Today’s Objectives
- Planning and search
  - Search methods
- Plans vs. Policies
  - Numerical potential fields

Let’s Recap

Your mapping software gives you a good map.... and you want to get from here to here

Planning as Search
- Find a sequence/set of actions that can transform an initial state of the world to a goal state
- Planning Involves Search Through a Search Space
  - How to represent the search space?
  - How to conduct the search?
  - How to evaluate the solutions?
Motion Planning as Search

- Find a sequence of poses that connects the initial pose of the robot to the goal pose
- State space is configuration space
  - To perform search, we discretize the space
  - This is a big issue in planning: how do we discretize the search space? Is it a graph? Is it a grid?
- Actions connect pairs of states
  - Assume a P-D controller
  - If the controller can get you from one pose to the other, then that action connects those states
  - In this course, we assume pairs of mutually visible states are connected

Setting up the State Space

- Real space
- Configuration space
- State space
- Actions get you from one state to another

Finding the free part of c-space using 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"

Planning as Search

- Find a sequence/set of actions that can transform an initial state of the world to a goal state
- Planning involves Search Through a Search Space
  - How to represent the search space?
  - How to conduct the search?
  - How to evaluate the solutions?
Planning as Tree Search

- Perform tree-based search (need c-space, cost)
  - 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:
    \[ g(x) = g(parent(x)) + c(parent(x), x) \]
    where \( c(x, y) \) is the cost of moving from x to y (distance, in this case)
Planning by Searching a Tree

Simple Search Algorithm

Let Q be a list of partial paths, Let S be the start node and Let G be the Goal node.

1. Initialize Q with partial path (S) as only entry; set Visited = {}.
2. If Q is empty, fail. Else, pick some partial path N from Q.
3. If head(N) = G, return N (goal reached)

4. Else
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create all the one-step extensions of N to each child.
   c) Add to Q all the extended paths;
   d) Add children of head(N) to Visited
   e) Go to step 2.
Simple Search Algorithm

```java
public class Search {
    public static Path search(State start, State goal) {
        Queue q = new Queue();
        HashSet<State> visited = new HashSet<State>;
        q.add(new Path(start));
        while (!q.empty()) {
            Path partialPath = q.pop();
            State head = partialPath.head();
            if (head.matches(goal))
                return partialPath; // Goal reached!
            for (int i = 0; i < head.numNeighbours(); i++) {
                State neighbour = head.neighbour(i);
                if (visited.contains(neighbour))
                    continue;
                Path extension = new Path(neighbour, partialPath);
                visited.add(neighbour); // Create a new path to a node we haven't seen before
                q.push(extension); // By adding the neighbour of the head to the current path
                extension = new Path(neighbour, partialPath);
                visited.add(neighbour); // by extending the current path
                q.push(extension); // to the current path
            }
        } // End of while (q.empty());
        return null; // No path found
    }
}
```

Careful: the HashSet object to get O(1) tests on whether we have seen this state before, but we may have to override the Object.hashCode method for our State class.

Move Generation

- How to determine the lowest-cost child to consider next?
  - Shallowest next
    - aka: Breadth-first search
    - Guaranteed shortest
    - Storage intensive
  - Deepest next
    - aka: Depth-first search
    - Can be storage cheap
    - No optimality 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
- Favour actions that might get closer to the goal
  - Each state gets a value:
    - $f(x) = g(x) + h(x)$
    - For example:
      - $g(x) = 3$
      - $h(x) = ||x-g||$
      - $f(x) = g(x) + h(x)$
      - $f(x) = 19.7$
      - $f(x) = 22.7$

Informed Search – A*

- Use domain knowledge to bias the search
- Favour actions that might get closer to the goal
  - Each state gets a value:
    - $f(x) = g(x) + h(x)$
    - For example:
      - $g(x) = 4$
      - $h(x) = ||x-g||$
      - $f(x) = g(x) + h(x)$
      - $f(x) = 21.1$
      - $f(x) = 25.1$
**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.

**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?
  1. It's a problem with planners! We should use behaviours!
  2. We can replan.
  3. We can keep a cached conditional plan.
  4. We can keep a policy.

**A Reactive Motion Planner**

- The potential of each obstacle generates a repulsive force:
  $$U_{rep} = \frac{1}{\|x - x_i\|}$$
  and the potential of the goal generates an attractive force:
  $$U_{at} = \frac{1}{2}\|x - x_{goal}\|^2$$

- Easy and fast to compute.
- Susceptible to local minima.
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 \( x \)

\[
V(x) = \min_x \int_a^b \left[ -\nabla U_{at}(\pi(t)) - \nabla U_{rep}(\pi(t)) \right] dt
\]

- If we discretize the path, we get

\[
V(x) = \min_{x_{\pi_r}} \sum_{r=1}^{n} \left( \nabla U_{at}(x') - \nabla U_{rep}(x') \right) h'
\]

Let's write this recursively:

\[
V(x) = \left( \nabla U_{at}(x) + \nabla U_{rep}(x) \right) h + \min_{x' \in \pi} V(x')
\]

\[= C(x) + \min_{x' \in \pi} V(x') \quad C(x) = F(x) = \nabla U_{at}(x) - \nabla U_{rep}(x) \]

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 $\infty$
- Label the goal with value 0
- Update all states so that $f(x) = \min(c(x,y) + f(y))$
- Repeat
- aka: Dijkstra’s algorithm
- After planning, for each state, just look at the neighbours and move to the cheapest one, i.e., just roll down hill

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*

The Output Potential Field

Data Structures Prof. Roy has Known and Loved
- *Priority Heap*
  - Look it up in Cormen, Leiserson, Rivest and Stein. (MIT gives you free access to the online edition.)
  - Add nodes in $O(\log n)$ time, remove the lowest (or highest) priority node in $O(\log n)$ time.
  - If your heap consists of *State* data structures of the form:
    ```java
    public class State {
        int id;
        double coordinates[2];
        int neighbours[];
        double priority;
    }
    ```
    then you can implement any search algorithm by changing the priority scheme
    - Uniform-first search if
      - current priority = parent state’s priority + action cost from parent to current state
      - choose lowest-priority state
    - Depth-first search if
      - current priority = parent state’s priority + 1
      - choose highest-priority state
    - A* search if
      - current priority = parent state’s priority + action cost from parent to current state + heuristic cost from current state to goal
      - choose lowest-priority state
More on 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
  ```java
  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.)

Design Choices

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

- 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.

Summary

- Planning as search
- The design decisions in setting up a planner
- Different forms of search
- A* and what an admissible heuristic is
- What a policy is, why it’s different from a plan, and when you might want one
- When to use each