6.141:
Robotics systems and science
Lecture 10: Motion Planning III

Lecture Notes Prepared by N. Roy and D. Rus

EECS/MIT

Spring 2012

Reading: Chapter 3, and Craig: Robotics

http://courses.csail.mit.edu/6.141/

Challenge: Build a Shelter on Mars

Last time we saw

- C-space: Minkowski sum
- Motion Planning with Visibility Graphs, cell decomposition, PRMs, RRTs

Today: Implementation Issues

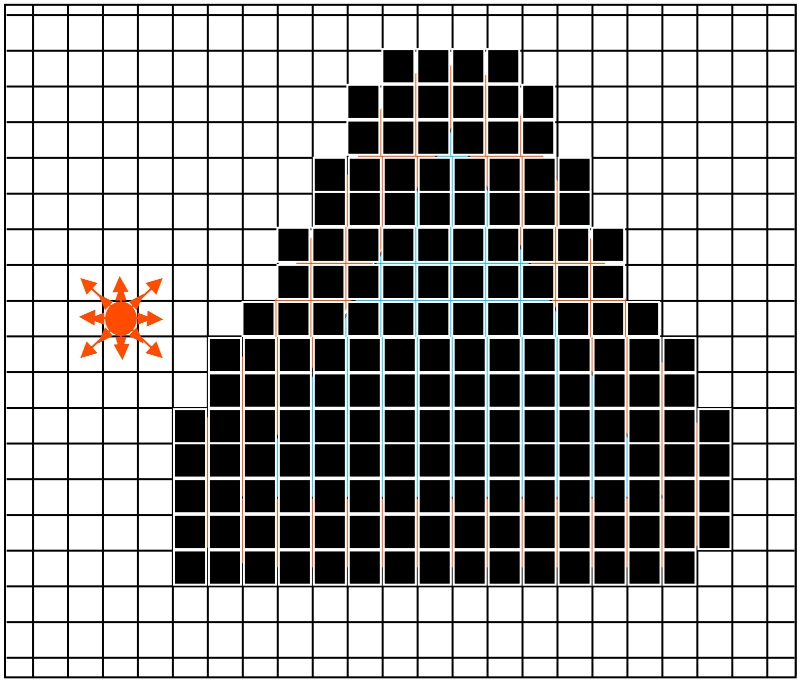
- Numerical Grid methods for Motion Planning
- Potential Fields for Motion Planning

Planning as Search

- Planning Involves ***Search*** Through a ***Search Space***
  - How to conduct the search?
  - How to represent the search space?
  - How to evaluate the solutions?
- Non-Deterministic ***Choice Points*** Determine Backtracking
  - Choice of actions
  - Choice of variable bindings
  - Choice of temporal orderings
  - Choice of subgoals to work on

Setting up the State Space

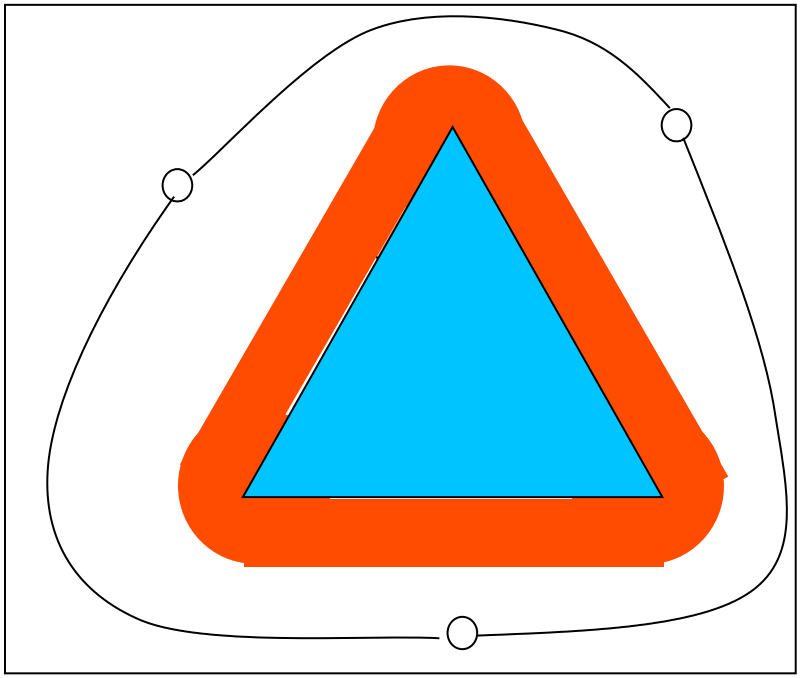
- Real space
- Configuration space
- State space



- Actions get you from one state to another

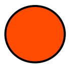
- Objective is to find a path from the start to the goal

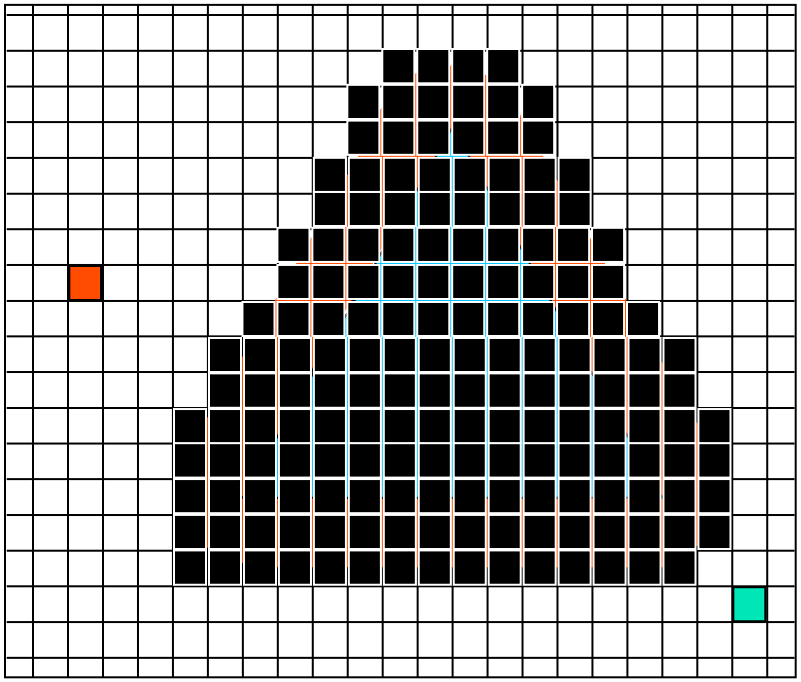
Topological Discretizations



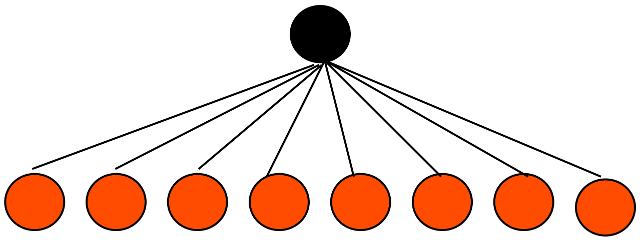
- State space could be states chosen from the c-space at random
- Sampling states at random is the "probabilistic roadmap"
- Visibility graph is optimal (in 2 dimensions only, however)
- PRM is only optimal in the limit of infinite number of samples
- Trade-off: optimality vs. difficulty of computing configuration space exactly

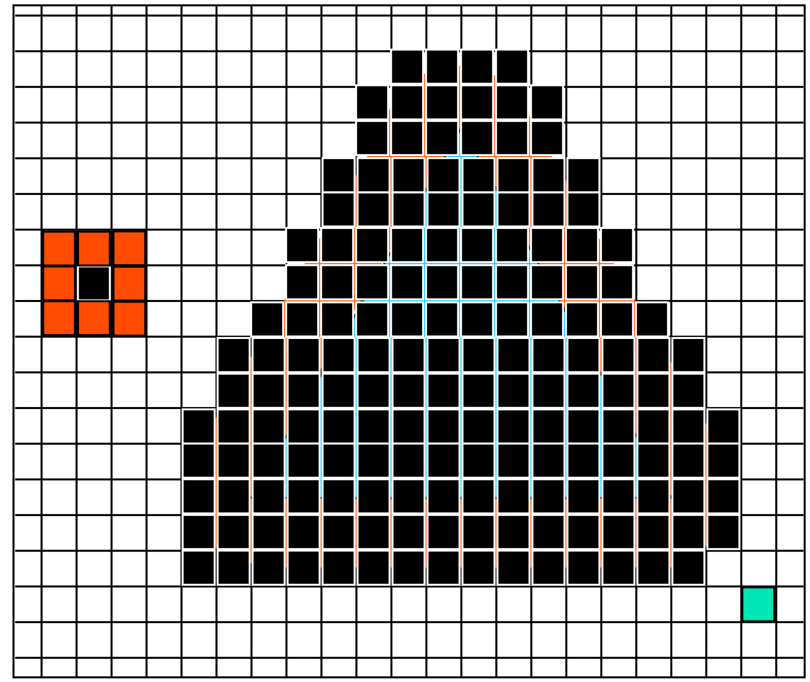
Planning by Searching a Tree



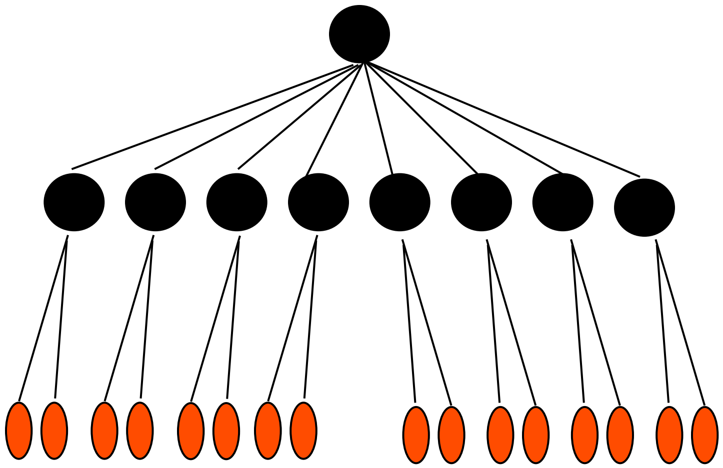


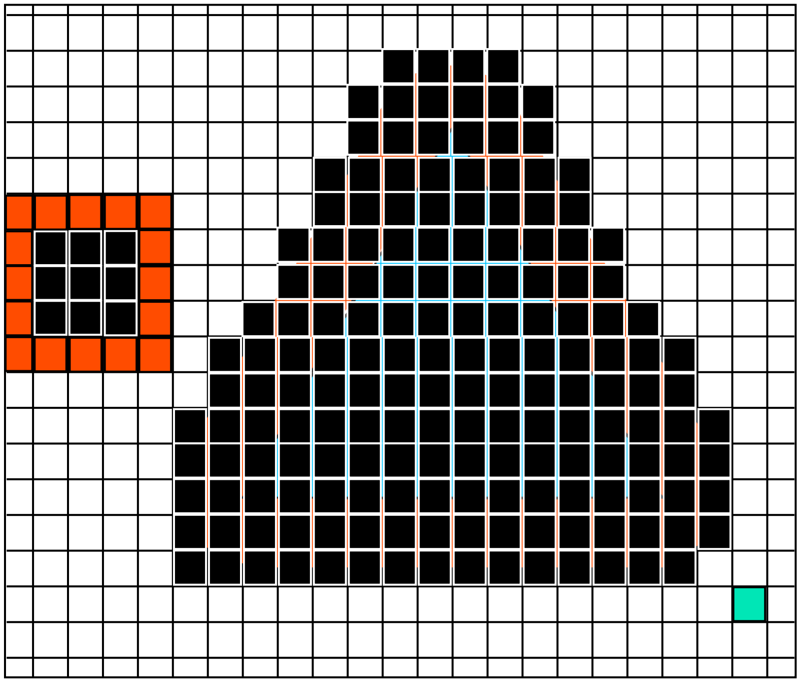
Planning by Searching a Tree



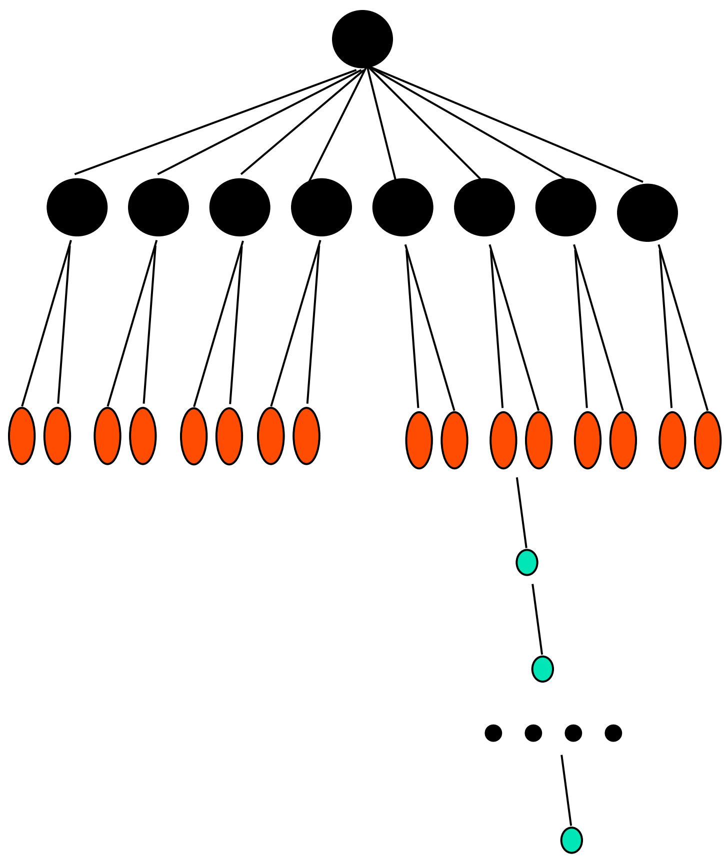


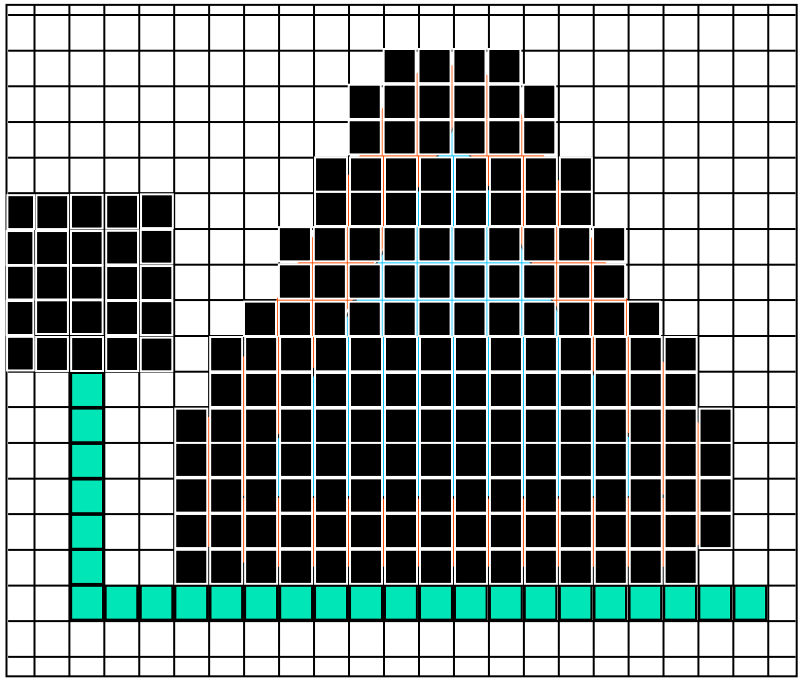
Planning by Searching a Tree





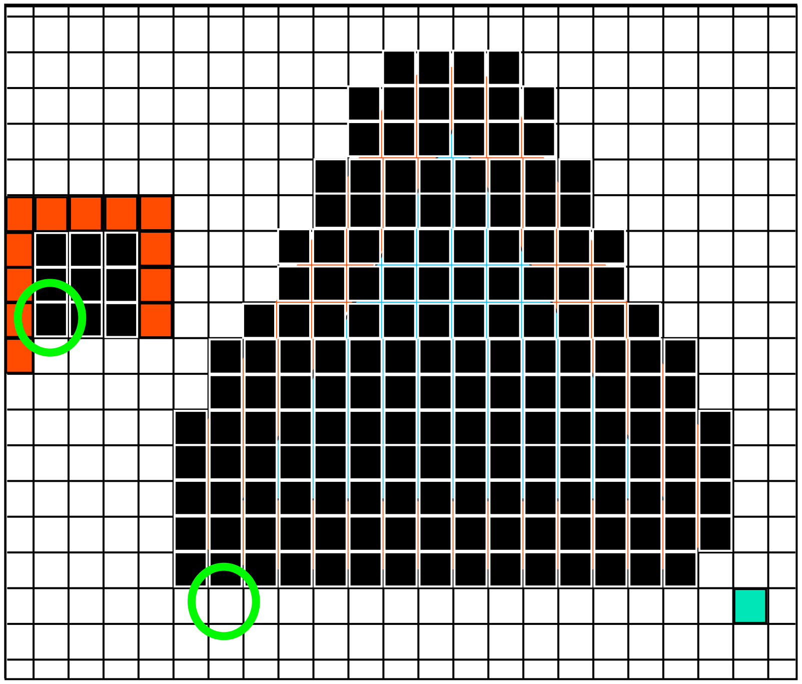
Planning by Searching a Tree





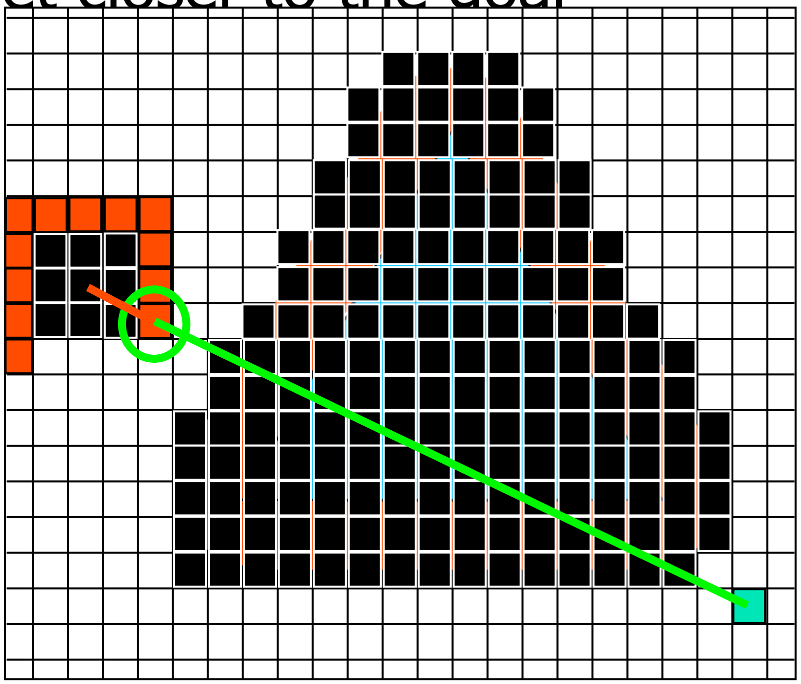
Move Generation

- Which state-action pair to consider next?
- Shallowest next
  - aka: Breadth-first search
  - Guaranteed shortest
  - Storage intensive
- Deepest next
  - aka: Depth-first search
  - Can be storage cheap
  - No shortness guarantees
- Cheapest next
  - aka: Uniform-cost search
  - Breadth-first search is the same if the cost == depth



Informed Search – A*

- Use domain knowledge to bias the search
- Favor actions that might get closer to the goal
- Each state gets a value
  $f(x)=g(x)+h(x)$

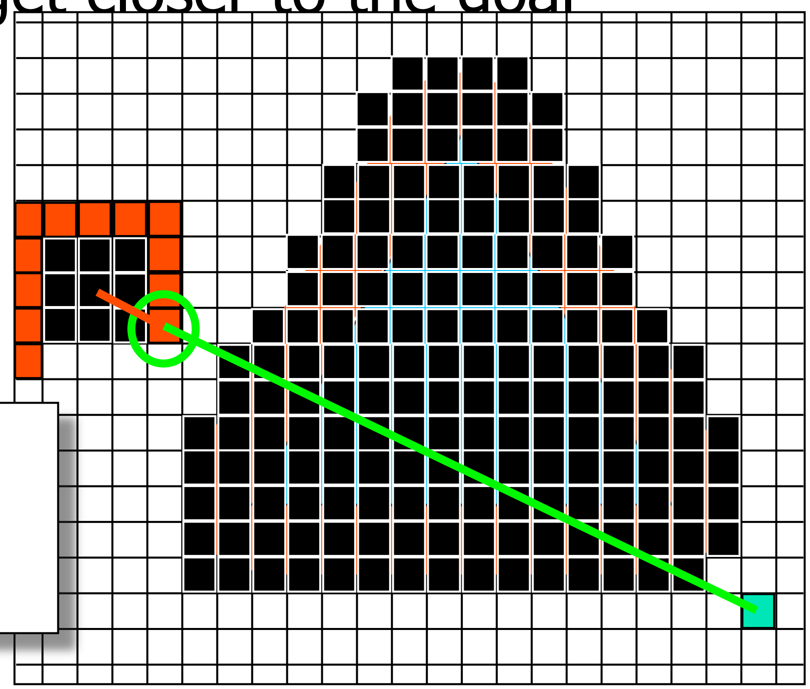


Informed Search – A*

- Use domain knowledge to bias the search
- Favor actions that might get closer to the goal
- Each state gets a value
  $f(x)=g(x)+h(x)$

Cost incurred so far, from the start state

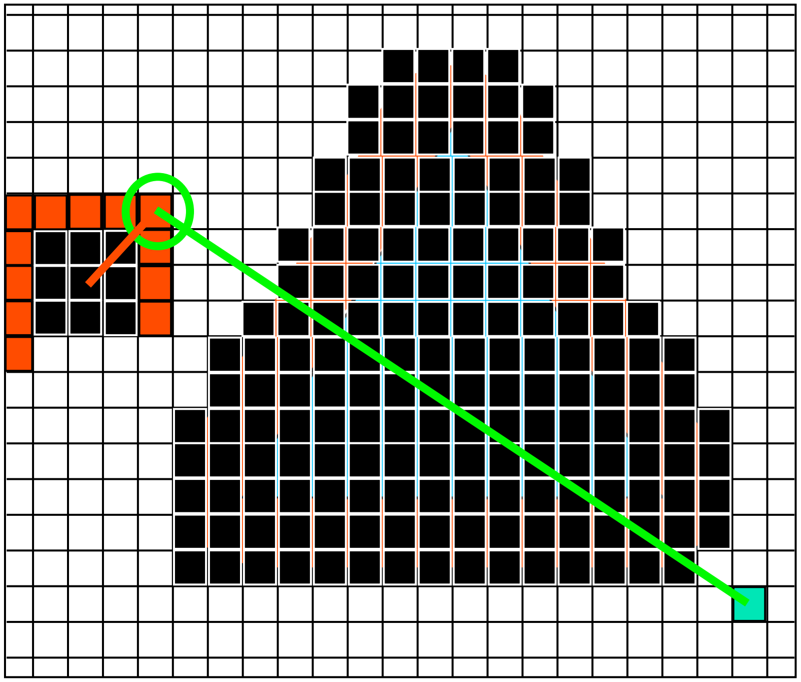
Estimated cost from here to the goal: “heuristic” cost



- For example
  - $g(x) = 3$, $h(x) = ||x-g|| = \mathrm{sqrt}(8^2+18^2)=19.7$, $f(x)=22.7$

Informed Search – A*

- Use domain knowledge to bias the search
- Favor actions that might get closer to the goal
- Each state gets a value $f(x)=g(x)+h(x)$
- Choose the state with best f



- For example
  - $g(x) = 4$, $h(x) = ||x-g||=\text{sqrt}(11^2+18^2)=21.1$, $f(x)=25.1$

How to choose heuristics

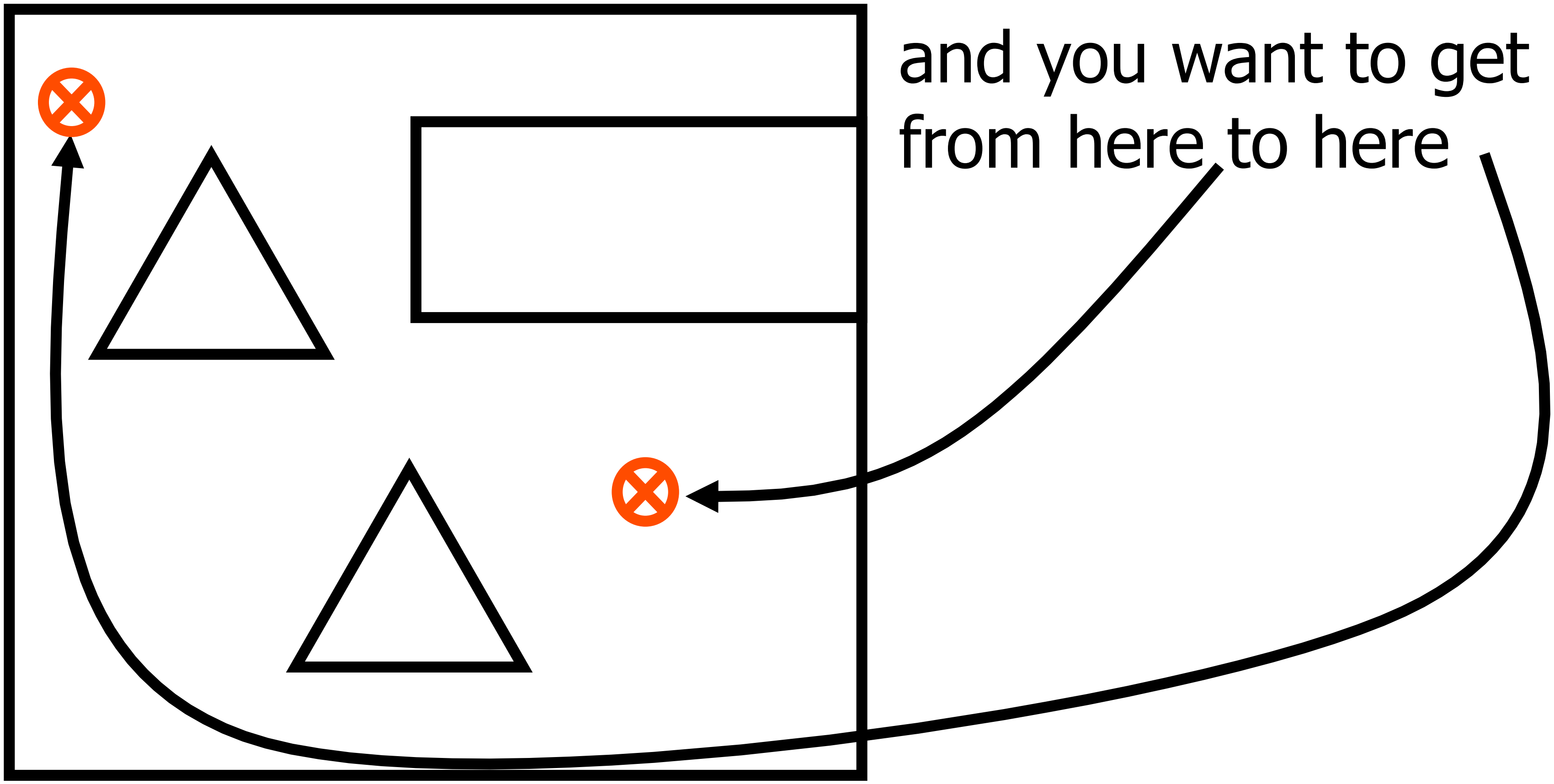
- The closer $h(x)$ is to the true cost to the goal, $h^*(x)$, the more efficient your search

BUT

- $h(x) \leq h^*(x)$ to guarantee that A* finds the lowest-cost path
- In this case, h is an "admissible" heuristic

Let’s Recap

- Your mapping software gives you a great map....

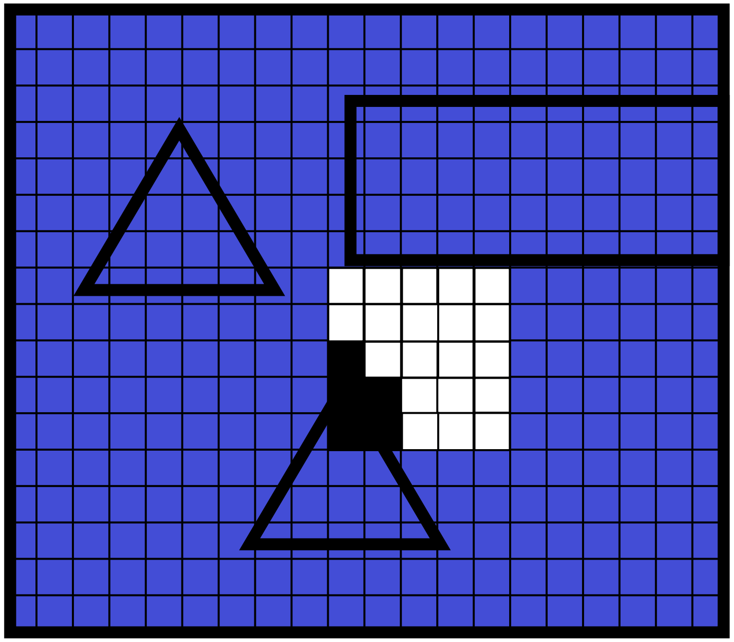


Decisions

- How is your map described? This may have an impact on the state space for your planner
  - Is it a grid map?
  - Is it a list of polygons?
- What kind of controller do you have?
  - Do you just have controllers on distance and orientation?
  - Do you have behaviours that will let you do things like follow walls?
- What do you care about?
  - The shortest path?
  - The fastest path?
- What kind of search to use?
  - Do you have a good heuristic?
  - If so, then maybe A* is a good idea.

What’s a good algorithm for turning a polygonal c-space into a grid?

- A grid square is in the c-space if it is:
  - not inside an obstacle
  - further than the radius of the robot from all obstacle edges
- Algorithm:
  - Pick a grid square you know is in free space
  - Do breadth-first search (or “flood-fill”) from that start square
  - As each square is visited by the search, compute the distance to all obstacle edges
  - label as “free” if the distance is greater than the radius of the robot or “occupied” if the distance is less
  - Once breadth-first search is done, also label all unlabelled squares as “occupied”

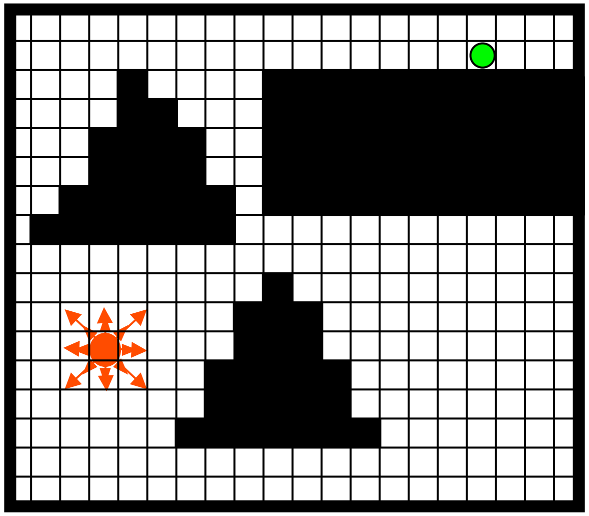


Once we have our state space (and action space, and cost function...)

- Perform A* search
  - Construct the root of the tree as the start state, and give it value 0
  - While there are unexpanded leaves in the tree
    - Find the leaf x with the lowest value
    - For each action, create a new child leaf of x
    - Set the value of each child as:

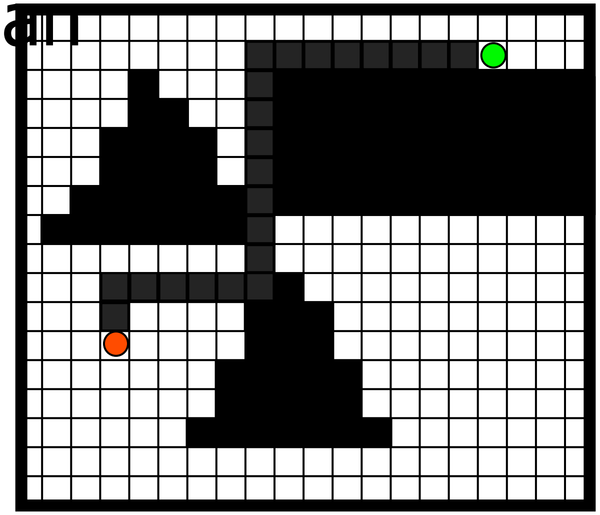
$$g(x) = g(\text{parent}(x)) + c(\text{parent}(x), x)$$
$$f(x) = g(x) + h(x)$$

where c(x, y) is the cost of moving from x to y (distance, in this case) and h(x) is the heuristic estimate of the remaining cost to the goal from x (euclidean distance, in this case)



Once the search is done, and we have found the goal

- We have a tree that contains a path from the start (root) to the goal (some leaf)
- Follow the parent pointers in the tree and trace back from the goal to the root, keeping track of which states you pass through
- This set of states constitutes your plan

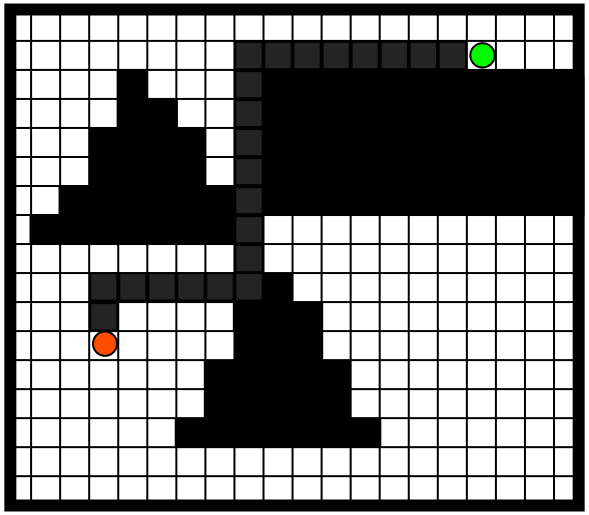


- How do we execute the plan?

Once the search is done, and we have found the goal

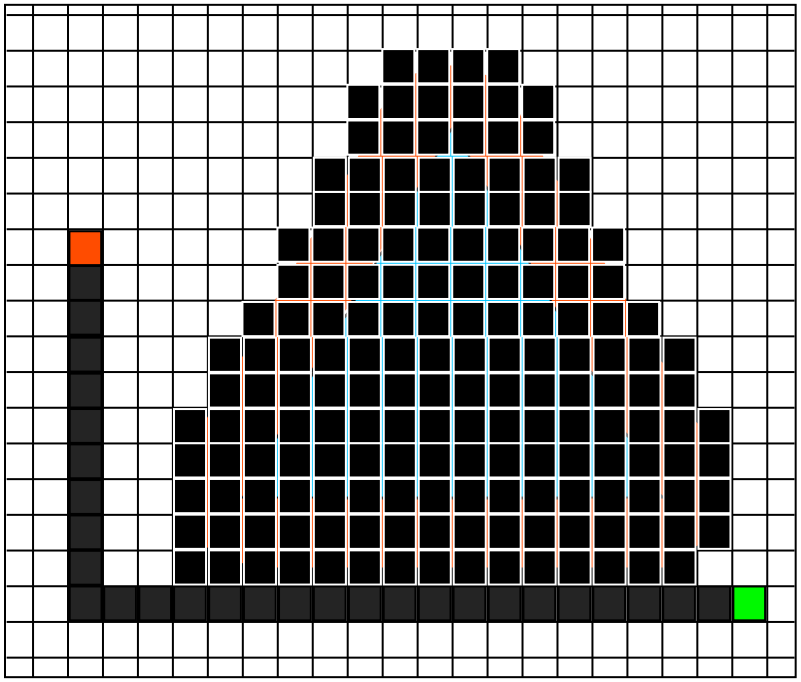
- We have a tree that contains a path from the start (root) to the goal (some leaf)
- Follow the parent pointers in the tree and trace back from the goal to the root, keeping track of which states you pass through
- This set of states constitutes your plan

- To execute the plan, use your PD controller to face the first state in the plan, and then drive to it
- Once at the state, face and drive to the next state



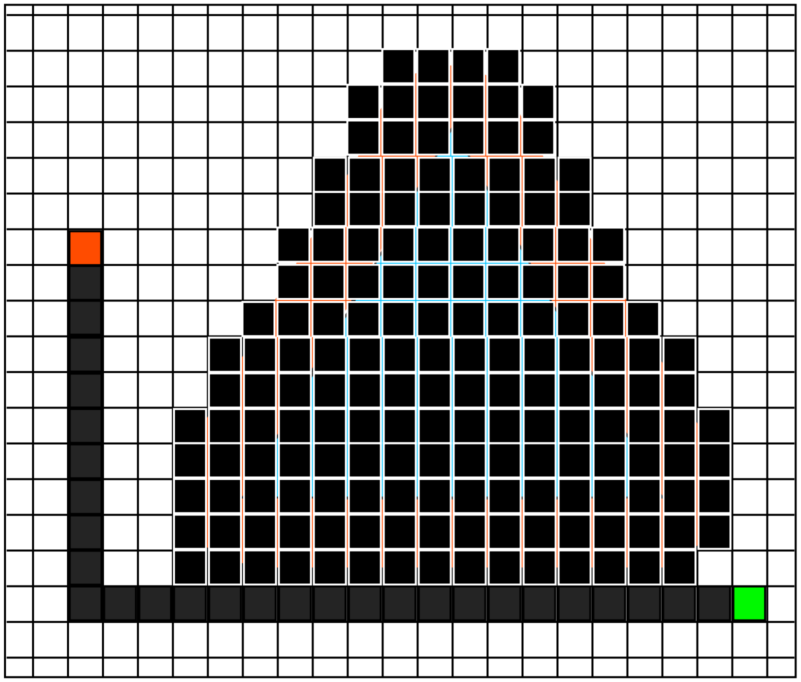
A problem with plans

- We have a plan that gets us from the start to the goal
- What happens if we take an action that causes us to leave the plan?



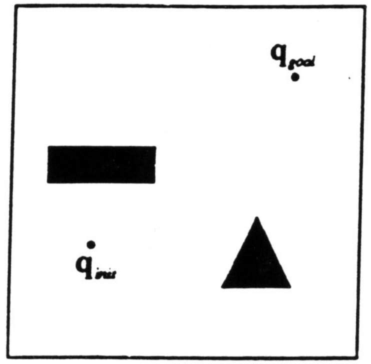
A problem with plans

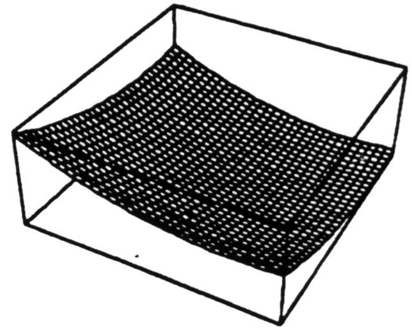
- We have a plan that gets us from the start ■ to the goal ■
- What happens if we take an action that causes us to leave the plan?



1) It’s a problem with planners!
   We should use behaviors!
2) We can replan
3) We can keep a cached conditional plan
4) We can keep a policy

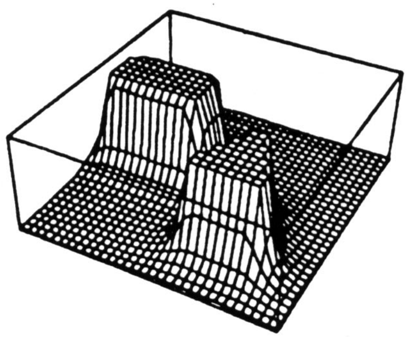
Potential Fields

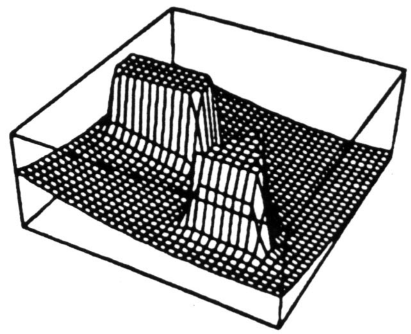




Attractive potential field for goal

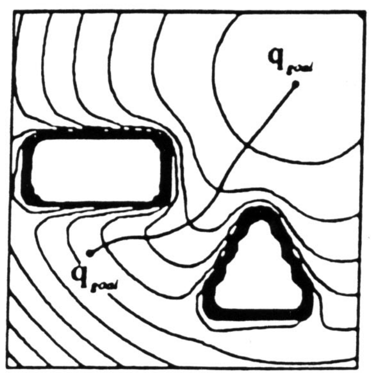
Repulsive potential field for obstacles

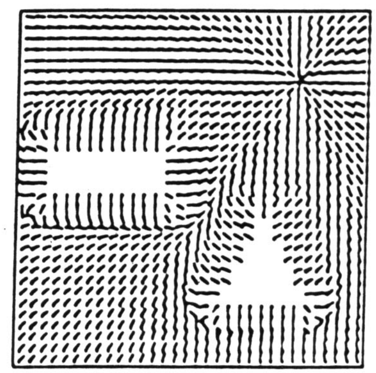




Attractive + repulsive potential fields

Equi-potential contours





Force field

[Latombe 91]

A Reactive Motion Planner

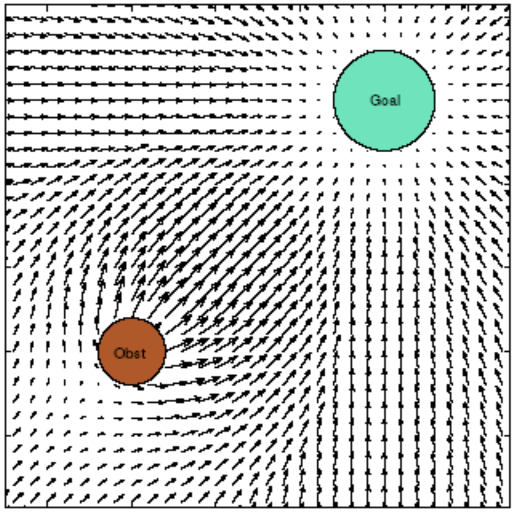
- The potential of each obstacle generates a repulsive force

$$U_{rep} = \frac{1}{\|x - x_c\|}$$

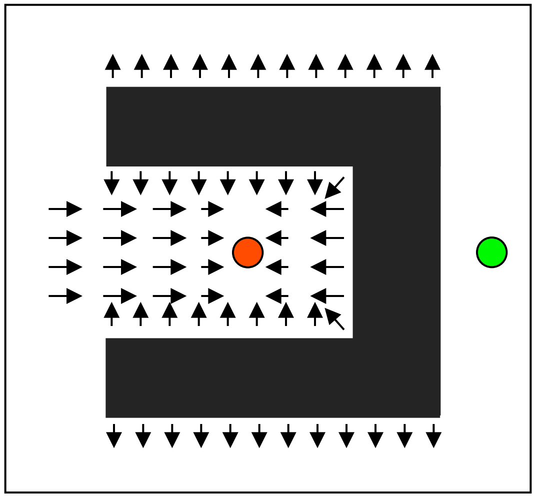
and the potential of the goal generates an attractive force

$$U_{att} = \frac{1}{2}\|x - x_{goal}\|^2$$

- Easy and fast to compute
- Susceptible to local minima



Potential Field



Potential Field Controllers

- Basic idea
  - Construct potential field for goal
  - Construct potential field for each obstacle
  - Add potential fields to create the total potential *V* (*x*, *y*)

    Assume two-dimensional space (robot is a point)

$$\frac{dx}{dt} = -k\frac{\partial V}{\partial x}$$
$$\frac{dy}{dt} = -k\frac{\partial V}{\partial y}$$

  - Force on a particle is given by *f = -grad* (*V*)
  - Command robot velocity according to the following control law (policy)

Numerical Potential Functions

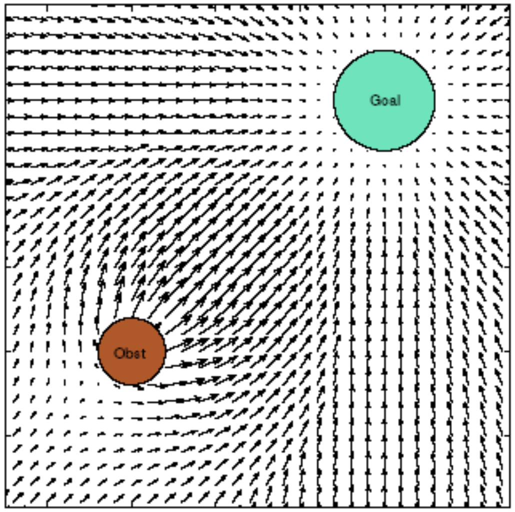
- We can compute the "true" potential at each point x by integrating the forces along the desired path from the goal to x

$$V(x) = \min_{\pi} \int_{\pi} -\nabla U_{att}(\pi(t)) - \nabla U_{rep}(\pi(t))\, dt$$

- If we discretize the path, we get

$$V(x) = \min_{x \to x_{goal}} \sum_{x' \in x \to x_{goal}} \left(-\nabla U_{att}(x') - \nabla U_{rep}(x') \right) \delta x'$$

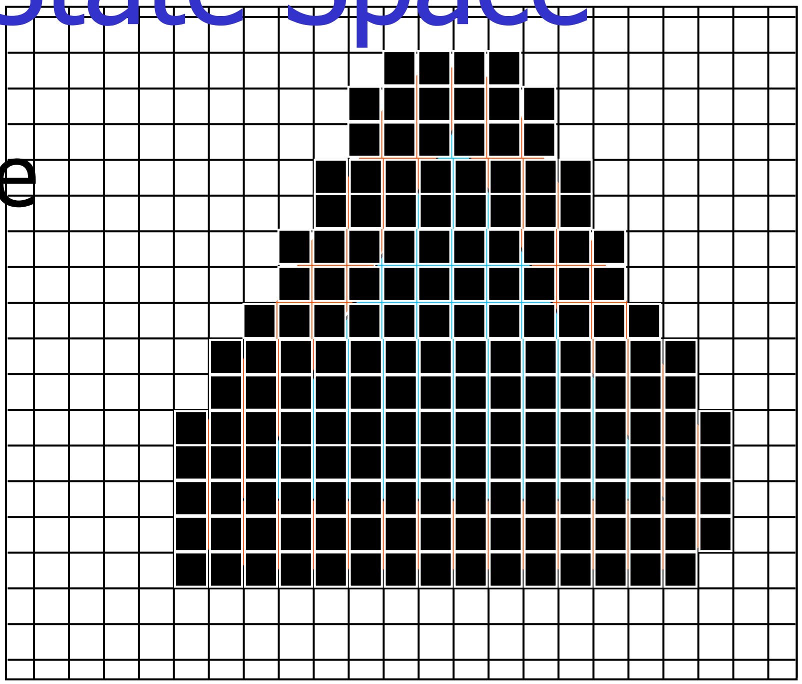
- Let's think about this recursively; intuitively potential at x is minimum over all places x' potential at x' + cost of moving from x to x'



Potential Field

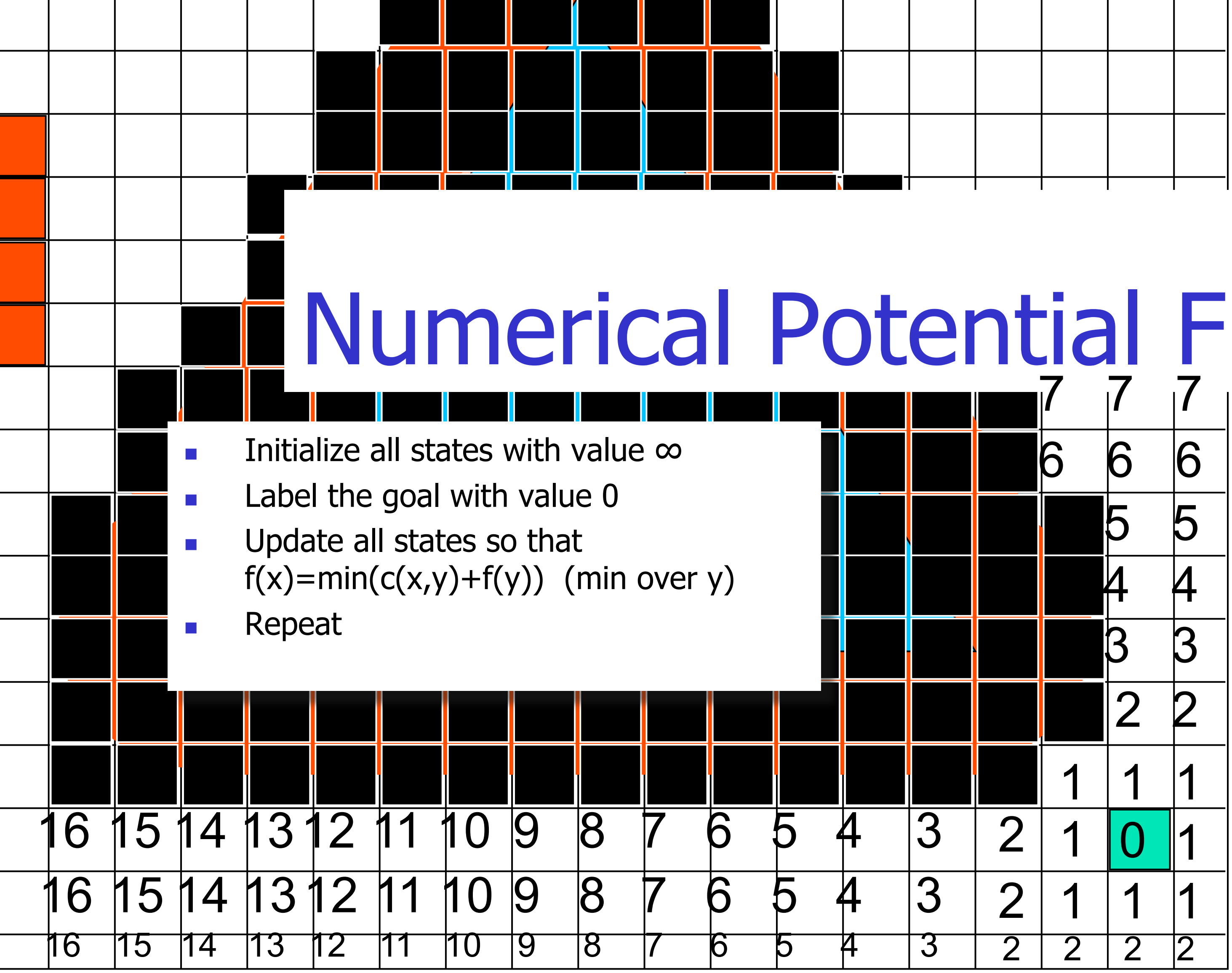
Setting up the State Space

- Again, we discretize our configuration space



Numerical Potential Field

- Initialize all states with value ∞
- Label the goal with value 0
- Update all states so that $f(x)=\min(c(x,y)+f(y))$ (min over y)
- Repeat



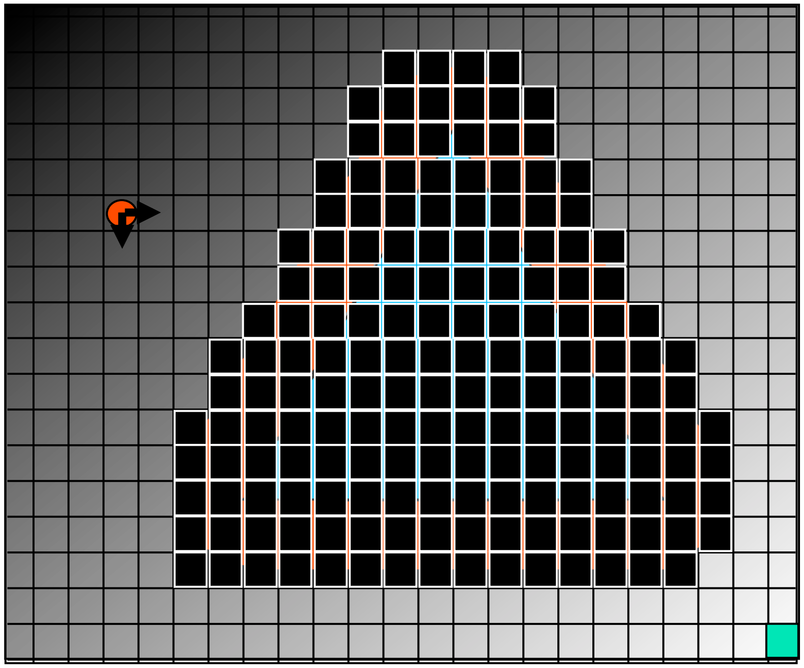
- The numbers shown are for an obstacle-induced cost of 0, and a goal-induced cost of 1 unit per grid cell

Uniform Cost Regression

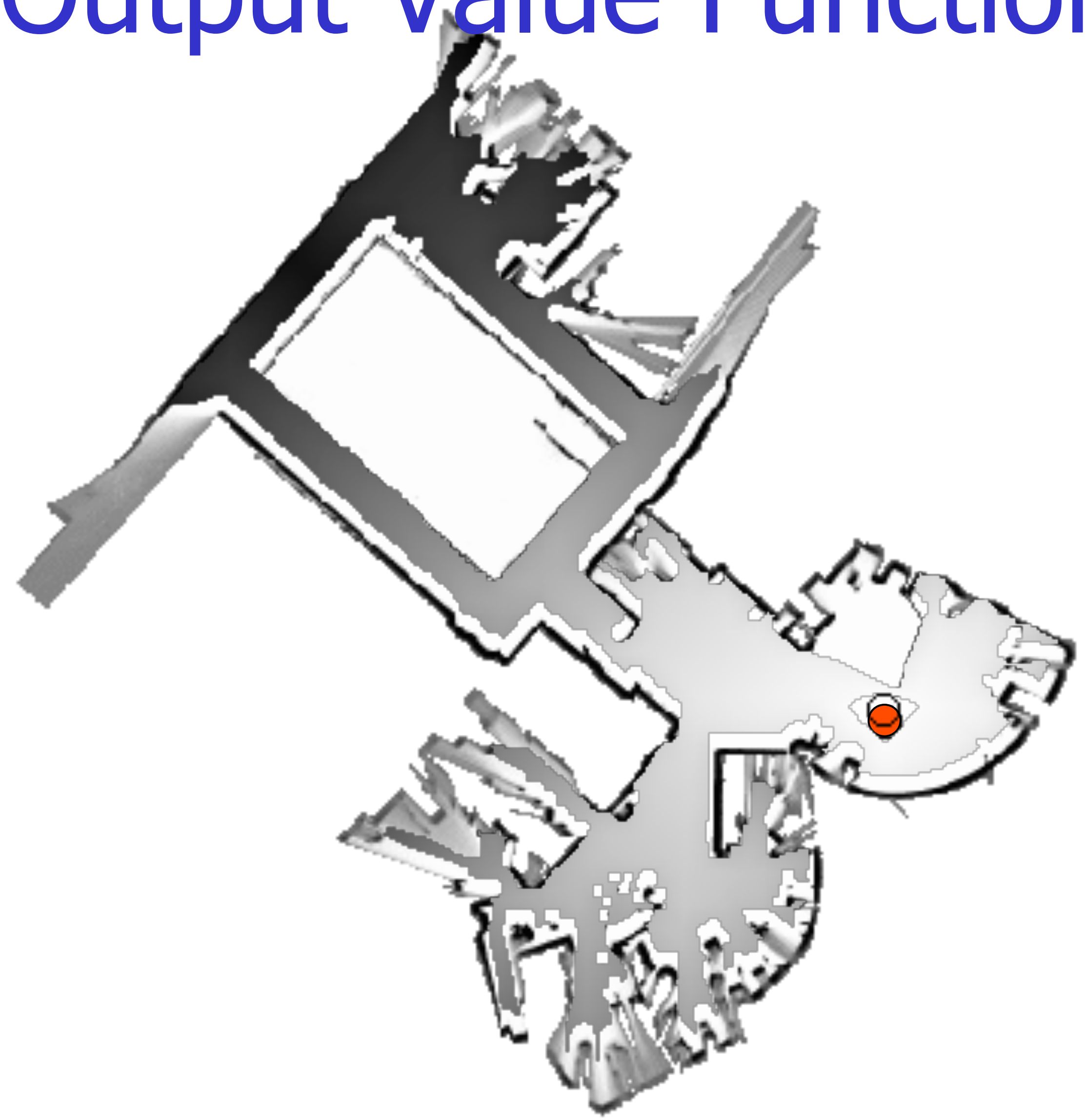
- Initialize all states with value ∞
- Label the goal with value 0
- Update all states so that $f(x)=\min(c(x,y)+f(y))$
- Repeat

- Bellman-Ford's algorithm

- After planning, for each state, just look at the neighbors and move to the cheapest one, i.e., just roll down hill

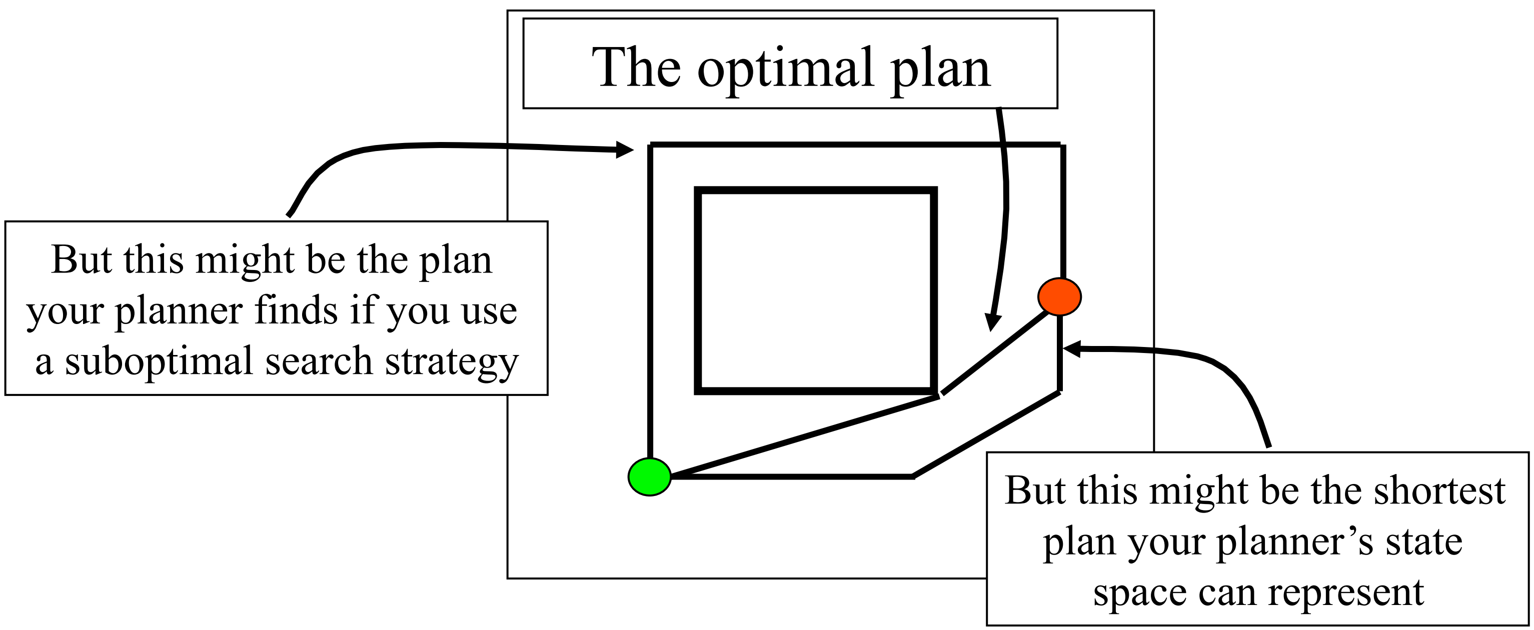


The Output Value Function



Planner Optimality

- There are two issues here
  - Can the planner *express* the best (shortest, fastest) plan possible?
  - Will the planner find the best plan it can express?
- Your state and action spaces affect the former
- Your search strategy (search vs. Dijkstra's algorithm, which state in the search tree to expand next, etc.) affects the latter

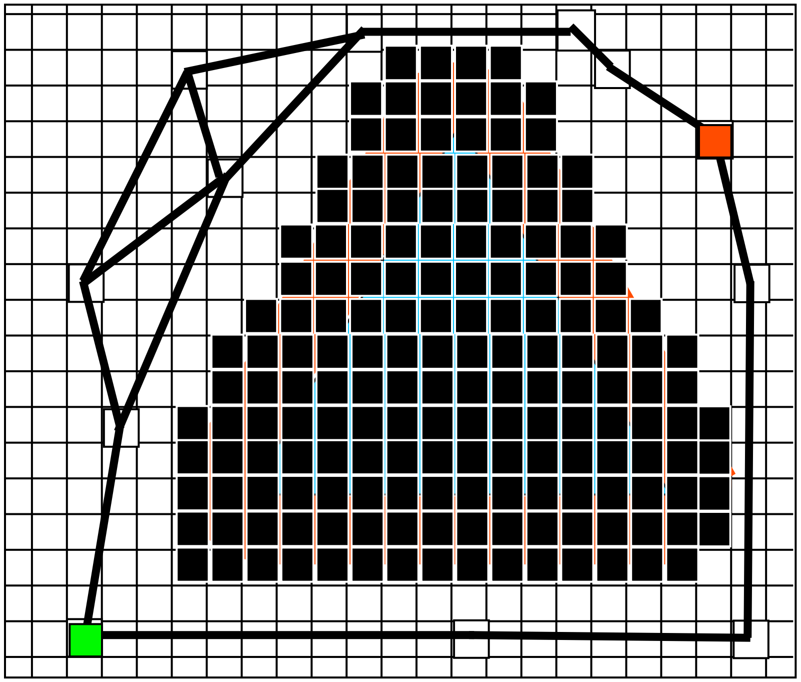


Planner Complexity

- Optimal search algorithms are $O(d^{|a|})$ where d is the depth of the solution in the search tree, and $|a|$ is the number of actions
- Dijkstra's algorithm (and therefore the numerical potential field) is $O(n^2)$ if implemented naively, $O(n \log n)$ if states are updated using priority queues, where n is the number of states
- There's a trade-off between the computational complexity you can afford, and how good a plan you need

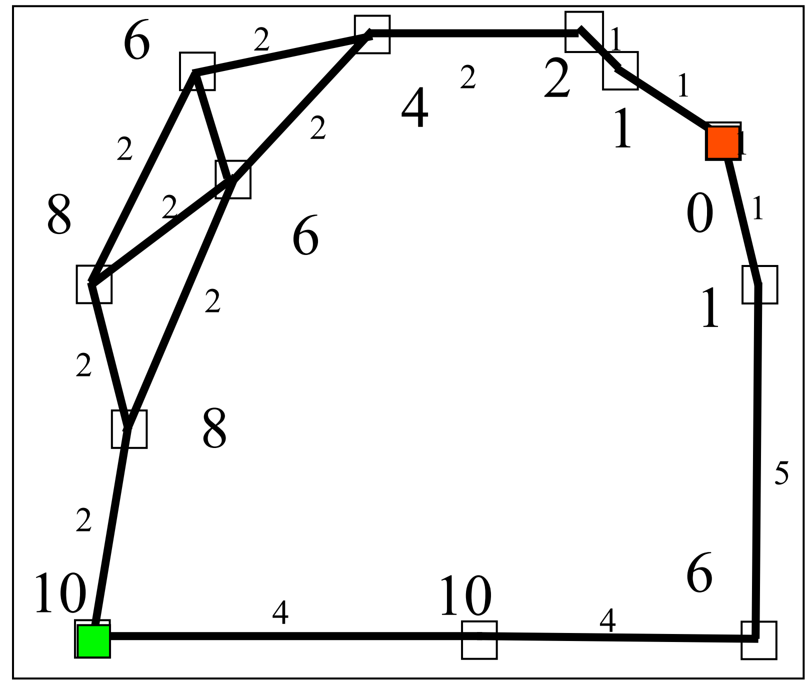
We can use shortest path algorithm on other State Spaces

- We can reduce the state space size by sampling from the configuration space, rather than using a regular grid
- Potential advantages:
  - More efficient search
- Potential disadvantages:
  - How to connect sampled states?
  - Good sampling strategies
  - Limited set of possible plans



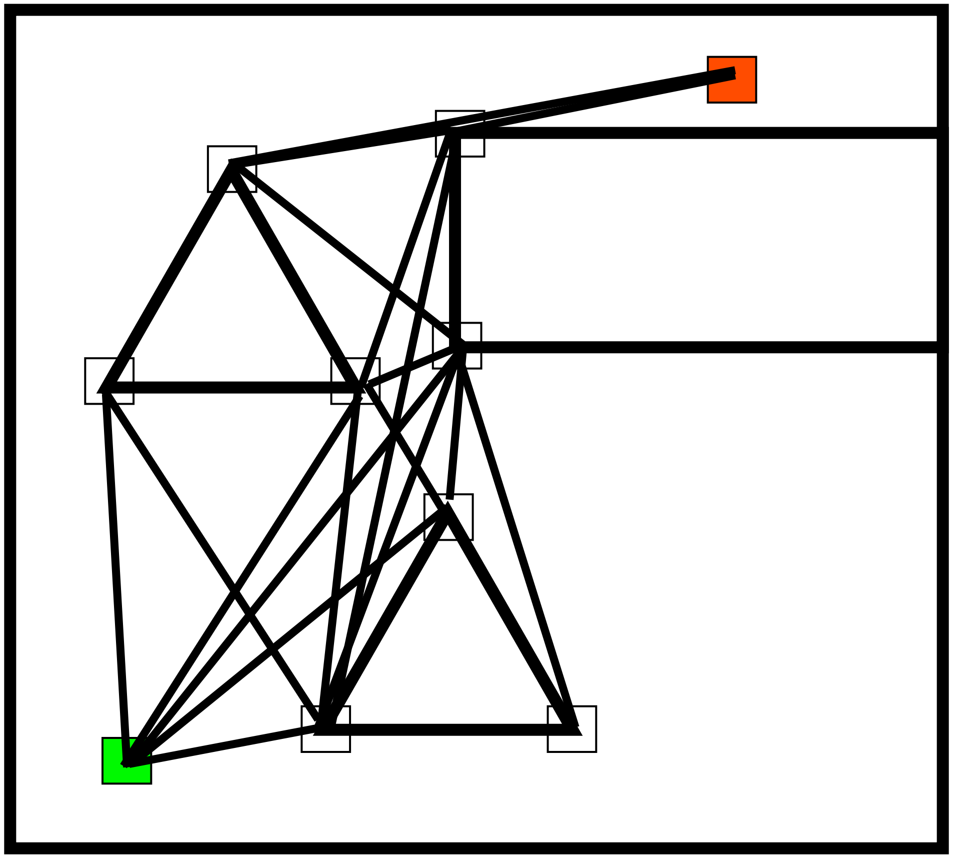
Running Shortest Path Algorithm on the Probabilistic Roadmap

- Sample states randomly
- Add the start and goal state
- Add action edges between states x and y if you can get from x to y with your controller, and set cost c(x, y)
- Initialize all states with value ∞
- Label the goal with value 0
- Update all states so that f(x)=min(c(x,y)+f(y))
- Repeat



Running Shortest Path Algorithm on the Visibility Graph

- Put states at the start, goal, and polygon corners
- Add action edges between states x and y if you can get from x to y with your controller, and set cost c(x, y)
- Initialize all states with value ∞
- Label the goal with value 0
- Update all states so that f(x)=min(c(x,y)+f(y))
- Repeat



What you should know

- Planning as search
- The design decisions in setting up a planner
- C-obstacle algorithm and grid approximation
- Different forms of search: breadth-first, depth-first, A*
- Mapping motion planning to graph search using v-graphs, cell decomposition, PRMs, grids, numerical potentials
- How to decide which is best to use each

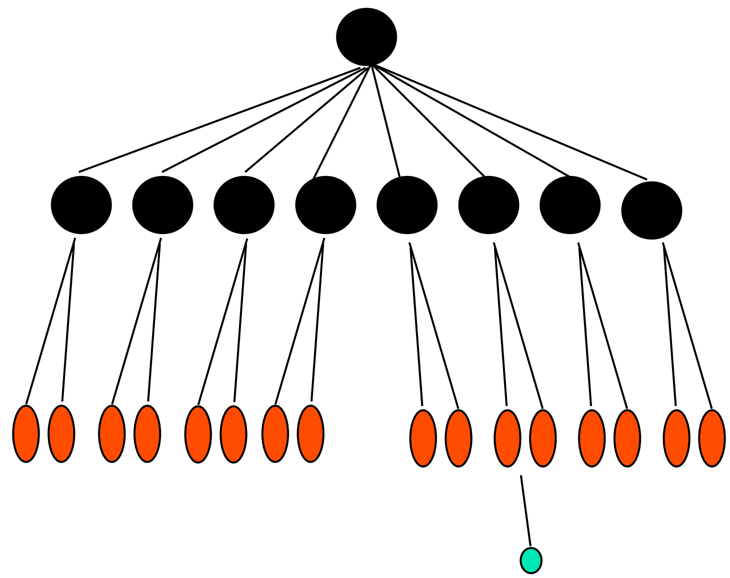
Design Choices

- What state space to use
- What set of actions to use
- What search method to use
- What cost function to use (distance, time, etc.)
- If using informed search, what heuristic to use

- The critical choice for motion planning is state space
- The other choices tend to affect computational performance, not robot performance

Data Structures

- While there are unexpanded leaves in the tree
  - Find the leaf x with the lowest value
  - For each action, create a new child leaf of x
  - Set the value of each child
- Let's say that each tree node is given by



```
public class State {
     int id;
     double coordinates[2];
     int neighbours[];
     double priority;
     State parent;
     State [] children;
}
```

- Then as you create new children, you store them in the children array inside the parent State
- The tree structure, however, does **not** automatically tell you the lowest (or highest) priority child
- Therefore, as you add each child to the parent state in the tree, also add the child to a sorted set (e.g., java.util.TreeSet) that has the methods add() and first() that will let you add items and retrieve the lowest (highest) items in O (log n) time. (NB: If using TreeSet, you would need to make sure your State class implements the comparable interface.)

Progression vs. Regression

- **Progression (forward-chaining):**
  - Choose action whose preconditions are satisfied
  - Continue until goal state is reached
- **Regression (backward-chaining):**
  - Choose action that has an effect that matches an unachieved subgoal
  - Add unachieved preconditions to set of subgoals
  - Continue until set of unachieved subgoals is empty
- Progression: + Simple algorithm ("forward simulation")
  - Often large branching factor
- Regression: + Focused on achieving goals
  - Need to reason about actions
  - ***Regression is incomplete, in general, for functional effects***

Potential Field Controllers

- Basic idea
  - Create attractive potential field to pull robot (*R*) toward a goal

$$V_{goal} = k\left[d(R, goal)\right]^2$$

  - Create repulsive potential field to repel robot (*R*) from obstacles

$$V_{obs} = \frac{c}{d(R, obs)}$$

$$d(R, goal) = \left[(x - x_{goal})^2 + (y - y_{goal})^2\right]^{\frac{1}{2}}$$

$$d(R, obs) = \left[(x - x_{obs})^2 + (y - y_{obs})^2\right]^{\frac{1}{2}}$$

  - In two-dimensional space (robot is a point, goal/obstacles are points)
  - Remember: Force on a particle is given by *f* = -*grad*(*V*)

Optimal vs. Satisficing

- In motion planning, we typically prefer “shortest” paths, in distance, time, power consumption or some other objective
- Our choice of objective function implies a cost (or reward) function on actions

- Sometimes, we just want to find **any** sequence of actions that connects the start and the goal

Numerical Potential Functions

- We can compute the "true" potential at each point x by integrating the forces along the desired path from the goal to x

$$V(x) = \min_{\pi} \int_{\pi} -\nabla U_{att}(\pi(t)) - \nabla U_{rep}(\pi(t))dt$$

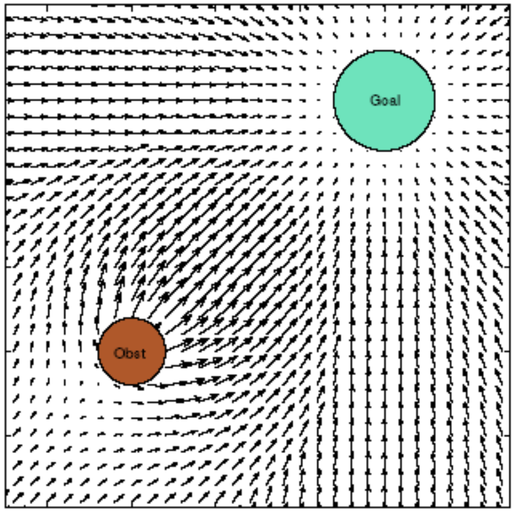
- If we discretize the path, we get

$$V(x) = \min_{x \to x_{goal}} \sum_{x' \in x \to x_{goal}} \left(-\nabla U_{att}(x') - \nabla U_{rep}(x')\right)\delta x'$$

- Let's write this recursively:

$$V(x) = -\left(\nabla U_{att}(x) + \nabla U_{rep}(x)\right)\delta x + \min_{x' \in a(x)} V(x')$$

$$= C(x) + \min_{x' \in a(x)} V(x')$$



Potential Field

$$C(x) = F(x) = \nabla U_{att}(x) - \nabla U_{rep}(x)$$