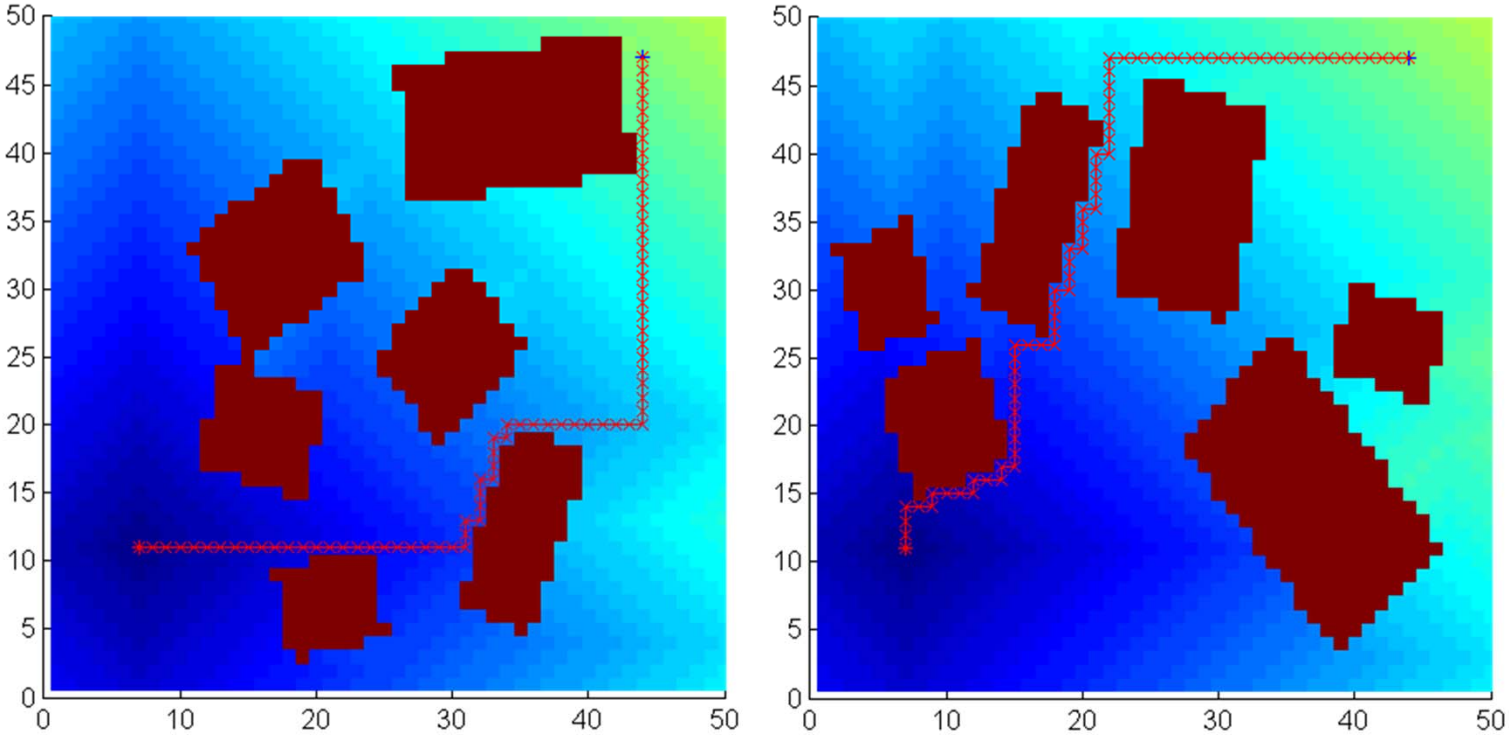
Planning

- Wavefront
  - 50x50 grid (converted to a graph and solved using breadth first search)
    - Link to video



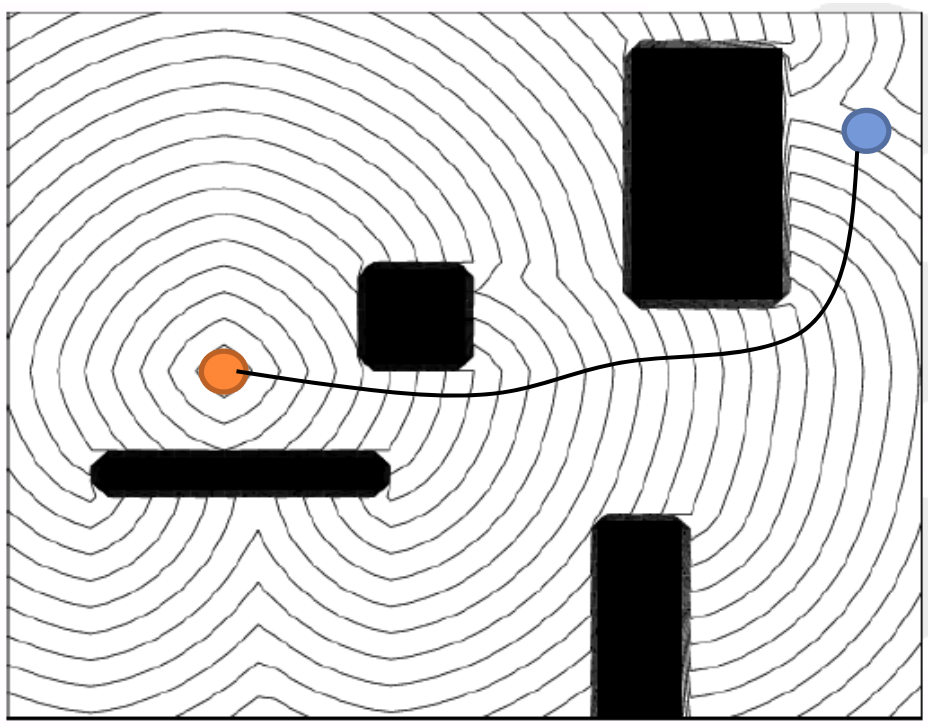
PLANNING

- Wavefront
  - The vehicle then identifies a path by always selecting a position that reduces the cost to goal.
    - Can be performed locally, wavefront is monotonic
    - Many possible trajectories result



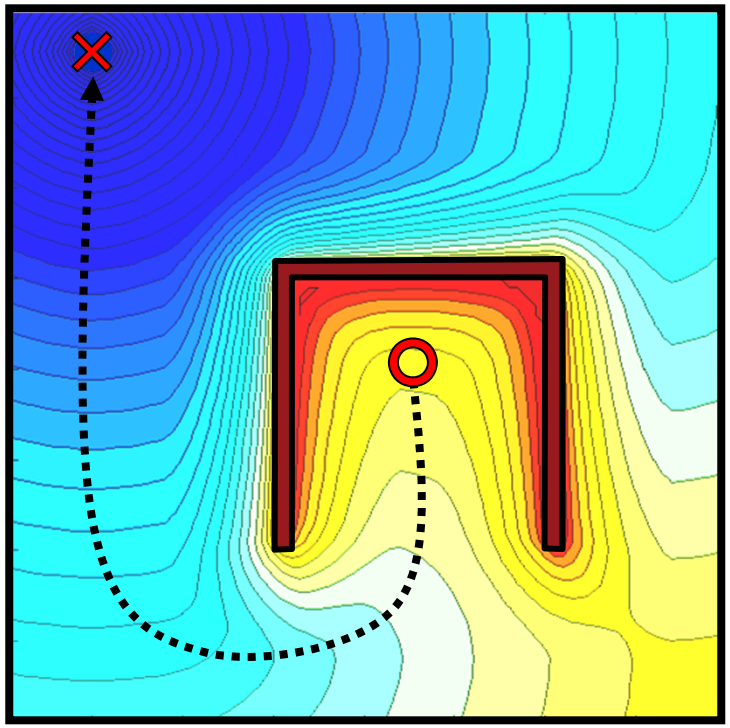
Planning

- Fast Marching
  - Can extend the basic wavefront algorithm to use more of a continuum based approach



Planning

- Fast Marching
  - Can define viscosity of flow around obstacles
  - Results in a smooth path that does not hug obstacle corners



PLANNING

- Breath-First, Wavefront and Fast Marching
  - Pros:
    - Monotic, always find path to goal if it exists
    - Easy to implement
  - Cons:
    - Computes path from every point in planning space to end goal
      - Not very efficient, but fast enough for 2D
    - Must treat environment as discretized graph with unit step edges (occupancy grid)
      - Approximation always leads to sub-optimality in resulting path

PLANNING

- Finding the shortest path over a graph
  - Dijkstra's algorithm
    - Start from starting node
    - Expand all links out of the node with lowest current cost
    - Find the next lowest current cost node, repeat previous step
    - Stop when end goal is closed, no other path can be shorter
  - A* Algorithm
    - Modified version of Dijkstra's
    - Rely on edge costs and cost to go heuristic
    - Pick most promising node at each step
    - Cost to go heuristic should never be greater than true cost
  - Can run all these algorithms from current location forward or from end point backward

PLANNING

- Dijkstra's algorithm
  - While best node is not goal
    - Move best node from open set to closed set
      - Store node, back pointer to previous node and current cost

$$f(n_{best}) \leq f(n), \forall n \in O$$

    - Add all neighbouring nodes of best node not currently in either set to the open set
      - Store node, current cost and back pointer to best node

$$O = \{O, A(n_{best}) \setminus (O \cup C)\}$$

    - For each node already in the open set, update current cost and back pointer if new path is shorter

$$\text{for all } n \in O \cap A(n_{best})$$

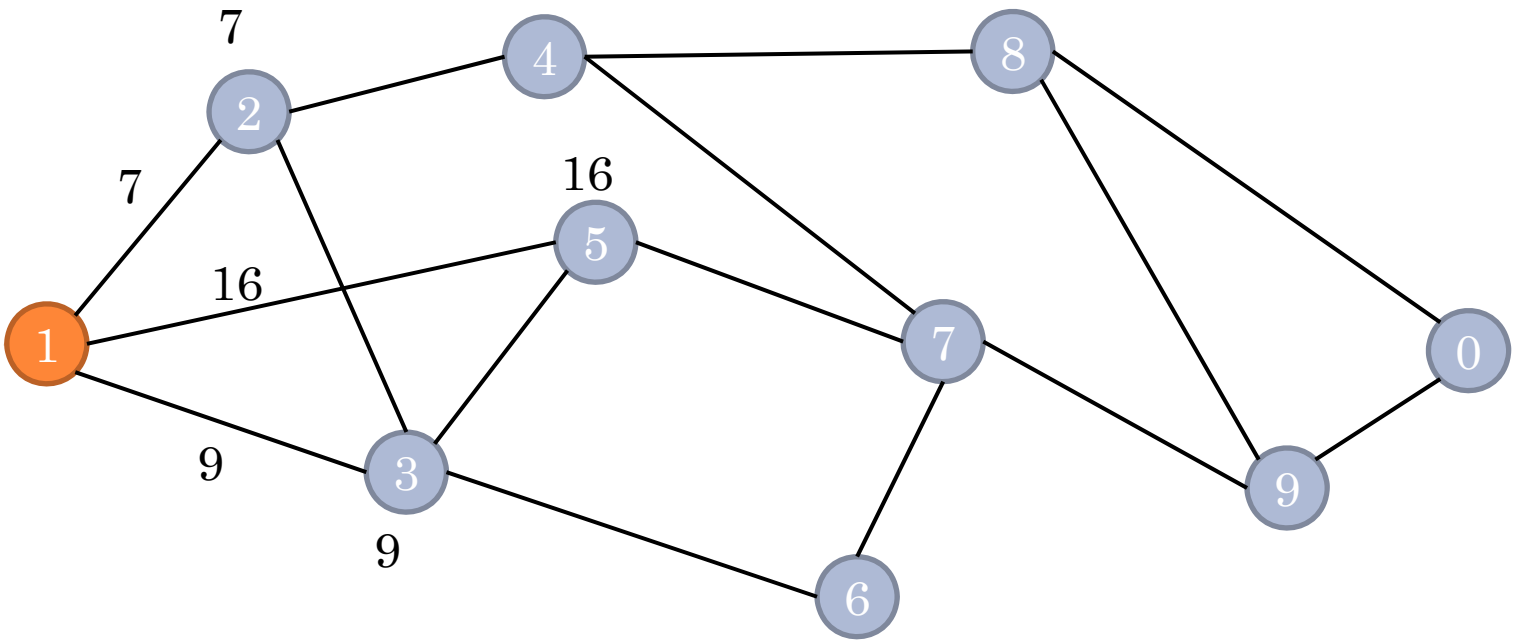
$$\text{if } (g(n_{best}) + c(n_{best}, n) < g(n))$$

$$\text{backpoint to } n_{best}, \text{ update } g(n)$$

PLANNING

O	C
(1,-,0)	

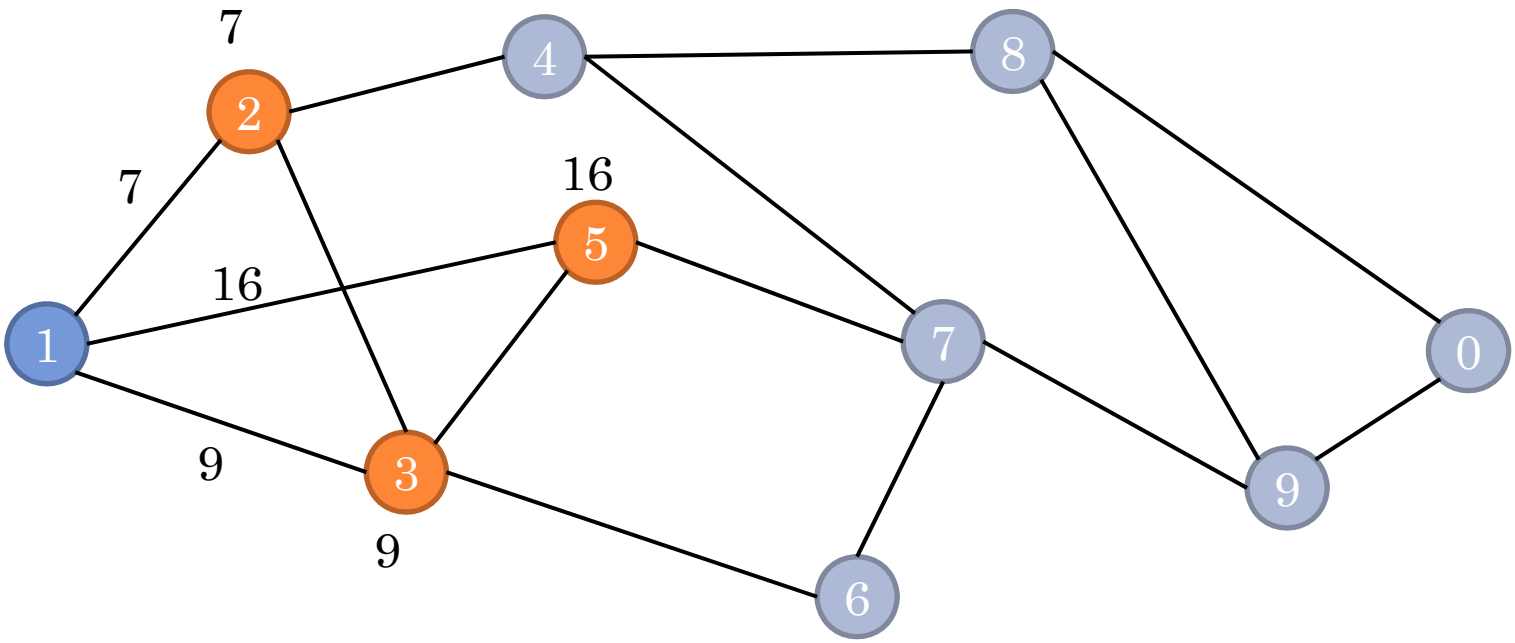
- Dijkstra's Search Algorithm
  - Take best node in O and move to C
  - Find all neighbours of best node, add to O in order of current cost



PLANNING

- Dijkstra's Search Algorithm
  - If a neighbour node is already in O, keep only shortest path to it

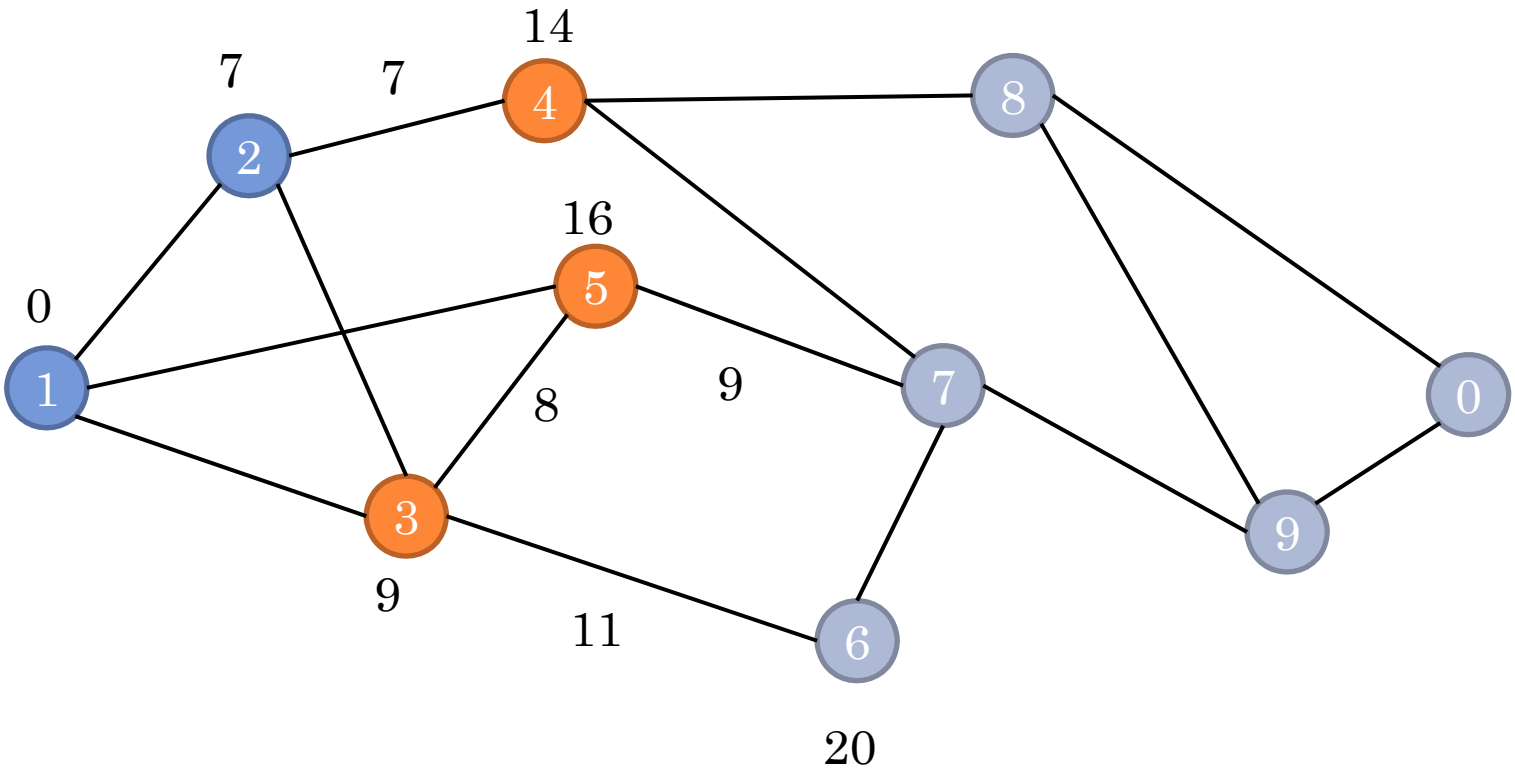
O	C
(2,1,7)	(1,-,0)
(3,1,9)	
(5,1,16)	



Planning

- Dijkstra's Search Algorithm
  - Repeat for each node in O

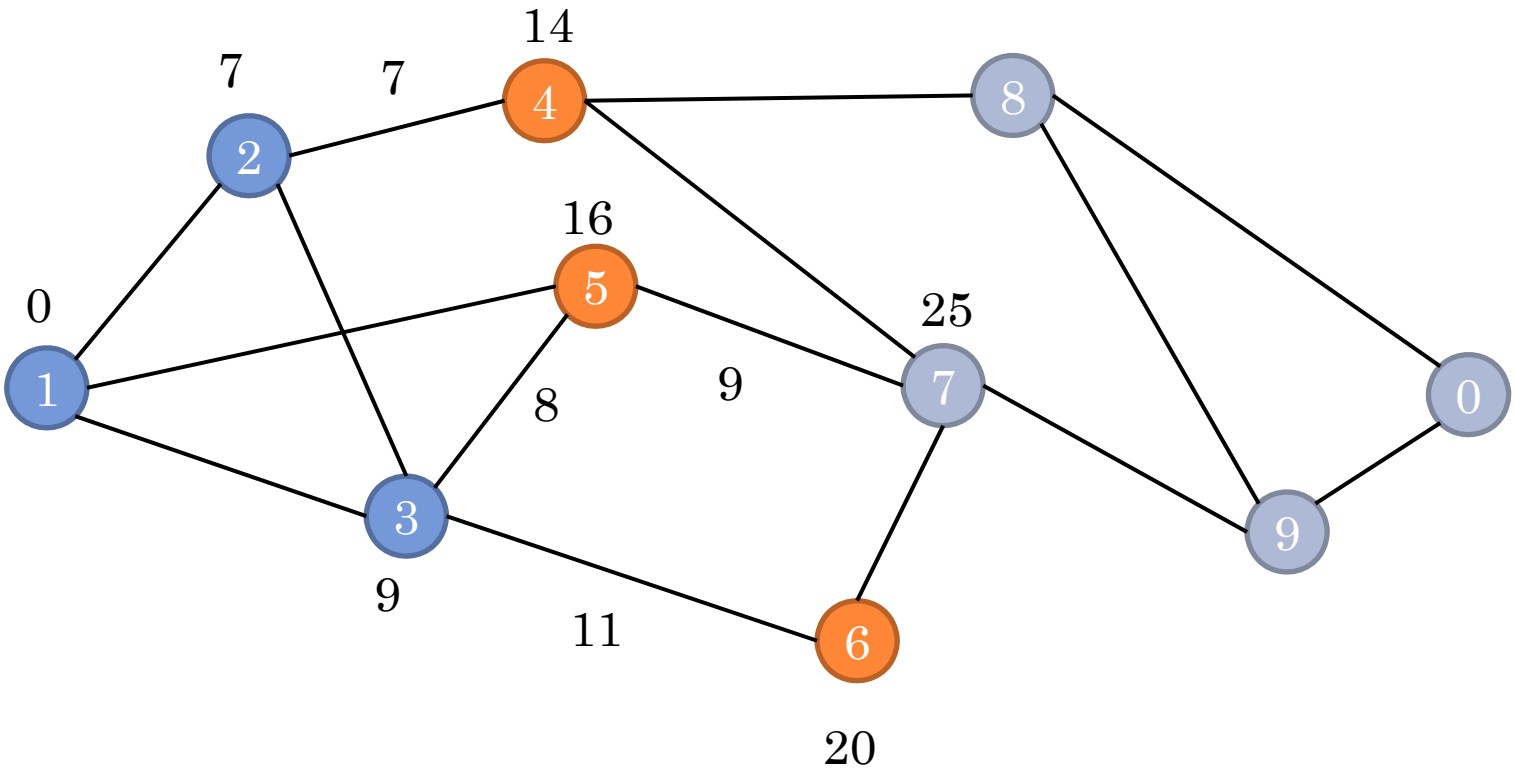
O	C
(3,1,9)	(1,-,0)
(4,2,14)	(2,1,7)
(5,1,16)	



PLANNING

- Dijkstra's Search Algorithm
  - Repeat for each node in O

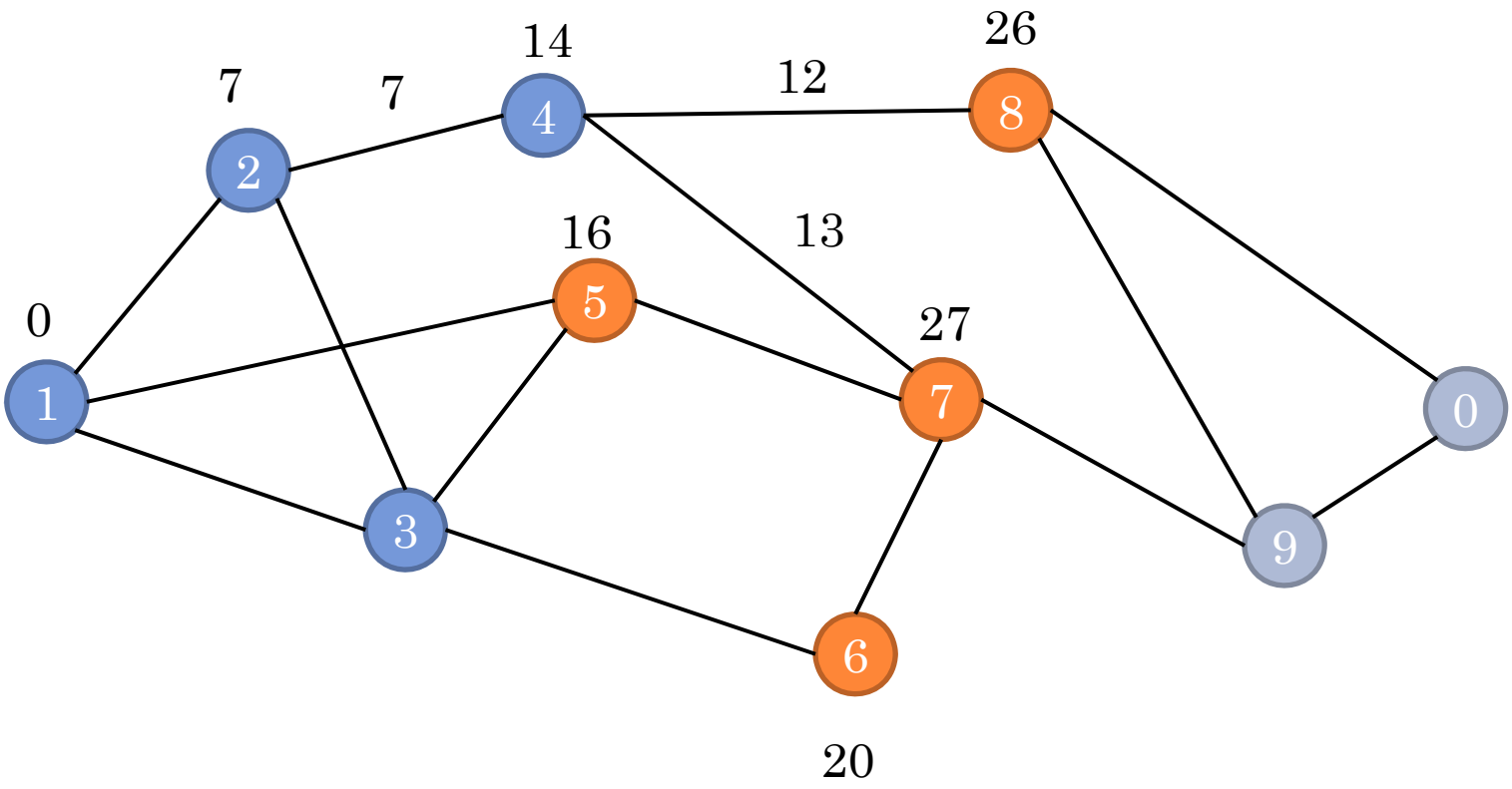
O	C
(4,2,14)	(1,-,0)
(5,1,16)	(2,1,7)
(6,3,20)	(3,1,9)



Planning

- Dijkstra's Search Algorithm
  - Repeat for each node in O

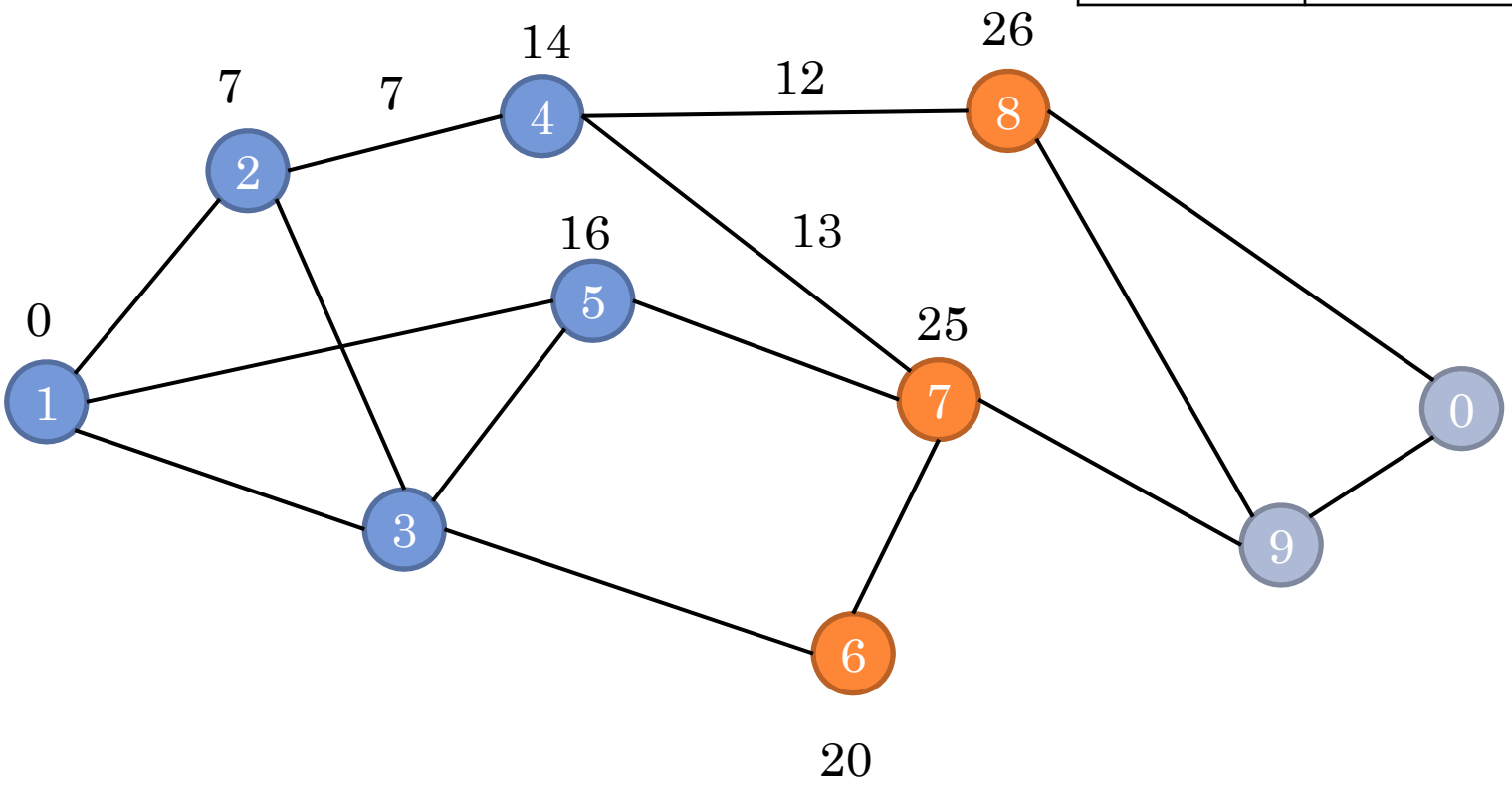
O	C
(5,1,16)	(1,-,0)
(6,3,20)	(2,1,7)
(8,4,26)	(3,1,9)
(7,5,27)	(4,2,14)



PLANNING

- Dijkstra's Search Algorithm
  - Repeat for each node in O

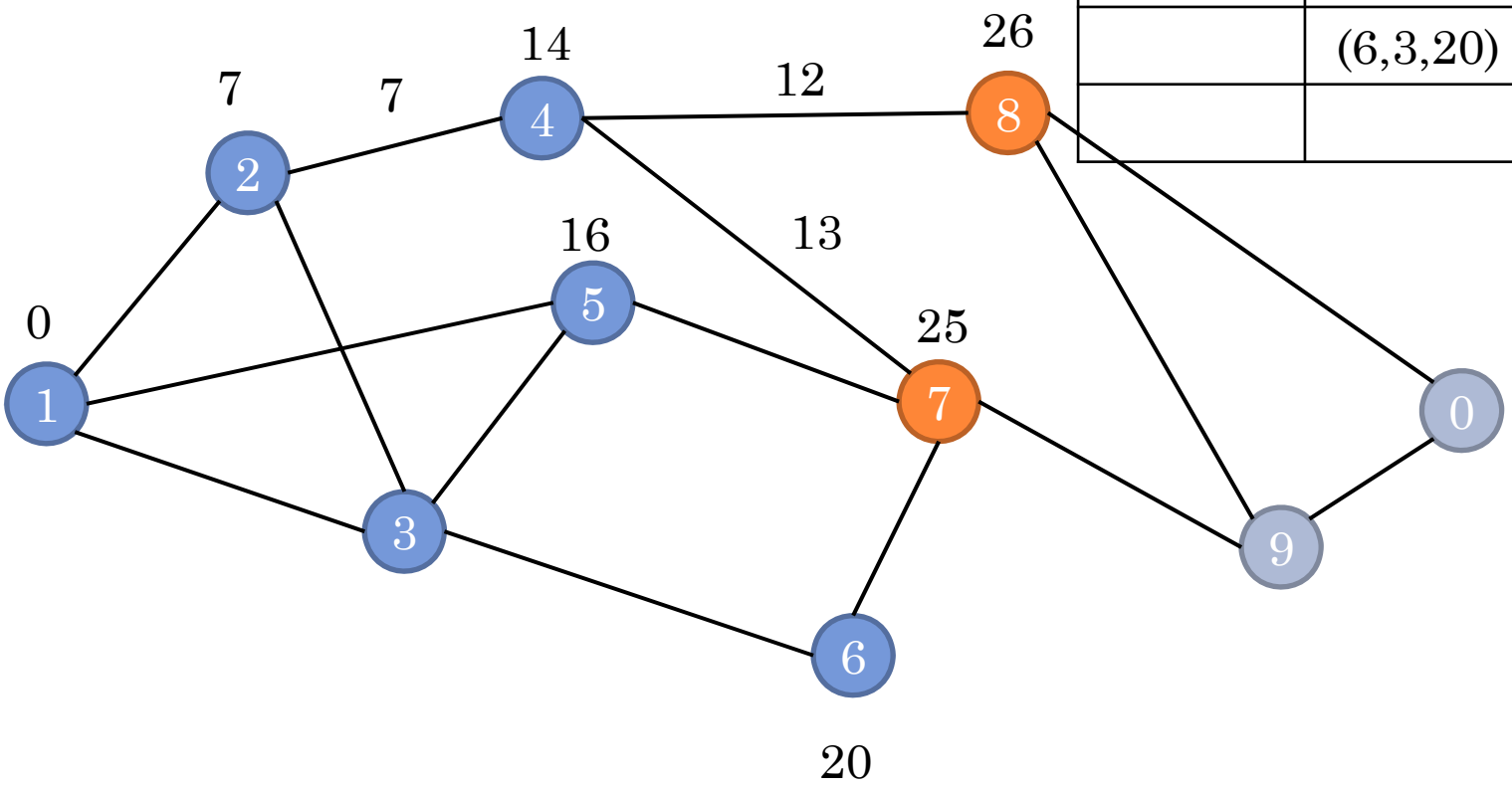
O	C
(6,3,20)	(1,-,0)
(7,5,25)	(2,1,7)
(8,4,26)	(3,1,9)
	(4,2,14)
	(5,1,16)



PLANNING

- Dijkstra's Search Algorithm
  - Repeat for each node in O

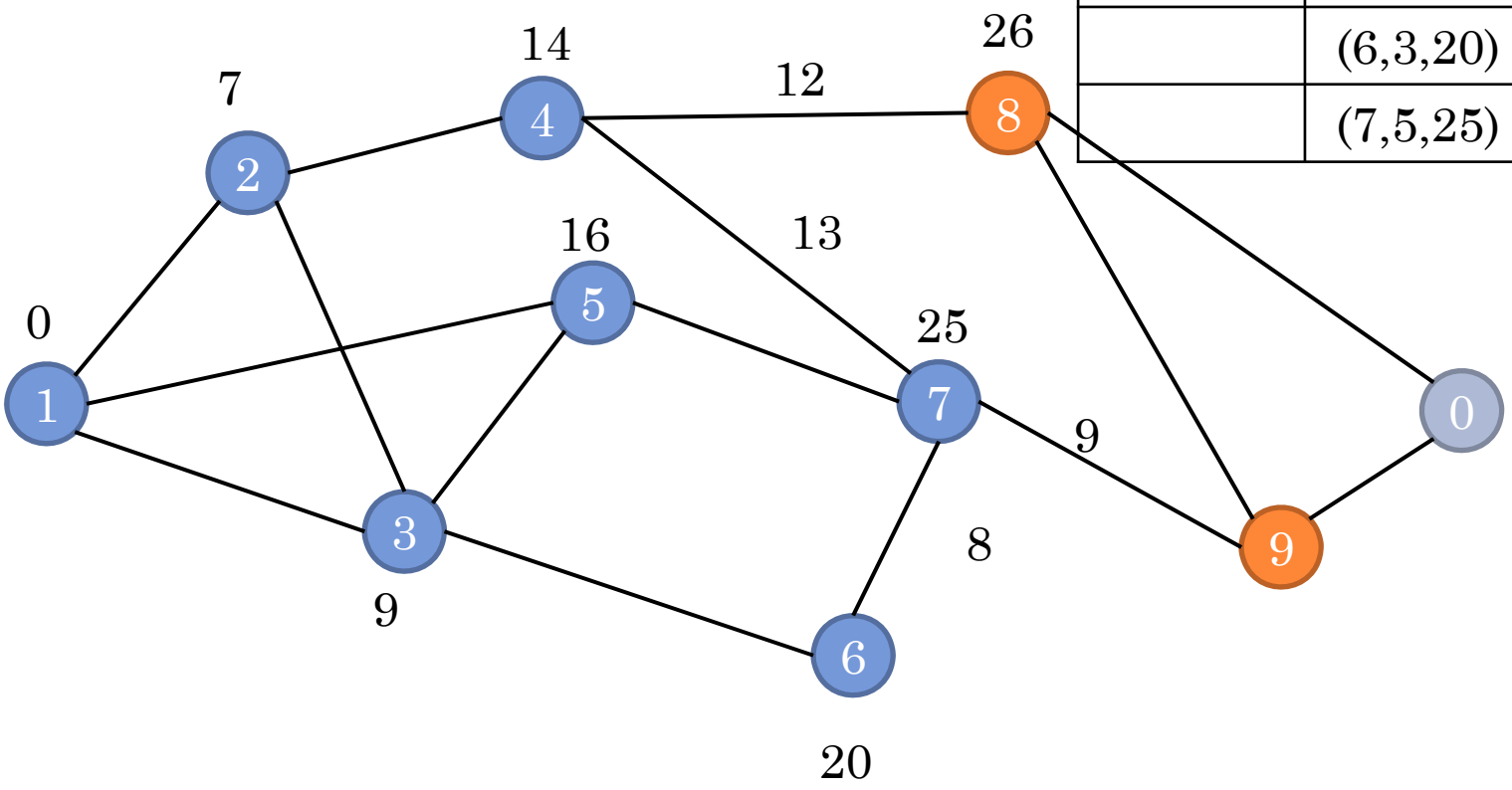
O	C
(7,5,25)	(1,-,0)
(8,4,26)	(2,1,7)
	(3,1,9)
	(4,2,14)
	(5,1,16)
	(6,3,20)



PLANNING

- Dijkstra's Search Algorithm

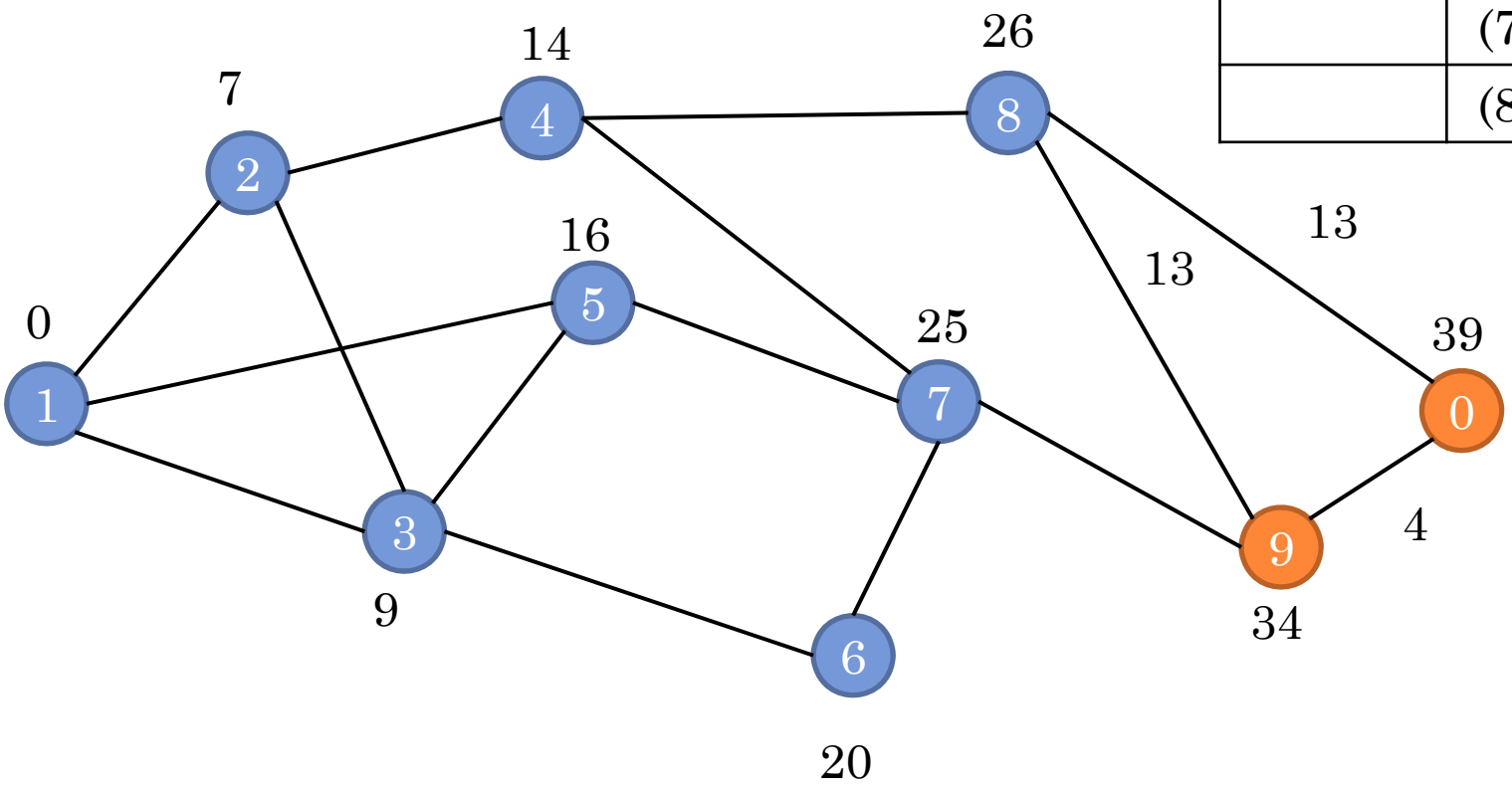
O	C
(8,4,26)	(1,-,0)
(9,7,34)	(2,1,7)
	(3,1,9)
	(4,2,14)
	(5,1,16)
	(6,3,20)
	(7,5,25)



Planning

- Dijkstra's Search Algorithm
  - Stop when end node is current best node in open list

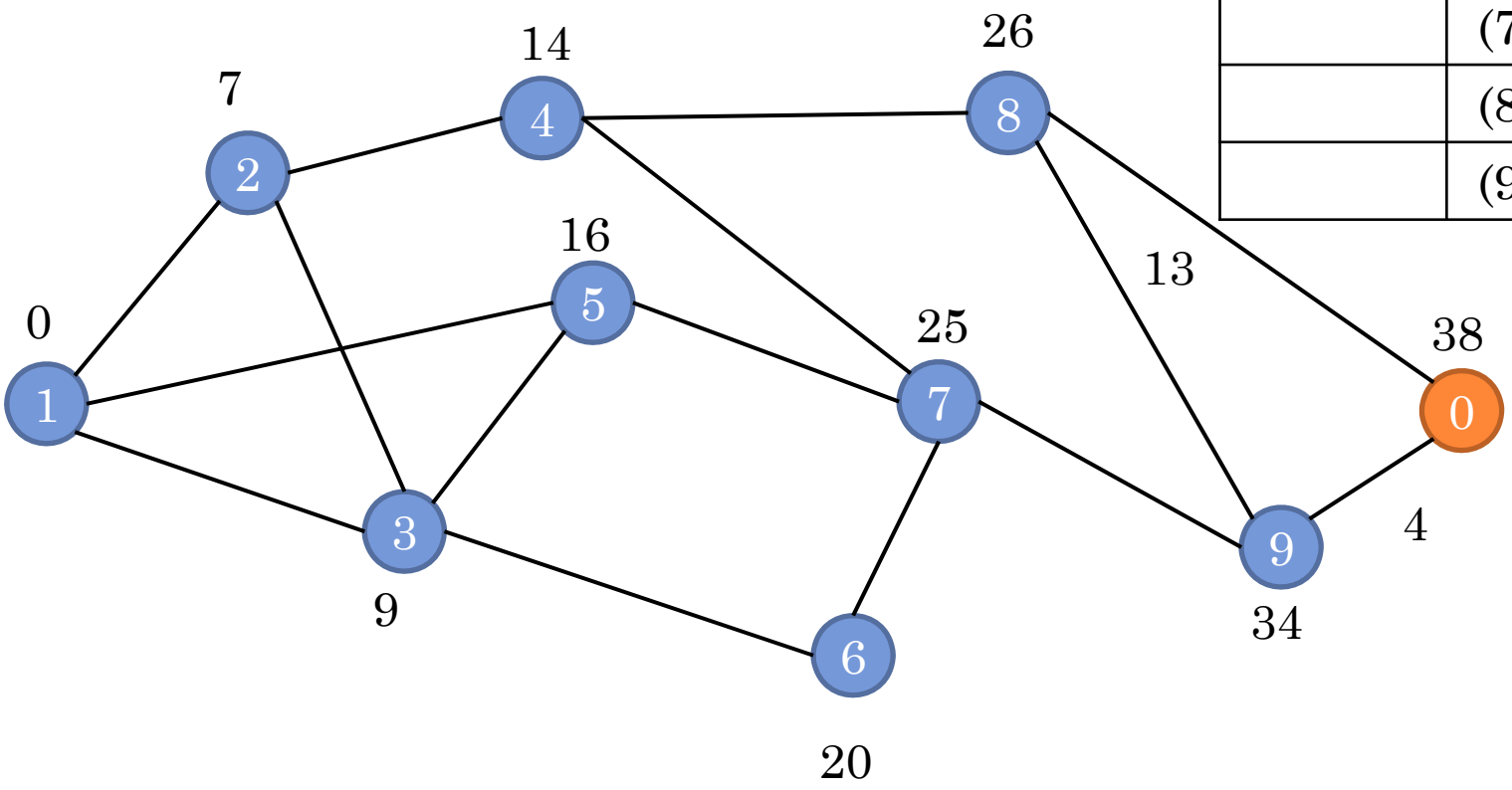
O	C
(9,7,34)	(1,-,0)
(0,8,39)	(2,1,7)
	(3,1,9)
	(4,2,14)
	(5,1,16)
	(6,3,20)
	(7,5,25)
	(8,4,26)



Planning

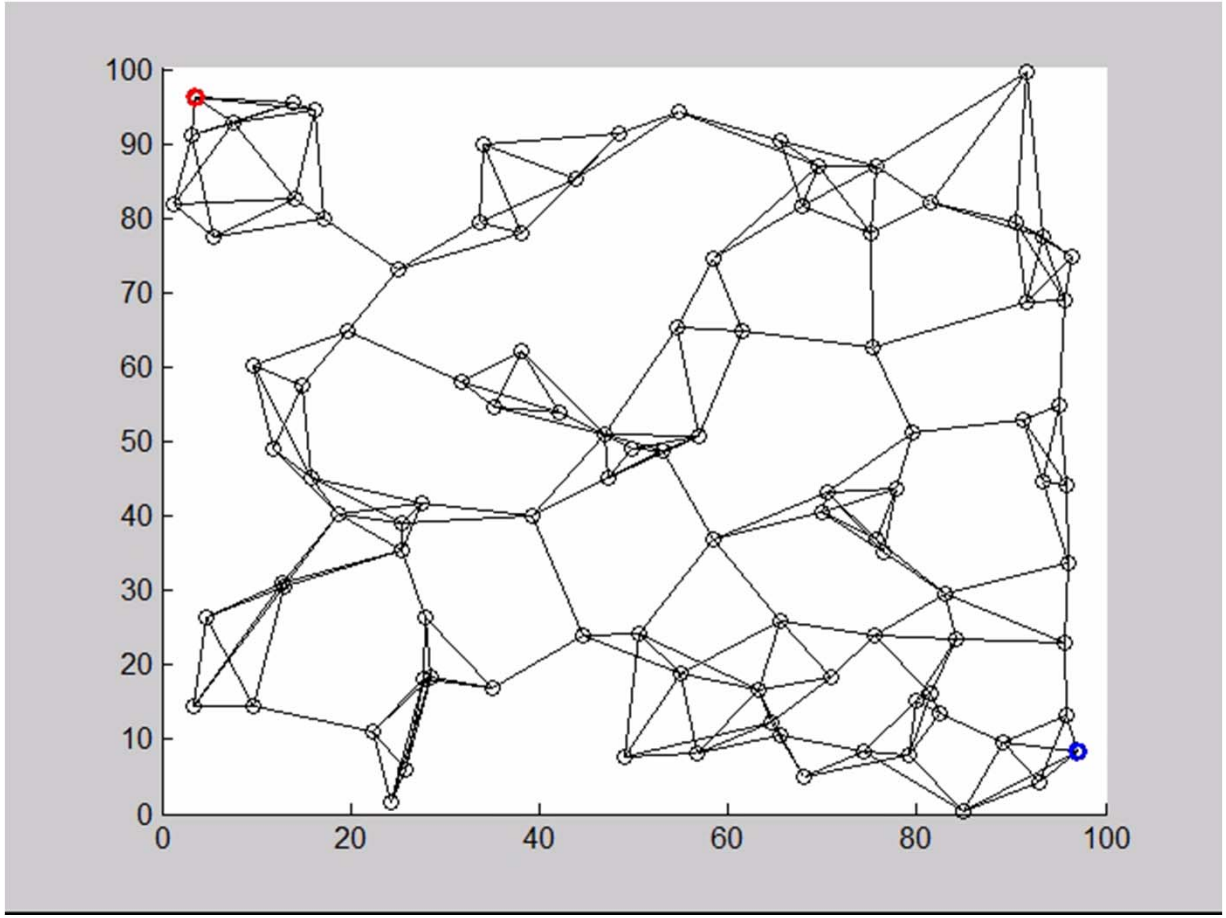
- Dijkstra's Search Algorithm
  - Stop when end node is current best node in open list

O	C
(0,9,38)	(1,-,0)
	(2,1,7)
	(3,1,9)
	(4,2,14)
	(5,1,16)
	(6,3,20)
	(7,5,25)
	(8,4,26)
	(9,7,34)



Planning

- Dijkstra's Example
  - 100 nodes, all connected to 4 closest neighbours



Planning

- Finding the shortest path over a graph
  - Dijkstra's algorithm
    - Start from starting node
    - Expand all links out of the node with lowest current cost
    - Find the next lowest current cost node, repeat previous step
    - Stop when end goal is closed, no other path can be shorter
  - A* Algorithm
    - Modified version of Dijkstra's
    - Rely on edge costs and cost to go heuristic
    - Pick most promising node at each step
    - Cost to go heuristic should never be greater than true cost
  - Can run all these algorithms from current location forward or from end point backward

PLANNING

- A* algorithm
  - While best node is not goal
    - Move best node from open set to closed set

$$f(n_{best}) \leq f(n), \forall n \in O$$

      - Store node, back pointer to previous node, current cost and lower bound cost
    - Add all adjacent nodes not currently in either set to the open set
      - Store node, current cost, lower bound cost and back pointer to n_{best}

$$O = \{O, A(n_{best}) \setminus (O \cup C)\}$$

    - For each node already in open set, update current cost, lower bound cost and back pointer if new path is shorter

$$\text{for all } n \in O \cap A(n_{best})$$

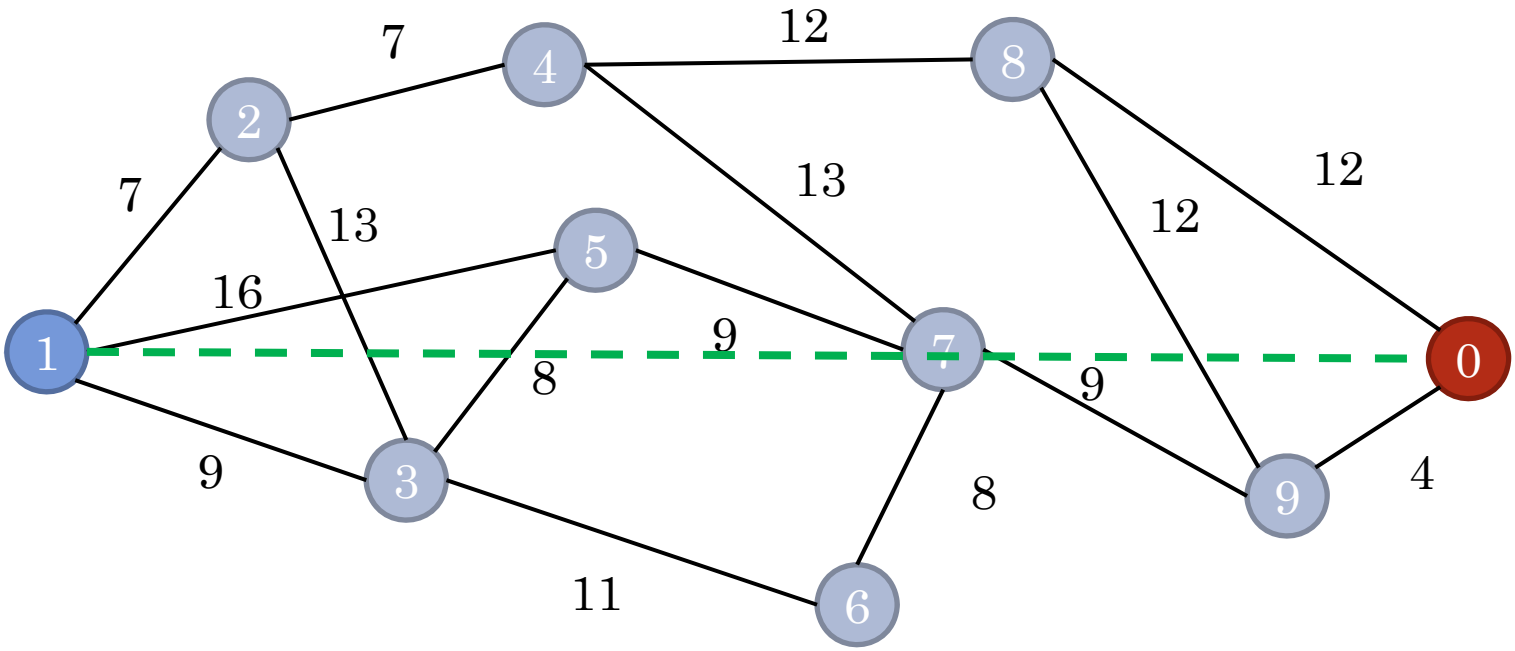
$$\text{if } (g(n_{best}) + c(n_{best}, n) + h(n) < f(n))$$

$$\text{backpoint to } n_{best}, \text{ update } f(n), g(n)$$

PLANNING

O	C
(1,-,33)	-

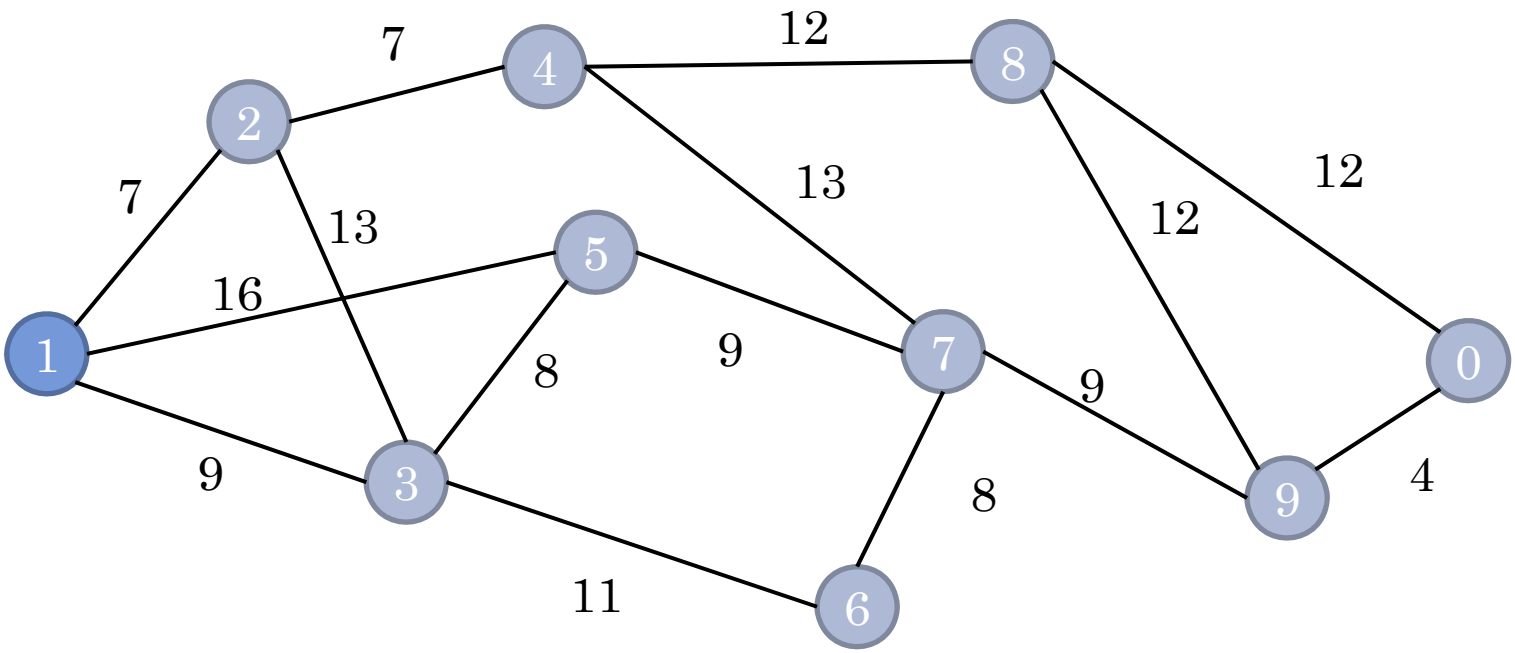
- Step 1
  - Add n_1 to O with a lower bound cost of 33



PLANNING

O	C
	(1,-,0)

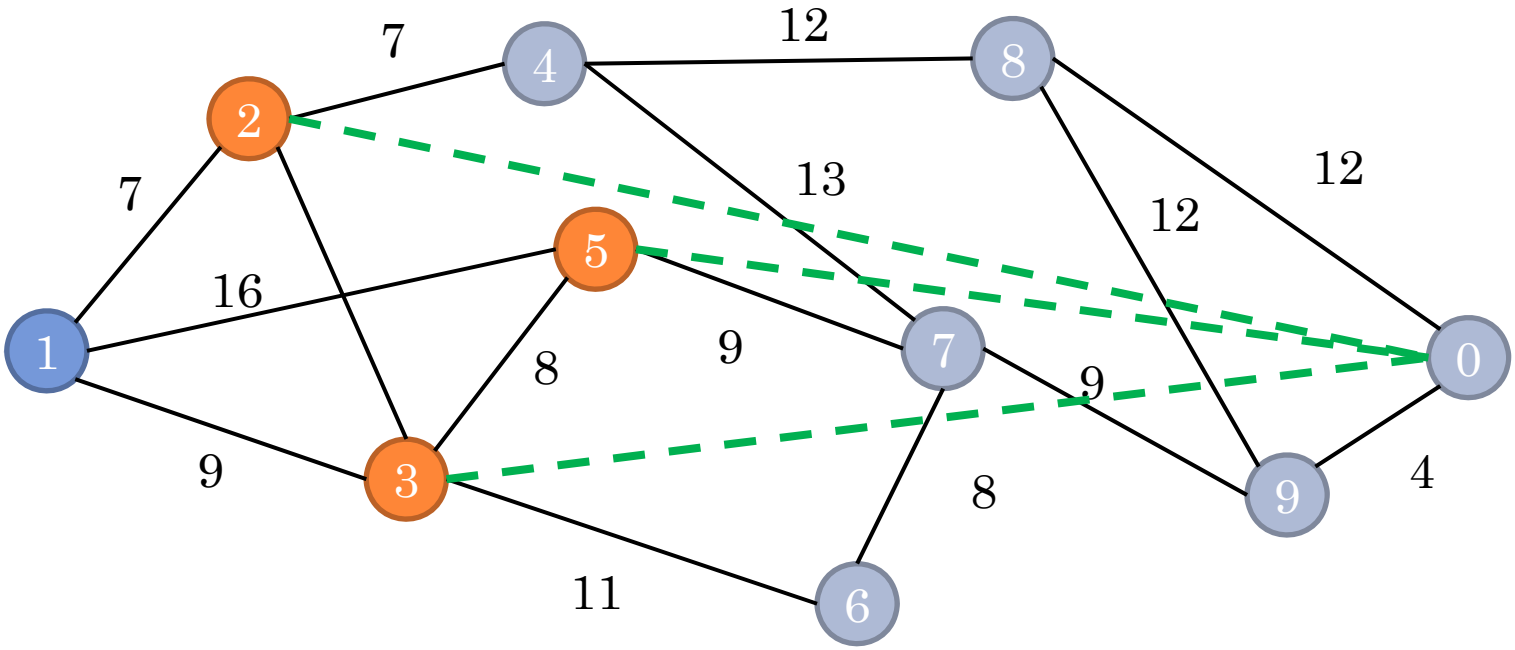
- Step 2
  - Take best node in O, move it to C, store current cost and back pointer (0,Null in this case)



PLANNING

O	C
(3,1,34)	(1,-,0)
(5,1,35)	
(2,1,36)	

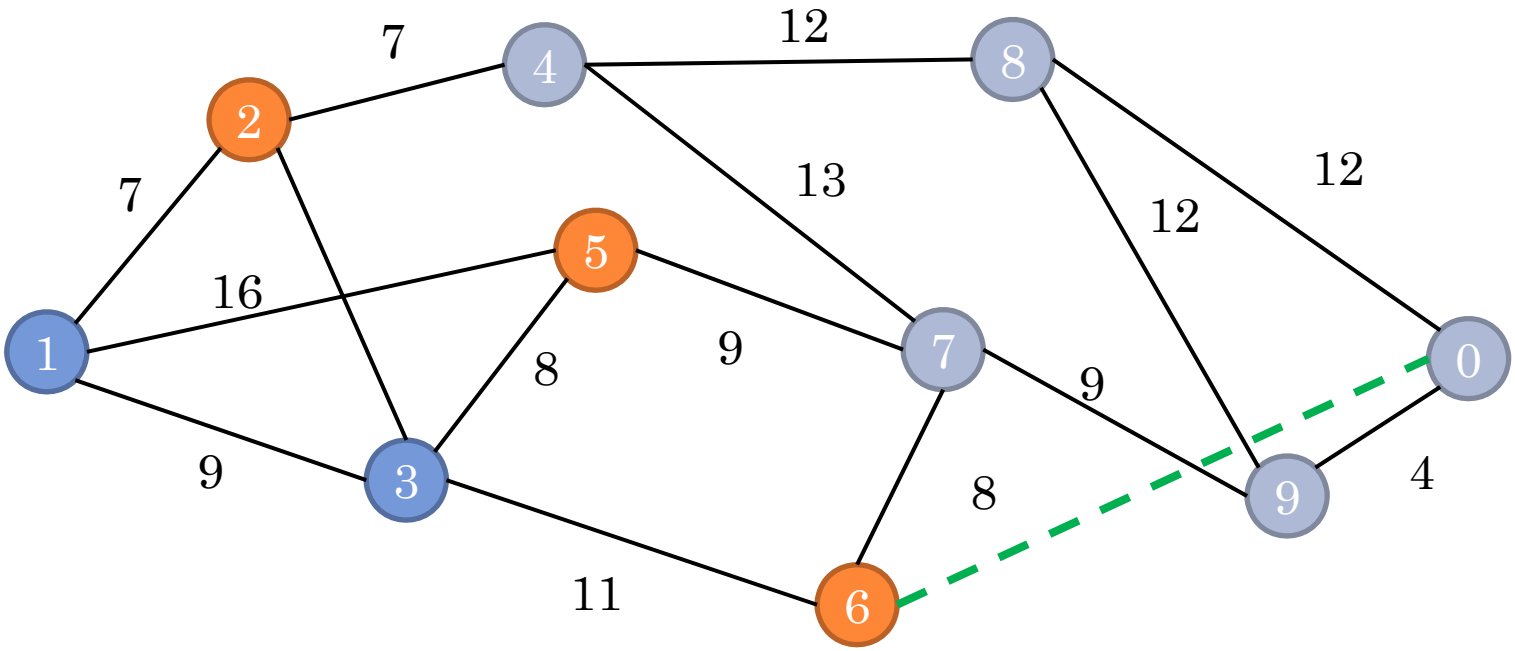
- Step 3
  - Add all nodes accessible from best node (1) to 0, ordered based on cost estimate. If node is already in O, update cost estimate and back pointer



PLANNING

O	C
(5,1,35)	(1,-,0)
(2,1,36)	(3,1,9)
(6,3,38)	

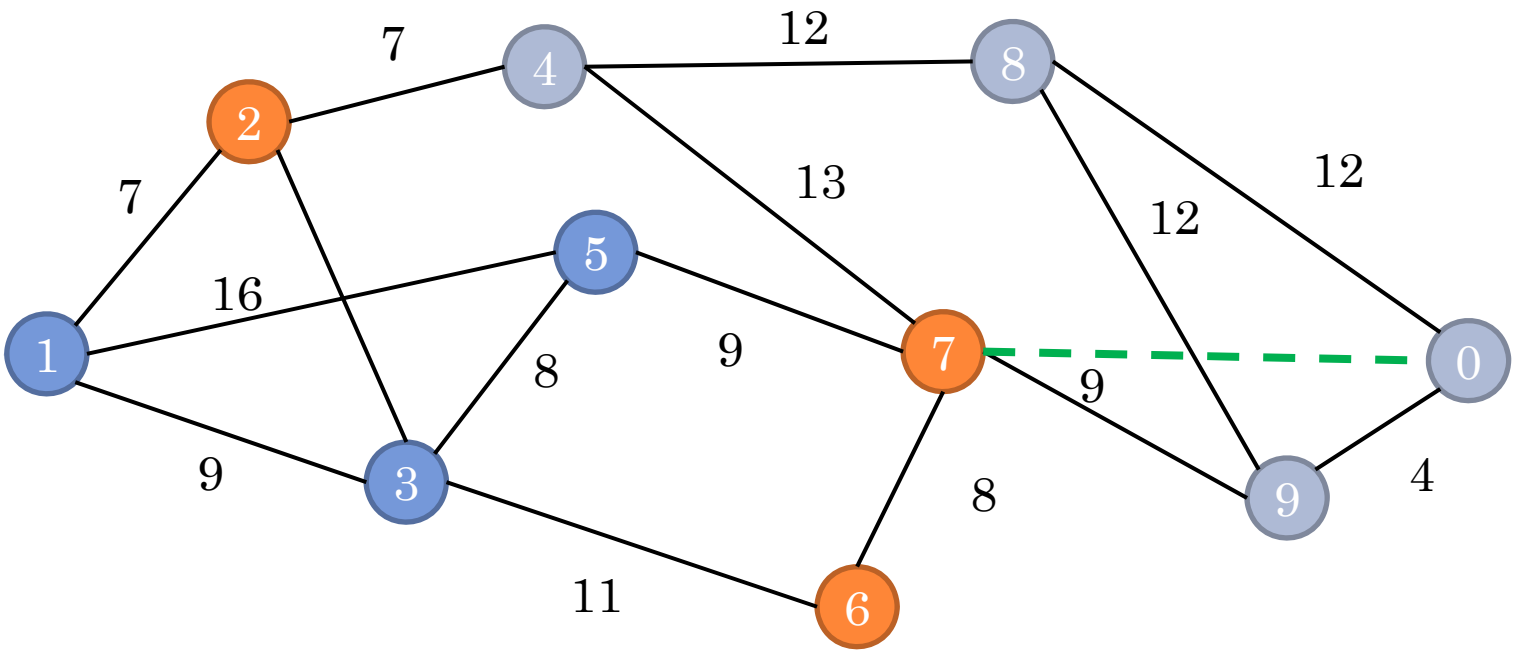
- Step 4: Repeat steps 2 and 3
  - Add n_6 to *0*
  - Cost of n_1-n_3-n_5 is greater than n_1-n_5, keep old cost



PLANNING

- Step 5
  - Add n_7 to *0*

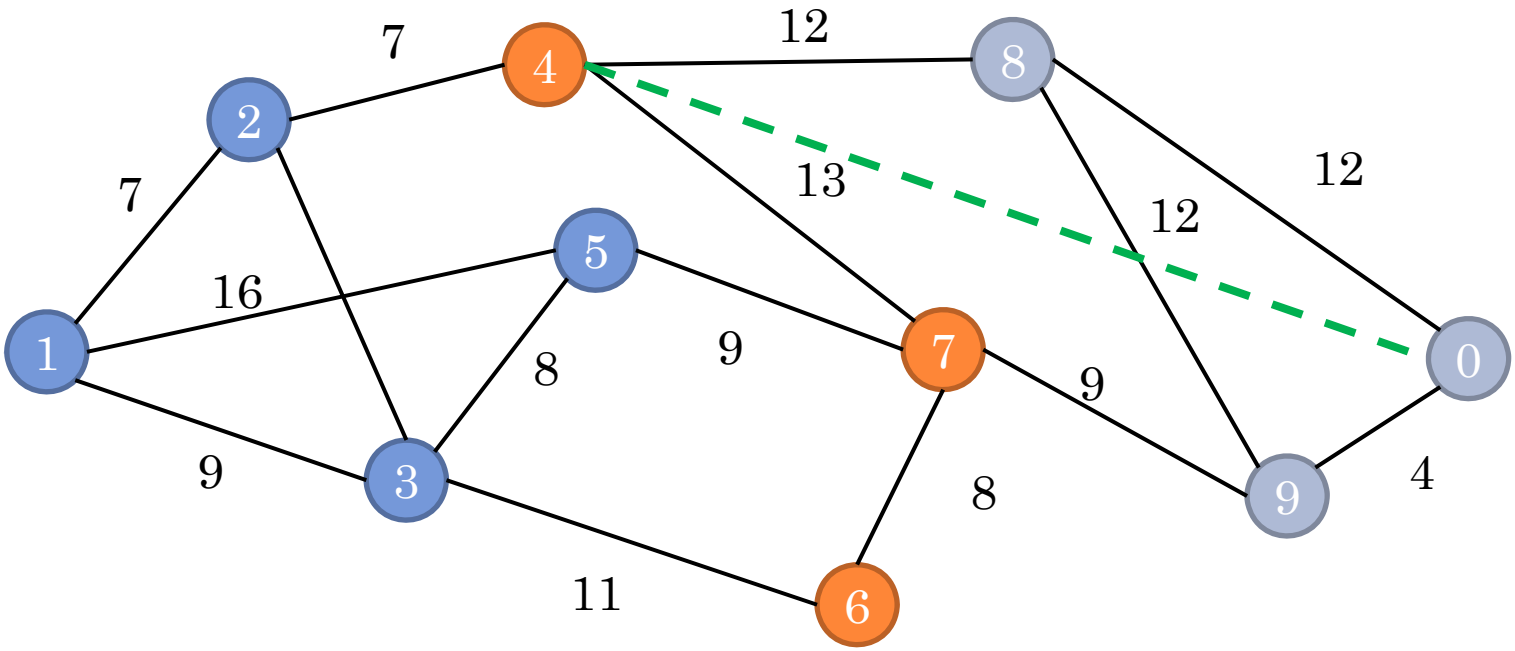
O	C
(2,1,36)	(1,-,0)
(7,5,37)	(3,1,9)
(6,3,38)	(5,1,16)



PLANNING

- Step 6
  - Add n_4 to *0*

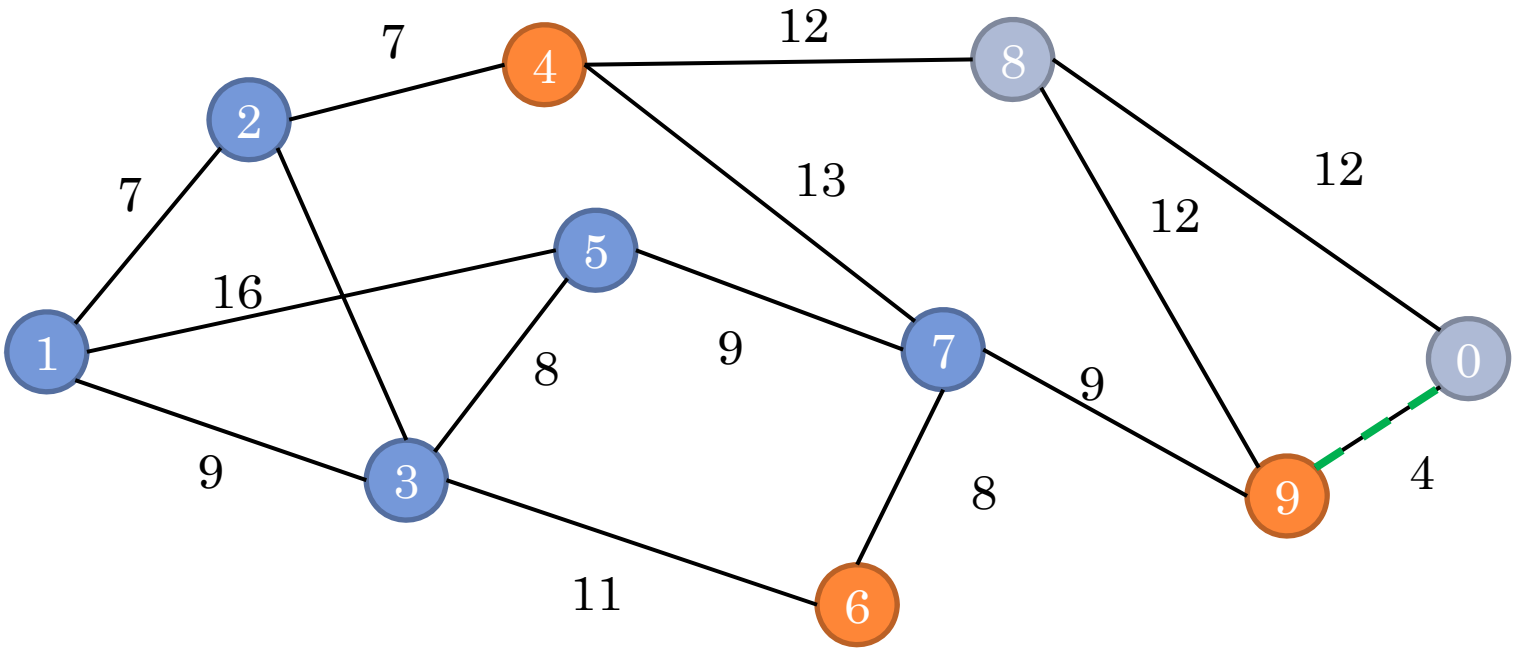
O	C
(7,5,37)	(1,-,0)
(6,3,38)	(3,1,9)
(4,2,39)	(5,1,16)
	(2,1,7)



Planning

- Step 7
  - Add n_9 to *O*

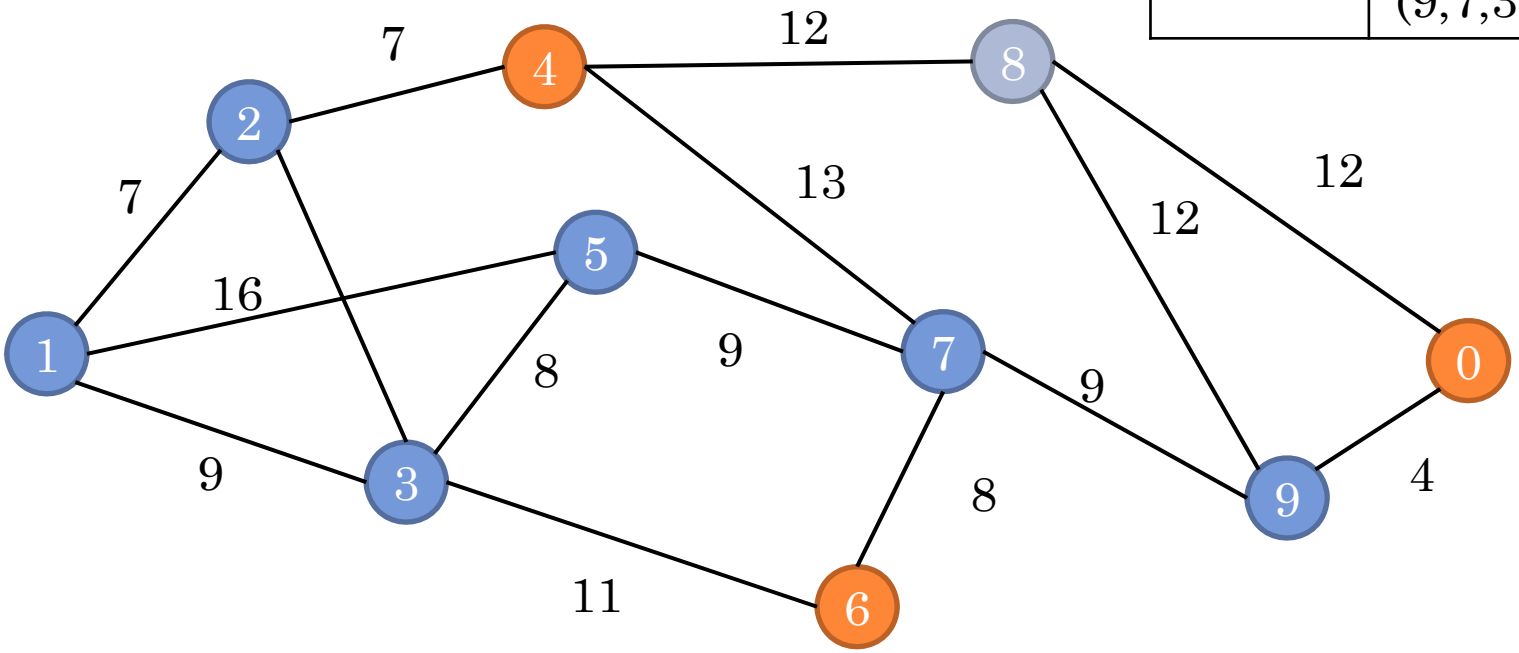
O	C
(9,7,38)	(1,-,0)
(6,3,38)	(3,1,9)
(4,2,39)	(5,1,16)
	(2,1,7)
	(7,5,25)



Planning

- Step 8
  - Add n_0 to *0*

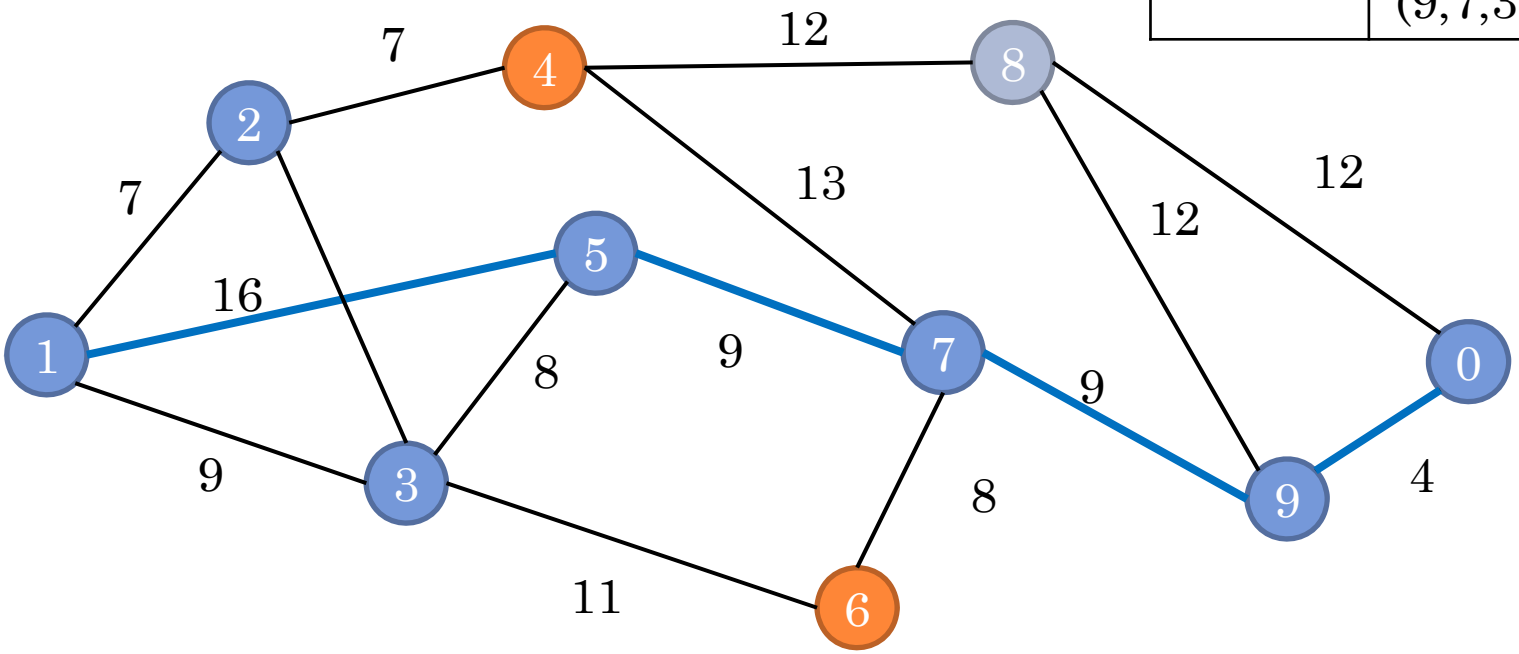
O	C
(0,9,38)	(1,-,0)
(6,3,38)	(3,1,9)
(4,2,39)	(5,1,16)
	(2,1,7)
	(7,5,25)
	(9,7,38)



Planning

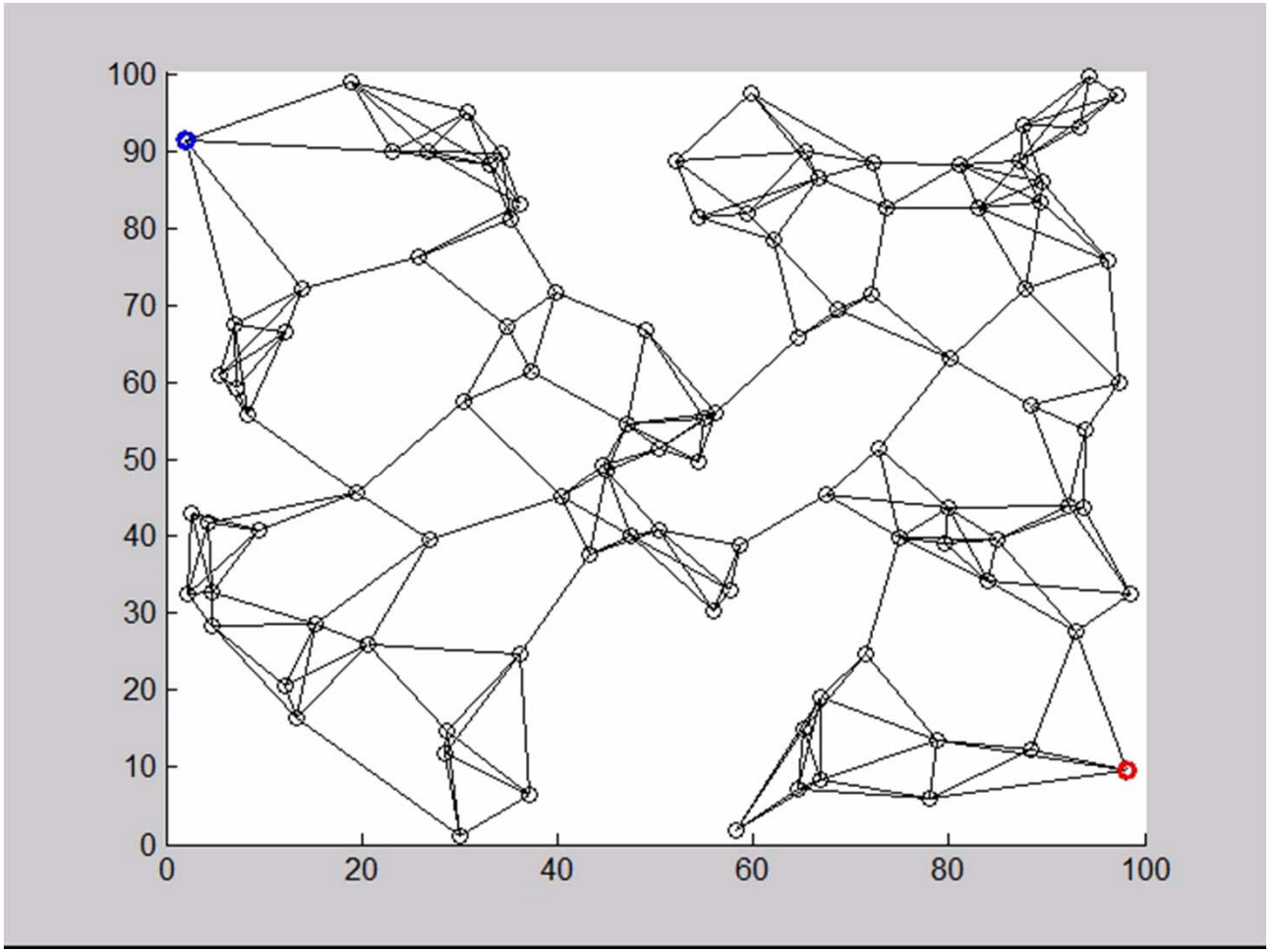
- Step 9
  - Done, node 0 is best node in open list

O	C
(0,9,38)	(1,-,0)
(6,3,38)	(3,1,9)
(4,2,39)	(5,1,16)
	(2,1,7)
	(7,5,25)
	(9,7,38)



Planning

- A* Example:
  - 100 nodes, all connected to 4 closest neighbours



Outline

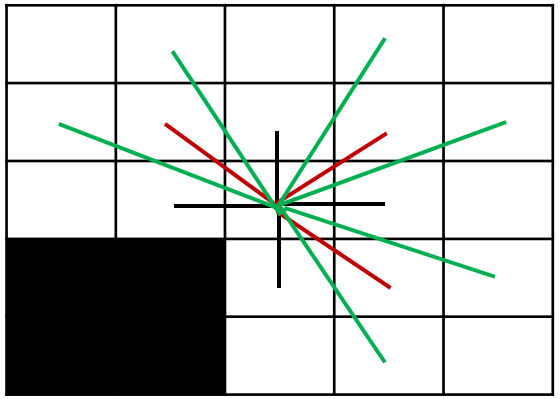
- Planning Concepts
- Reactive Motion Planning Algorithms
  - Bug
  - Potential Fields
  - Trajectory Rollout
- Graph Based Motion Planning
  - Finding paths on graphs
    - Wavefront
    - Dijkstra, A*, D*
  - Generating Graphs from environments
    - Visibility Graphs
    - Decompositions

PLANNING

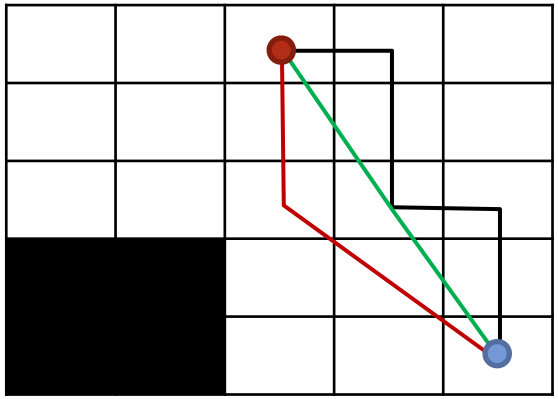
- How to make a map into a graph
  - Deterministically
    - Occupancy Grid-based Graph
    - Visibility Graph
    - Cell Decomposition
    - Voronoi Diagram
    - Constrained Delaunay Triangulation
  - Randomly
    - Probabilistic roadmaps (PRMs)

Planning

- Occupancy grid to graph
  - Each cell is a node
  - Can connect to 4,8 or 16 nearest neighbours if not occupied
  - Edge length either 1 unit or true distance
    - Wavefront or Dijkstra/A*

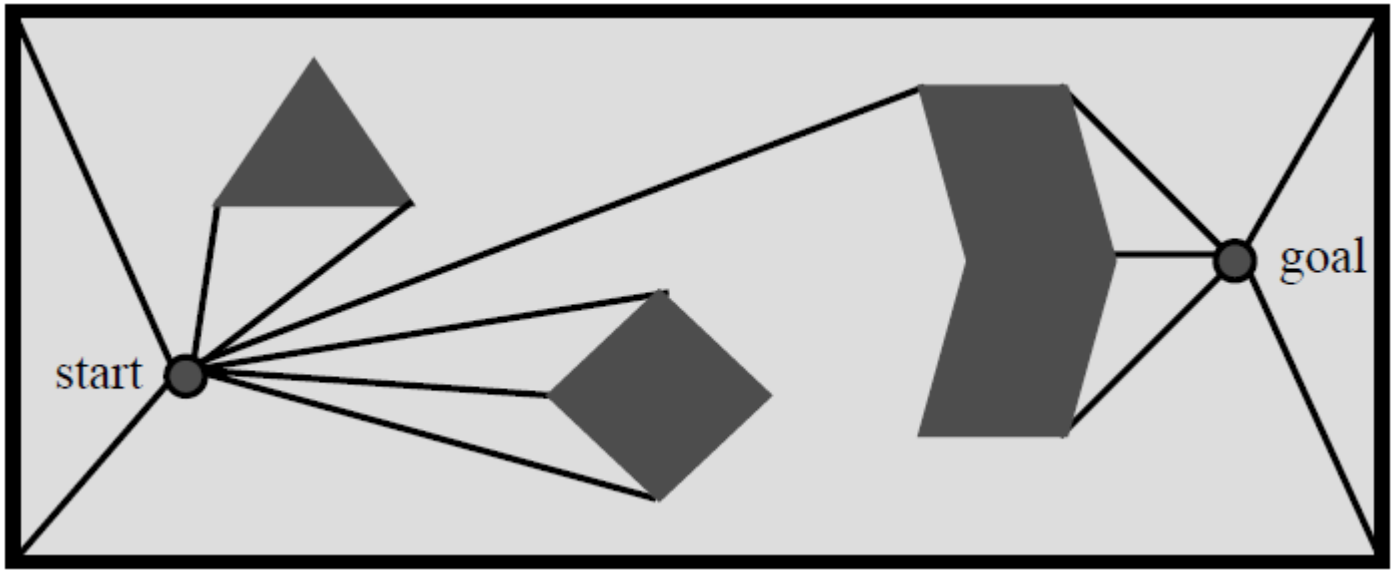


  - The more connections, the harder the search, but the more direct the path
    - Memory limitations
    - Time complexity
    - For small 100x100 grid
      - 10,000 nodes
      - 20,000, 40,000, 80,000 edges



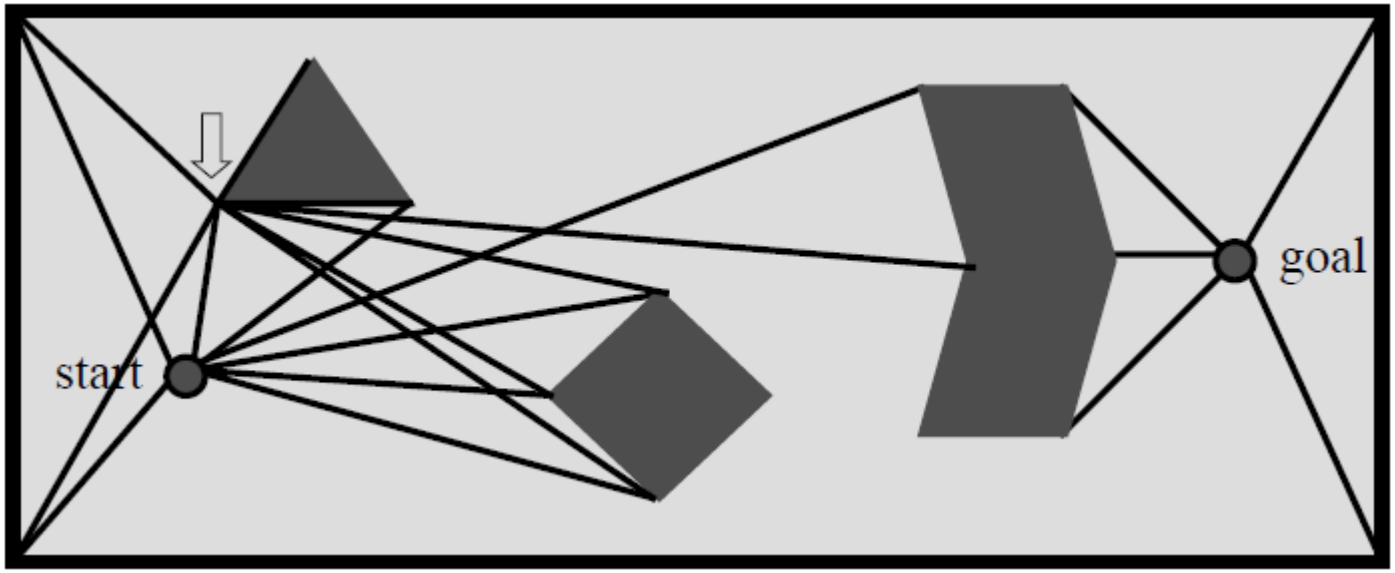
PLANNING

- Visibility Graph
  - If 2D map is defined as a polygon with polygonal obstacles (holes)
    - Connect all vertices in map to create a visibility graph
      - Line of sight between each vertex pair
      - Remove all edges that intersect obstacles
  - Step 1: Connect start and end point to all visible vertices



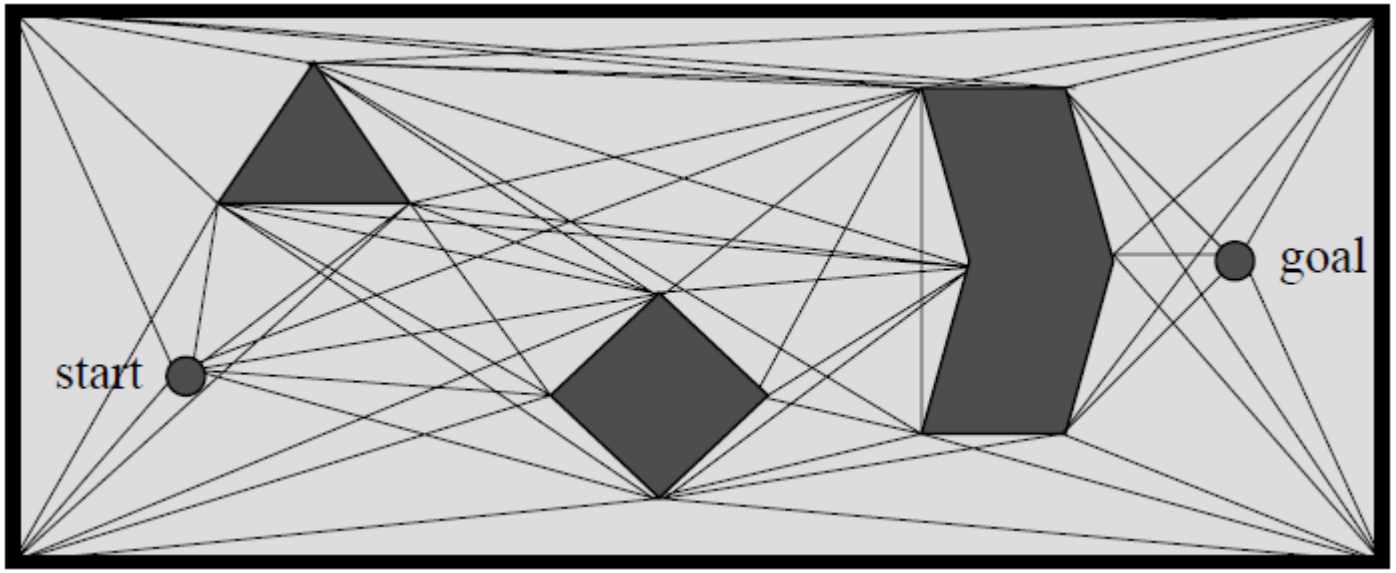
PLANNING

- Visibility graph
  - Step 2: For each obstacle vertex reached in step 1, add all its connections, including connections along obstacle edges



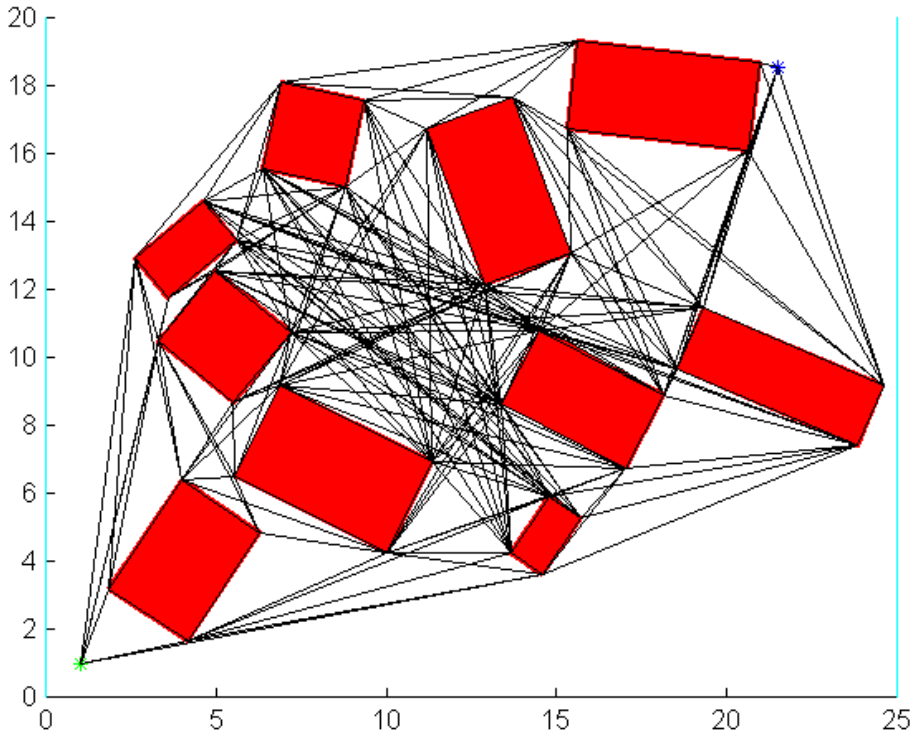
Planning

- Visibility Graph
  - Step 3: Repeat until no new edges are added



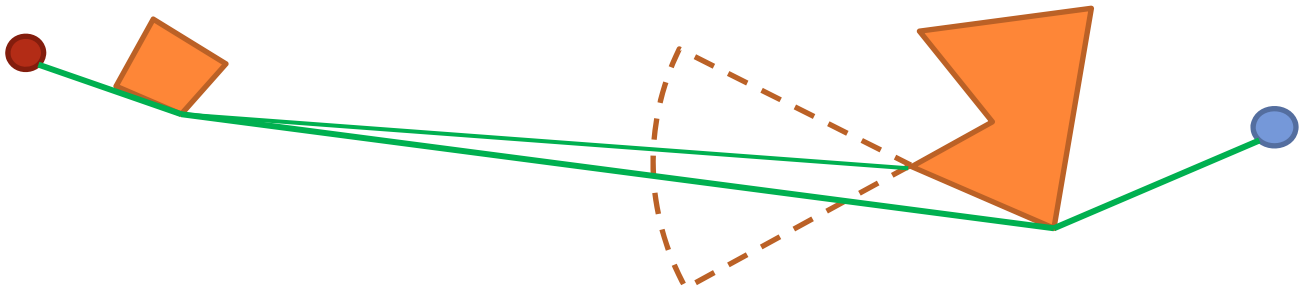
Planning

- Example of Visibility Graph
  - Brute force: $O(n^3)$
    - For each connection, check n edge intersections
  - 10 Convex obstacles
  - 218 links
  - 4 seconds

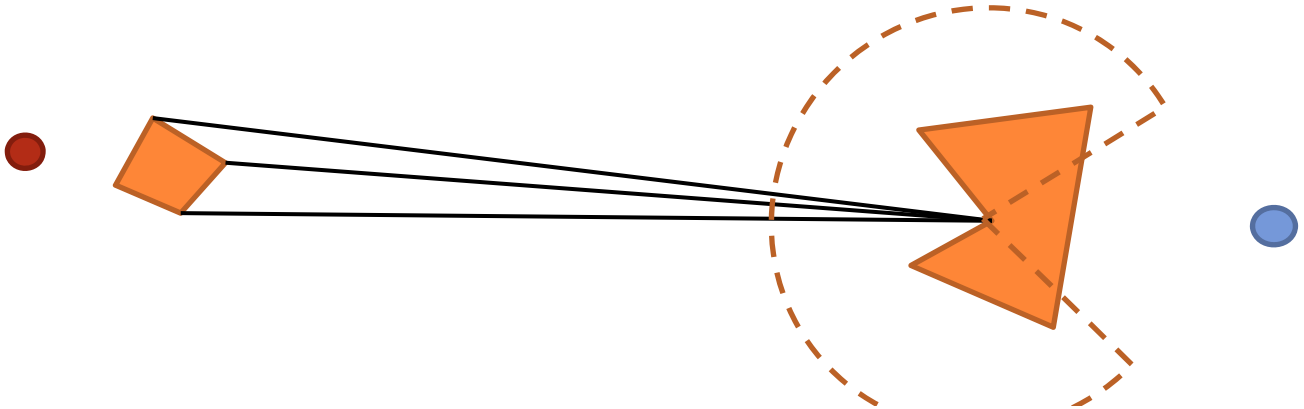


Planning

- Visibility graph
  - Can eliminate many unnecessary edges
    - All edges that head into obstacle
      - Nodes in regions defined by convex nodes can also be ignored

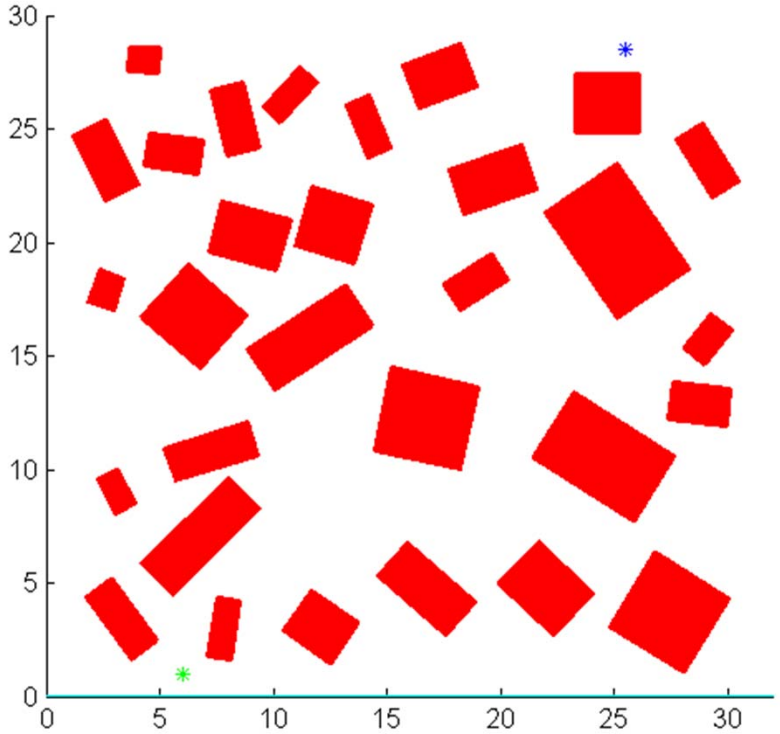


      - As a result, concave obstacle nodes can be ignored



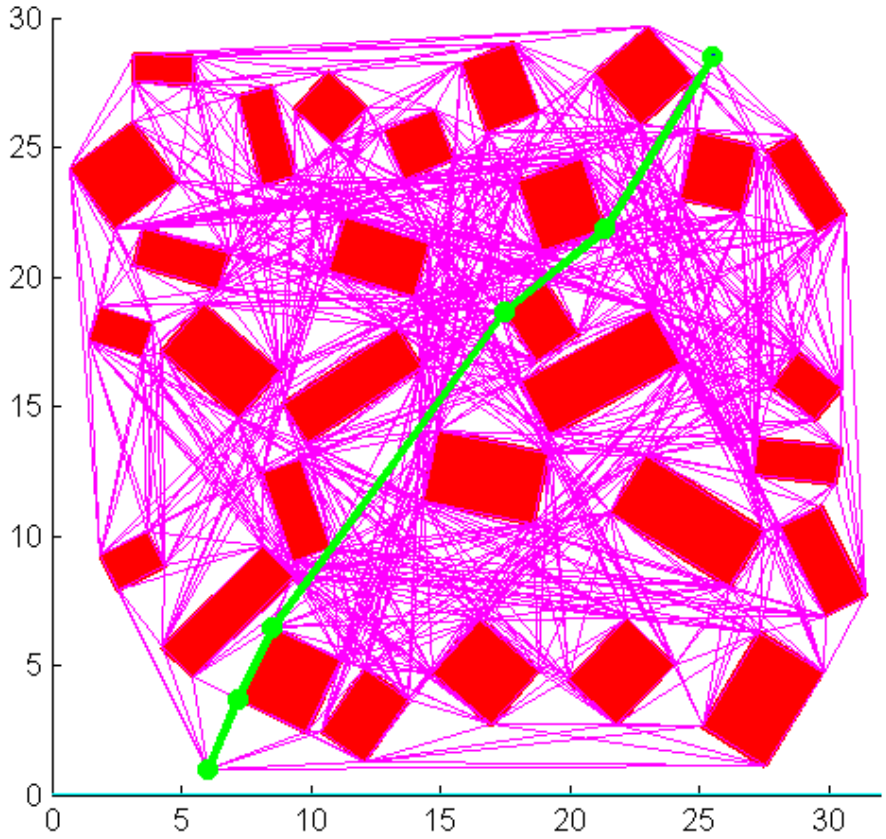
PLANNING

- Example – 2D path planning
  - 30 Obstacles
  - Guaranteed shortest path
  - Many collision checks
    - Connecting all nodes requires 7503 edge collision checks
  - Resulting network has
    - 122 nodes
    - 976 edges



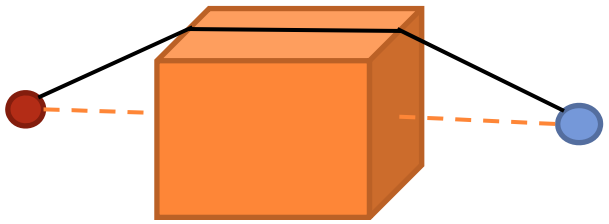
Planning

- Example – 2D path planning
  - Brute Force Runtime: 30 s



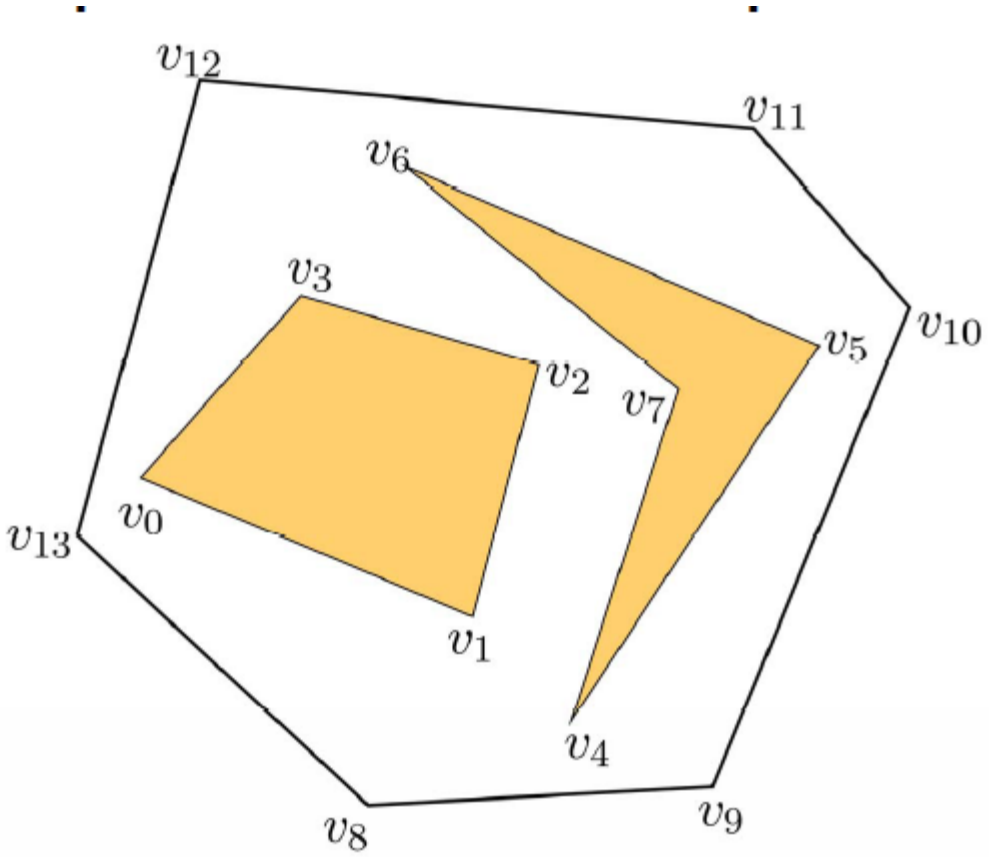
Planning

- Visibility Graph
  - Pros
    - Guaranteed to find shortest path
    - Fairly quick in 2D
  - Cons
    - Passes too close to obstacles
    - Requires nodes and edges view of the world
    - Not possible in 3D



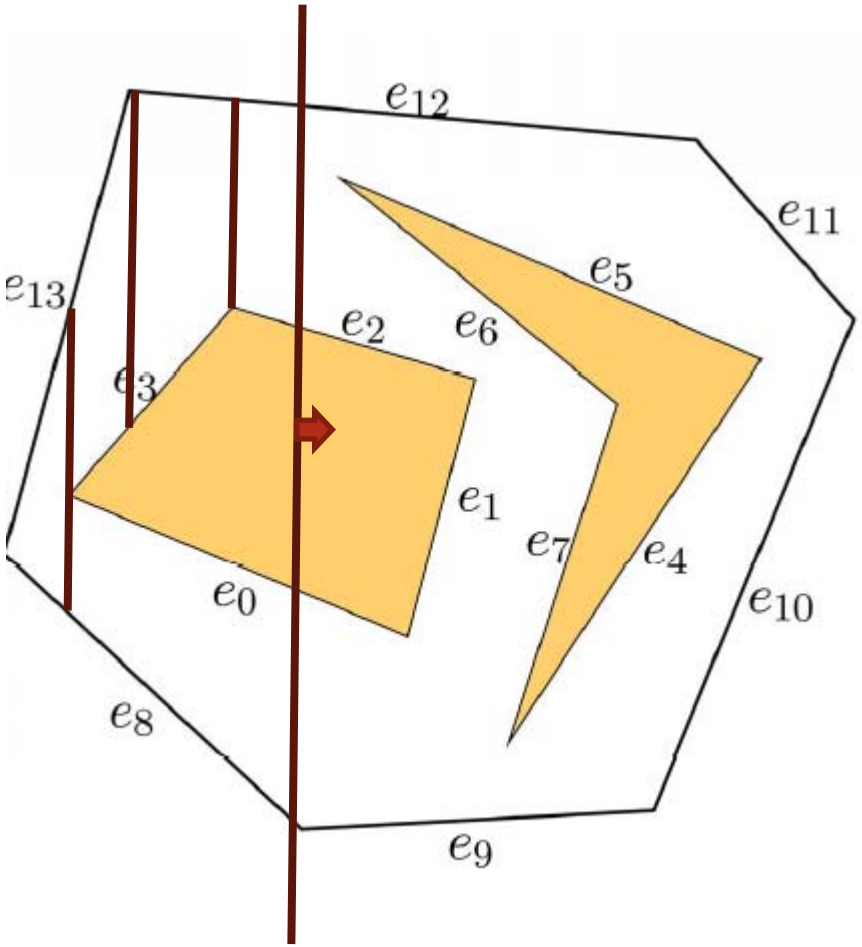
PLANNING

- Trapezoidal decomposition
  - 2D map cut vertically at each obstacle vertex



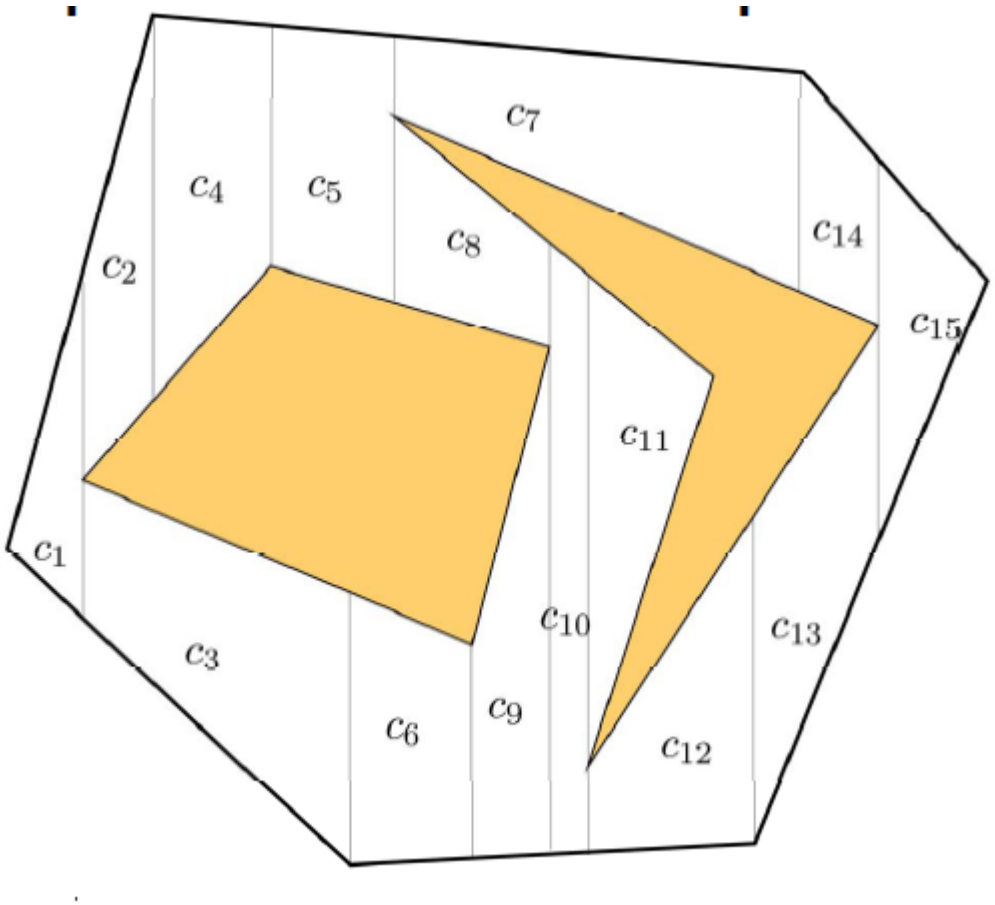
PLANNING

- Trapezoidal Decomposition



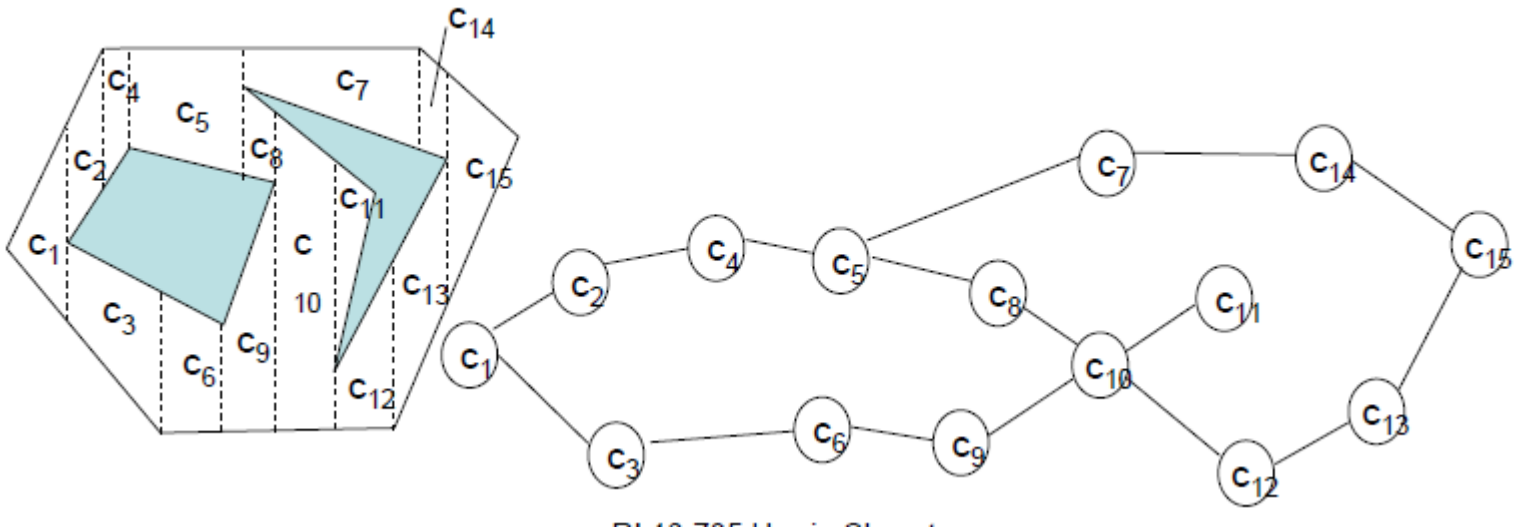
PLANNING

- Trapezoidal Decompositon



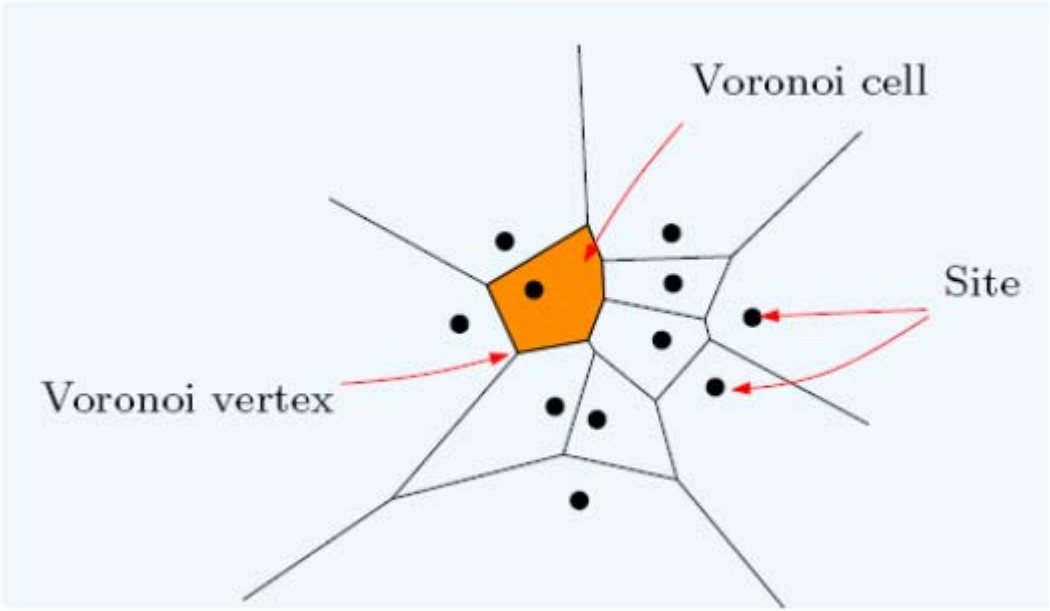
PLANNING

- Topological graph from decomposition
  - Create map by connecting adjacent open cells
    - Adjacency graph
  - Can connect cell centroids to form path (may intersect obstacles)
  - Distance between cells is unclear



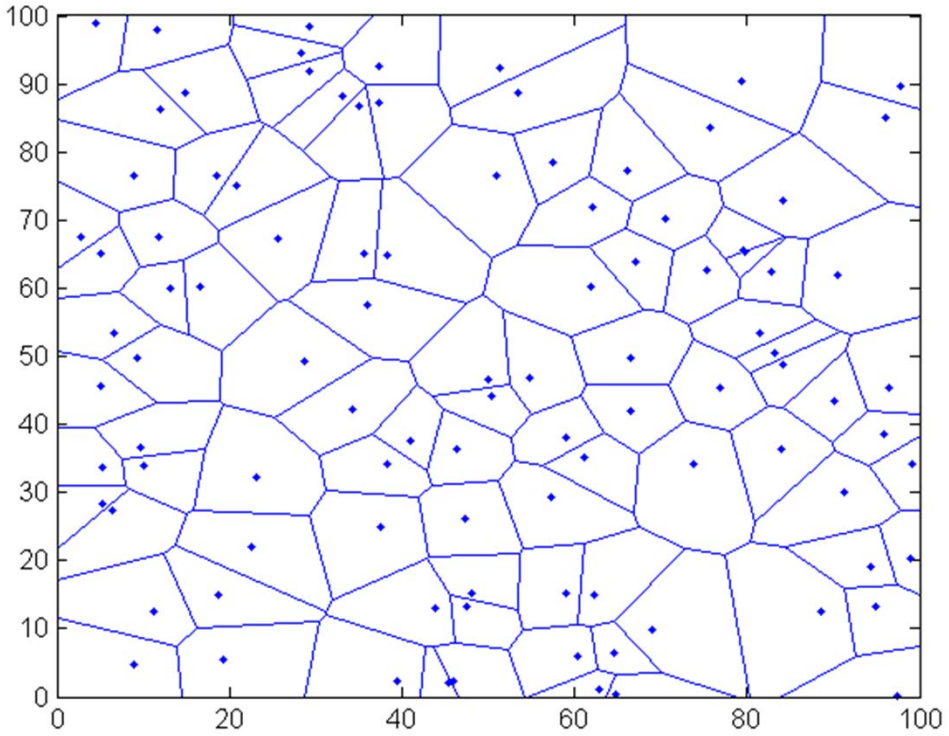
PLANNING

- Voronoi Diagram
  - An alternative that does not find the shortest path, but perhaps the "safest" path
    - Each edge is equidistant between two points
    - Results in paths that are furthest away from points



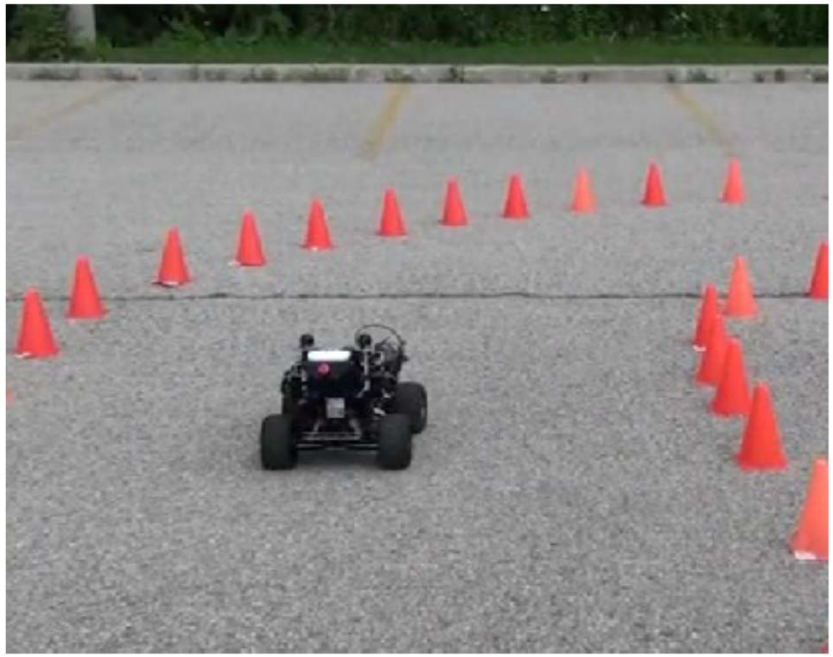
PLANNING

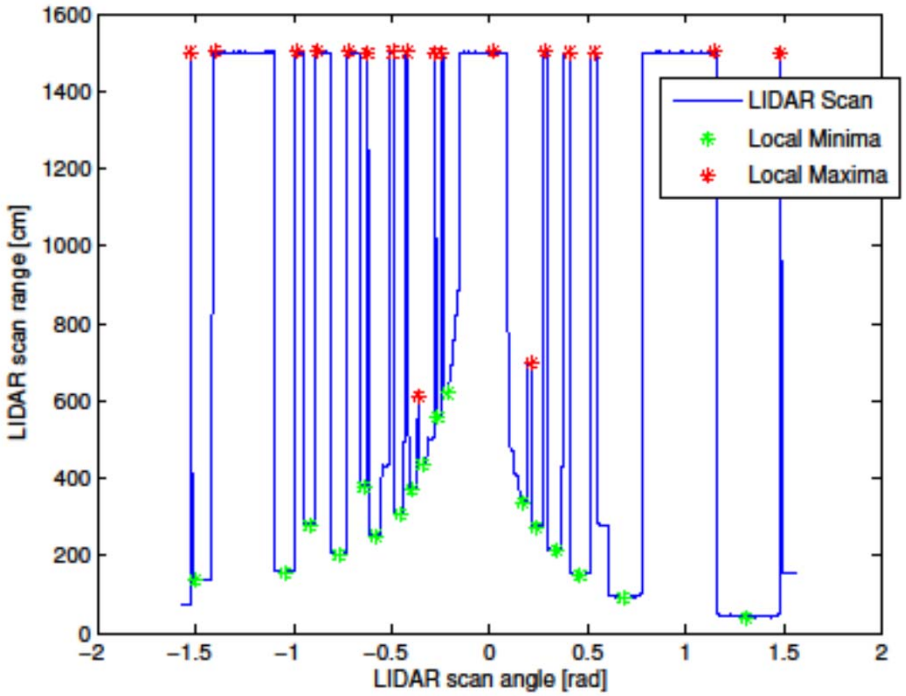
- Voronoi diagrams in Matlab
  - Very fast algorithm, relies on qhull software
    - Cannot handle non-point obstacles



Planning

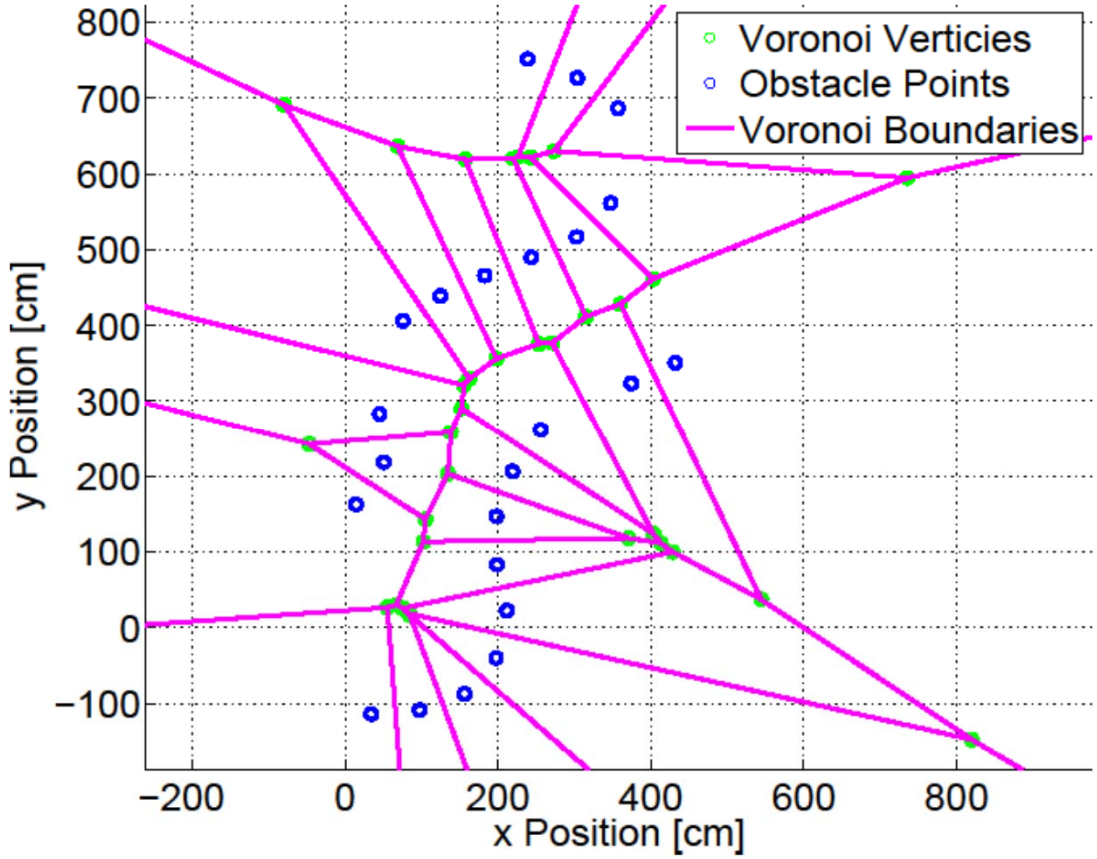
- Voronoi Diagrams in Robot Racing Planner
  - Detect pylons through peak detect algorithm





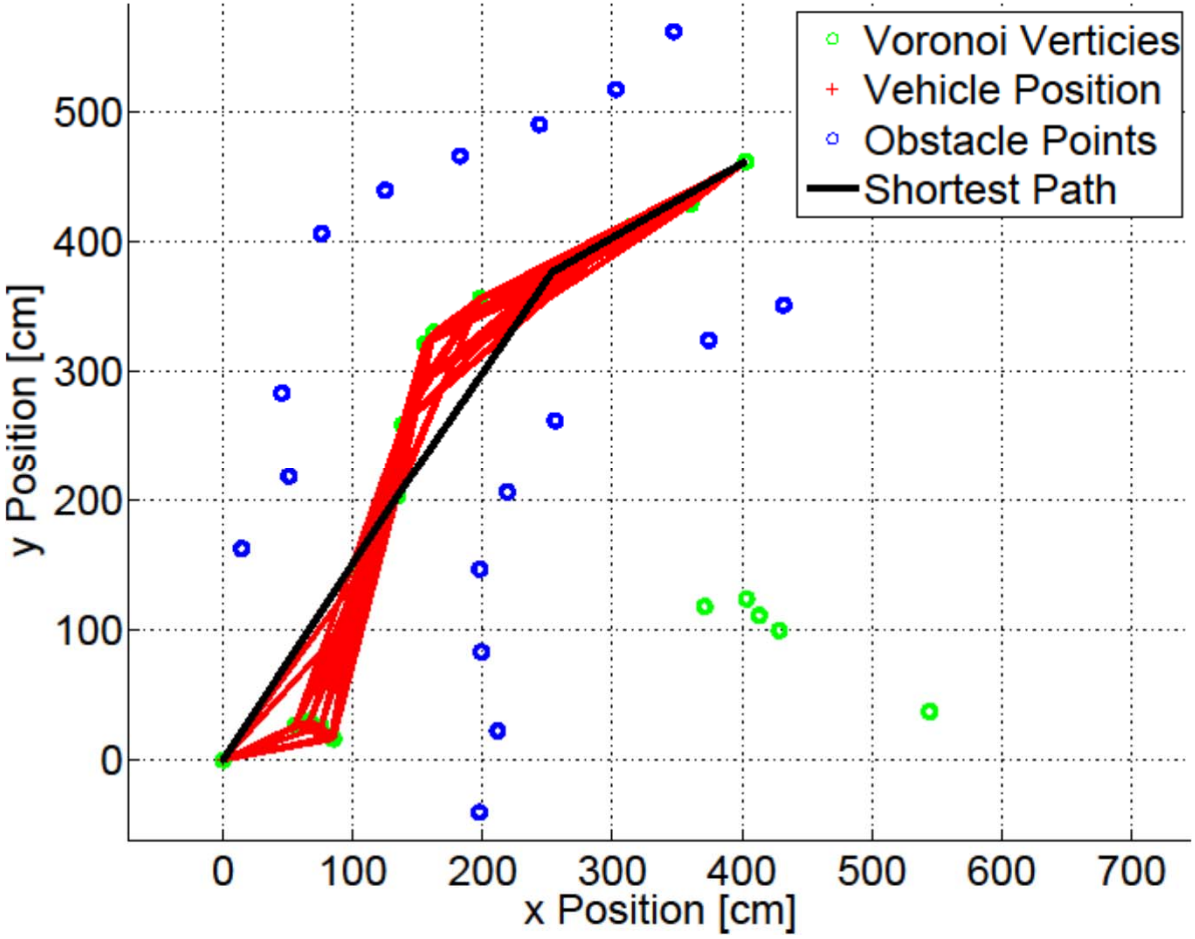
Planning

- Voronoi Diagrams in Robot Racing Planner
  - Create Voronoi diagram, connect graph, apply A*



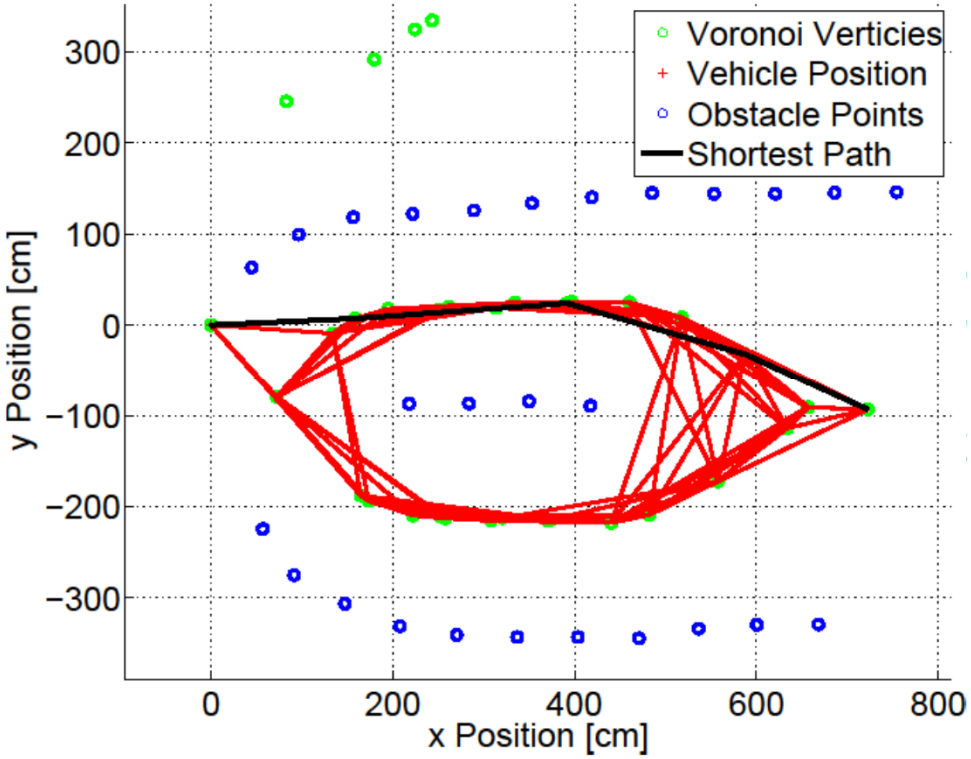
Planning

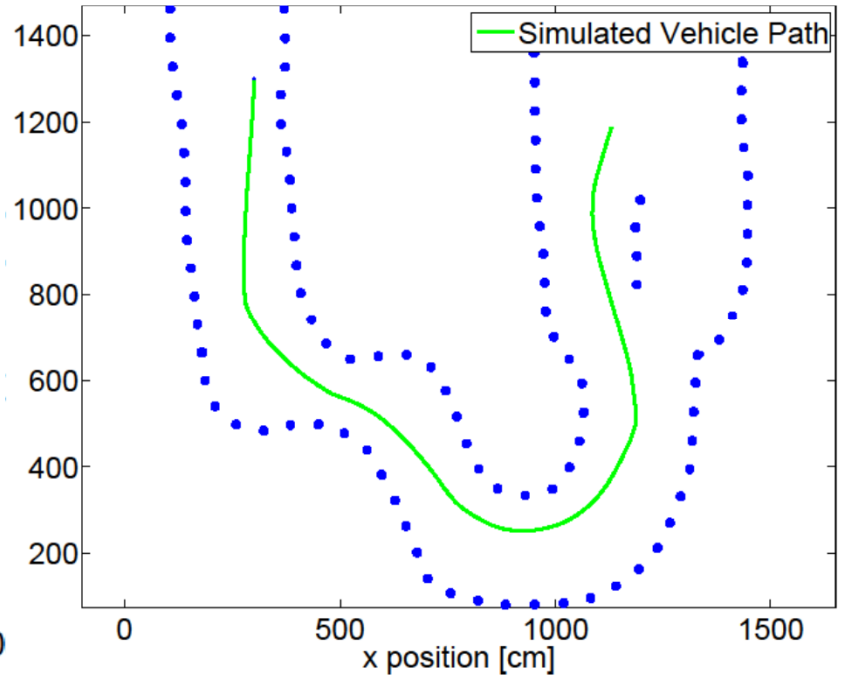
- Voronoi Diagrams in Robot Racing Planner
  - Connect graph using bounding box on obstacles, apply A*



PLANNING

- Voronoi Diagram in Robot Racing Planner
  - Simulation results





International Autonomous Robot Racing Competition 2010

Outdoor Testing

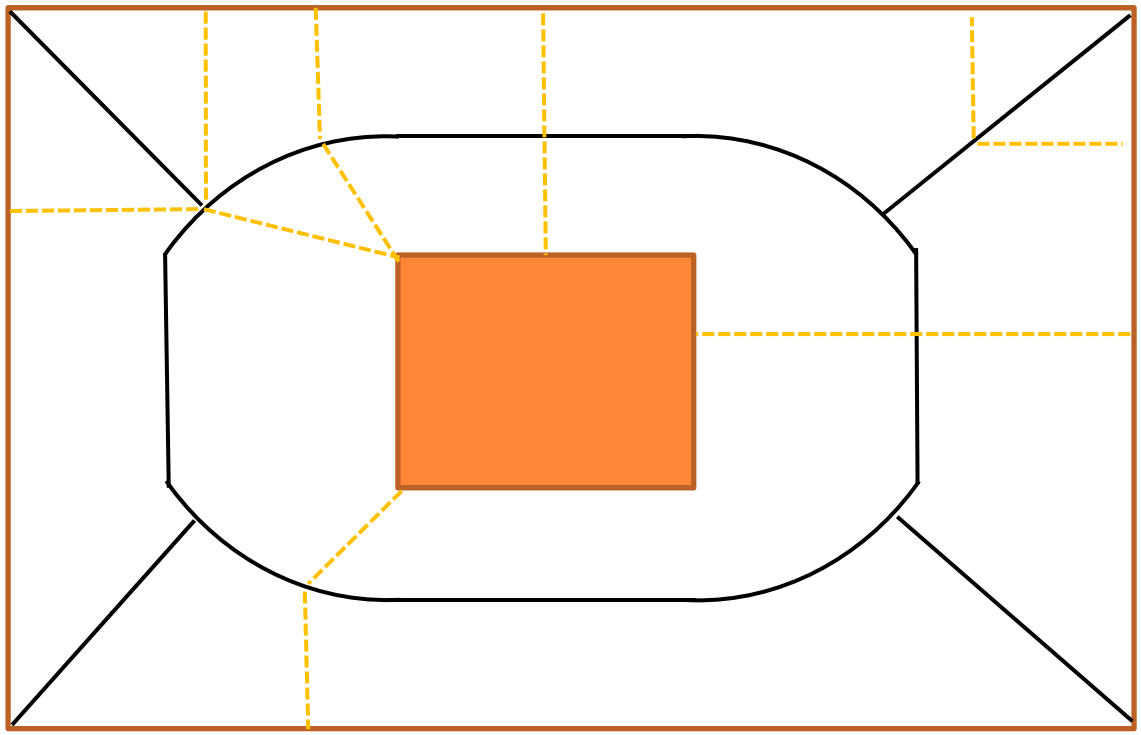
Demo 2

Static Obstacle Avoidance
using the
Trajectory Rollout Algorithm

Extra Slides

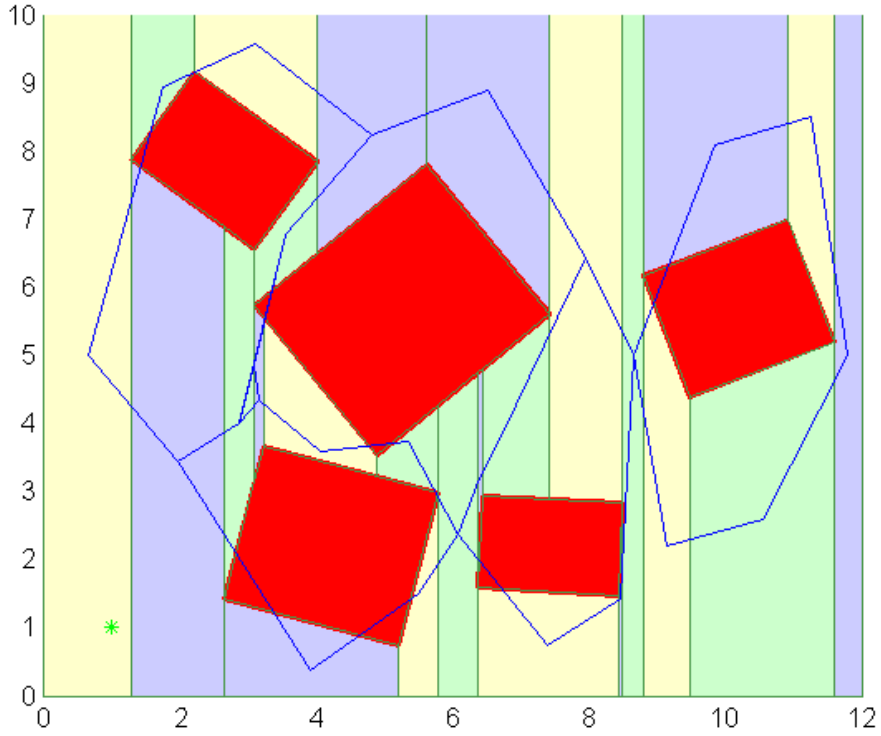
Planning

- Generalized Voronoi Diagram
  - Uses distance to object function (same as potential fields)
    - Find equidistant points between two obstacles
    - For polygonal obstacles, results in lines, ellipse segments



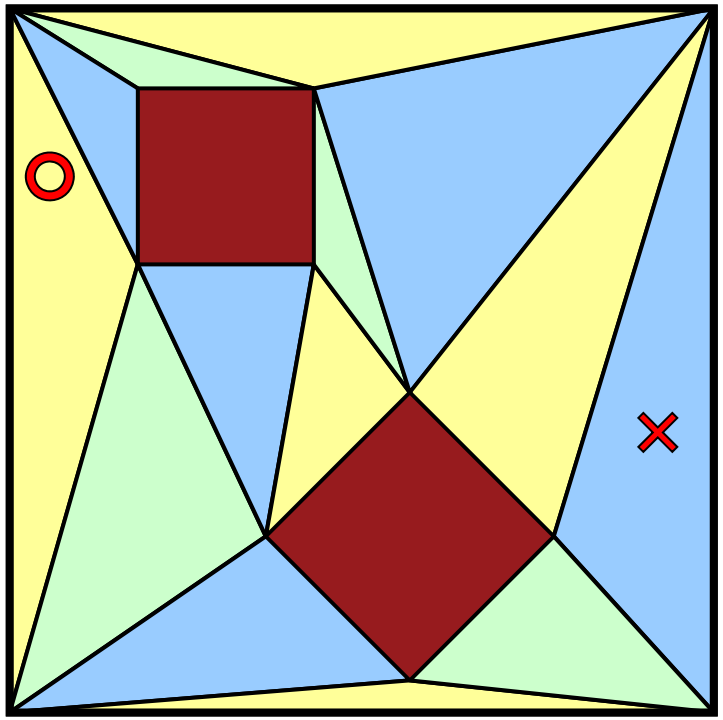
Planning

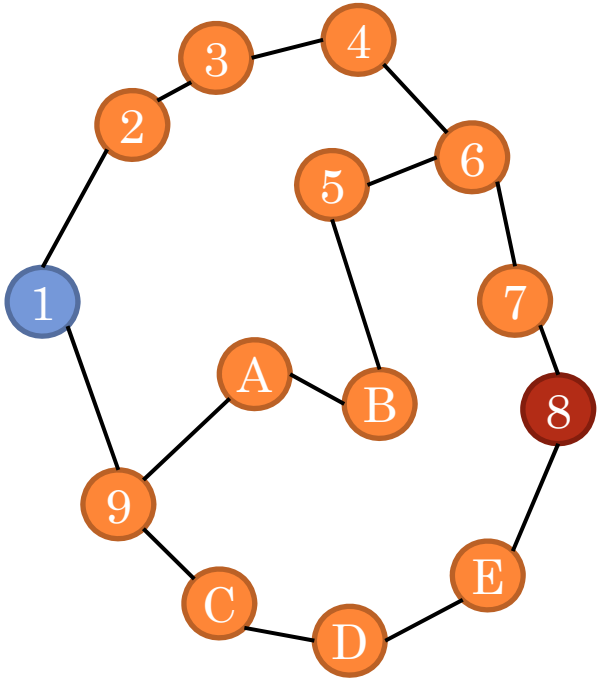
- Example
  - Trapezoid centroids connected in a graph
  - Graph represents connectivity of space, not navigable paths, utility of shortest path is therefore dubious



Planning

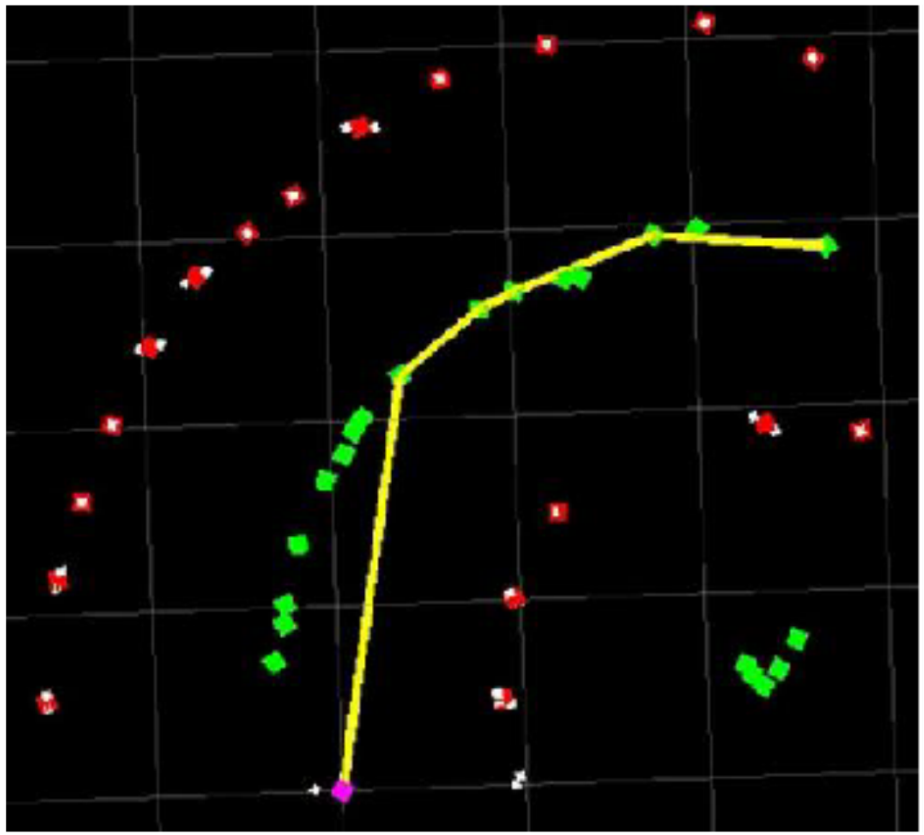
- Constrained Delaunay Triangulation
  - Complex algorithm, not often used, but interesting





Planning

- Voronoi Diagram in Robot Racing Planning
  - Competition results, success!



PLANNING

- D*
  - Dynamic A* algorithm
  - Adapted to be finite horizon, replan locally with new link information
  - Intended for robots that uncover new information as they travel
  - Solve for a path from start to end using A* from end to start
  - If new path length info becomes available
    - Affected nodes are marked Raised
    - All downstream nodes also marked raised, until all nodes that can be affected by the change are marked
    - New costs are assigned using the usual update, except that if a node cost can be reduced, it is marked Lowered, and all upstream nodes are improved
  - The result is a sequences of downstream and upstream waves updating the costs for only those nodes affected by the new information

  - Anthony Stentz "**The Focussed D* Algorithm for Real-Time Replanning",** In Proceedings of the International Joint Conference on Artificial Intelligence, August 1995
    - See Choset et al. Appendix H for summary