Motion Planning

• **Inverse Kinematics:** Determine actuator commands that achieve a desired robot pose

• **Motion Planning:** Computing a series of actuator commands that put the robot into the desired state.
  - State can include the robots position in the world
  - Superset of inverse kinematics
  - But often applied to simpler vehicles.
Motion Planner Properties

• Complete
  ▶ Finds a path if one exists
  ▶ “Doesn’t get stuck”

• Optimality
  ▶ Finds the best path
  ▶ Computational complexity irrelevant
Simple motion planning with “bug” algorithms

- Algorithm 0
  - Drive towards goal
  - If an obstacle gets in the way, follow the obstacle until we can once again drive towards the goal.
The bug is buggy

We could pick a direction randomly, but today is about deterministic planners.
A Better Bug

- Algorithm 1
- Drive towards goal
- If an obstacle gets in the way, circumnavigate the entire obstacle.
- Note the point of closest approach to the goal
- Back track to closest point
- Loop
Bug 1

- This bug is special: it’s *provably complete*.
  - Definitions: Let $d()$ be the distance to goal. Let entry$_i$ be the point at which we encountered the $i$th obstacle, exit$_i$ is the point at which we left that obstacle.

- **Lemma 1:** The bug leaves every obstacle at a point no farther from the goal than the point it arrived at. I.e., $d(\text{exit}_i) \leq d(\text{entry}_i)$.
- **Proof:** Both $d(\text{entry}_i)$ and $d(\text{exit}_i)$ belong to the perimeter, and $d(\text{exit}_i)$ is the closest point to the goal.

- **Lemma 2:** $d(\text{entry}_{i+1}) < d(\text{exit}_i)$
- **Proof:** The robot heads directly towards the goal after exiting obstacle $i$. The obstacles do not overlap, and so the robot makes finite progress towards the goal before hitting another obstacle.

- **Thm:** Bug 1 is complete.
- **Proof:** The sequence \{d(entry$_1$), d(exit$_1$), d(entry$_2$), d(exit$_2$), d(entry$_3$), d(exit$_3$), \ldots\} is monotonically decreasing by Lemma 1 and 2.
Bug 1 Summary

• Complete?

• Optimal?

• Best case runtime of Bug 1

• Worst case runtime of Bug 1
Bug 1 Summary

- Complete?
- Optimal?
- Best case runtime of Bug 1
- \( D \)
- Worst case runtime of Bug 1

complete: yes
Optimal: no
\( P_i \) = perimeter of \( i \)th obstacle.
Bug 1 Summary

- Complete?
- Optimal?
- Best case runtime of Bug 1
  \[ D \]
- Worst case runtime of Bug 1
  \[ D + 1.5 \sum_i P_i \]
Bug 2

- Algorithm 1
  - Construct “m line”
  - Drive toward goal on “m line”
  - If an obstacle gets in the way, begin to circumnavigate the entire obstacle.
    - When we encounter the m-line again closer to the goal, leave the obstacle and drive towards goal on m line.

- Loop
Bug 2 Challenge

• Bug 2 seems to be much better than bug 1.

• It turns out that it is not always better.

• Challenge
  ▶ Find a world in which Bug 1 outperforms Bug 2.
Bug Challenge: Solution
Bug 2 Summary

- Complete?

- Optimal?

- Best case runtime of Bug 2

\[ D \]

- Worst case runtime of Bug 2

\[ D + \sum_{i} \frac{n_i}{2} P_i \]

\( n_i = \# \text{ of times "m-line" intersects polygon. At most } n_i/2 \text{ of these can be valid exit points. Each time, we might go all the way around the obstacle.} \)

\( \text{Hard to compare bug2 and bug1 directly, since complexity depends on the number of obstacles they encounter, which may not be the same. But if same } \# \text{ of obstacles, bug2 is better for convex polygons (where } n_i=2) \)
Planning on a (known) grid

- Assume we know the whole world.
  - Obviously, we can do better than a bug algorithm!

- Suppose robot can move in any direction
State-space search
Depth-First Search

- Recursively explore each action until no additional actions are possible.
- In many problems, we can always do something
  - Infinite search depth

- Complete?
- Optimal?
Breadth-First Search

• Expand all action sequences of depth n before considering sequences of depth n + 1

• Complete?

• Optimal?

• Complexity?
Informed Search

- Some of these paths are getting farther away from the goal!
  - Why search a bad solution when a better possibility exists?
root = new node();
root.state = initialstate;
root.parent = null;
root.cost-so-far = 0;
root.cost-to-go = h(root.state);

fringe = {  root }

do forever
    parent = get node from fringe with minimum cost-so-far + min-cost-to-go

    if parent.state == goalstate
        return solution parent;

    for each action:
        child = new node();
        child.state = propagate(parent.state);
        child.parent = parent;
        child.cost-so-far = parent.cost-so-far + cost(action);
        child.cost-to-go = h(child.state);
        child.action = action;

        add child to fringe
    end for
end do
**A* Optimality Proof**

- Thm: A* is optimal

- **Proof:** (By Contradiction) Suppose that A* computes a sub-optimal answer \(x\). This means that \(\text{cost}(x) > \text{cost}(x')\) for some other \(x'\).

- We know that some prefix \(p\) of \(x'\) exists in the fringe. Since the heuristic never over-estimates the cost to goal, we have \(\text{cost-so-far}(p) + \text{cost-to-go}(p) \leq \text{cost}(x')\).

- Since nodes are removed in order of least total cost, node \(p\) will be expanded before node \(x\). Thus, we never expand a sub-optimal node.
A* Summary

• Pros:
  ‣ Complete
  ‣ Optimal
  ‣ Optimally Efficient
  ‣ Easy to implement
  ‣ Works well when number of actions is small
  ‣ Size of state space does not directly search complexity

• Cons:
  ‣ Worst-case cost no better than breadth-first: $O(b^d)$
  ‣ Large memory costs: $O(b^d)$
  ‣ Large computational costs
  ‣ Even with a MinHeap
  ‣ Can be exceptionally slow if the optimal solution has large cost and there are many good looking (but ultimately futile) paths
Wavefront Planning

- Searching in along possible action sequences can be wasteful
  - Same cells are visited over and over again.

- Idea: Compute best path from every node to the goal
  - This is trivial for the goal node
  - It’s trivial for the nodes adjacent to the goal node
  - It’s trivial for the nodes adjacent to the nodes adjacent to the goal node
  - It’s trivial for nodes (n+1) away from the goal, given the best path for nodes (n) away from the goal.
Wavefront: Algorithm

\[
\text{wavefront} = \{ \text{goal} \} \\
\text{for all nodes } n \\
\quad \text{dist}(n) = \text{infinity} \\
\text{end for} \\
\text{dist}(\text{goal}) = 0 \\
\]

\[\text{do} \]
\[\quad \text{newwavefront} = \{ \} \]
\[\quad \text{for each node } n \text{ in wavefront} \]
\[\quad \quad \text{for each reachable neighbor } n' \text{ of } n \]
\[\quad \quad \quad \text{dist}(n') = \min(\text{dist}(n'), \text{dist}(n) + \text{cost}(n, n')) \]
\[\quad \quad \quad \text{newwavefront} = \{ \text{newwavefront}, n' \} \]
\[\quad \text{end for} \]
\[\text{end for} \]
\[\text{wavefront} = \text{newwavefront} \]
\[\text{until (wavefront contains start node)} \]

The algorithm computes cost-to-goal at every grid cell. Once computed, do gradient descent!
Wavefront, A*, Dijkstra

• In EECS492, you might be familiar with two variants of A*  
  ‣ Tree Search  
  ‣ Graph Search  
    - Only first path to any particular state is retained

• A* (GraphSearch) is essentially Wavefront

• Dijkstra’s Algorithm is a (slight) generalization of Wavefront  
  ‣ If different cells have different costs, then wavefront doesn’t propagate at uniform speed.  
  ‣ Don’t literally maintain a wavefront; maintain a front heap

• **Recommendation:**  
  ‣ Dijkstra is the best way to go in almost all cases
Configuration Space

• What if the robot takes up more than one grid cell?

• We could check for obstacles anywhere under the robot’s “footprint”
  ▶ Lots of collision tests!

• Can we pre-compute the set of acceptable locations?
Configuration Space

- Conventional collision checking using free space
  - Must check for collisions at every position occupied by the robot

- Configuration space
  - Precompute collision test for every state of the robot
    - Can be easily computed via convolution of obstacles with robot footprint
  - Collision check requires testing configuration space at a single point
Configuration Space

- What if our vehicle is not radially symmetric?
  - I.e., legality of a position depends not just on (x,y) but also θ?

- No problem!
  - Expand configuration space into third dimension
    - Pre-compute collision for all (x,y,θ) tuples.
    - Um, wait….
Approximate Configuration Spaces

- Solution 1: Use bounding circle of robot as footprint
  - Overly conservative
  - Collision checks = single point test in configuration space (fast).
Approximate configuration spaces

- Exploit kinematic constraints
  - Cars can only drive in a (mostly) straight line.
  - We need to perform collision tests for vehicle trajectories, not just stationary vehicles.
  - For a car, the collision test for the trajectory is coincident with the collision test for the vehicle body.
    - I.e., the same line test used to test trajectories for collisions can compute the collision test for the vehicle body “for free”!
    - (Actually, we have to extend the trajectory by the length of the car.)

- Huge win: Only need a 2D configuration space. Get speed and most of the accuracy of a 3D configuration space.
Multi-valued configuration:

- The “goodness” of a path is complex
  - Not just the minimum distance path
  - Not just a path that avoids obstacles

- For example:
  - Avoid obstacles by as much of a margin as possible (why?)
  - Well, not so much that we leave our lane
  - Well, maybe we shouldn’t run over the pedestrian either.

- We have trade-offs to make. How do we express them?
Multi-valued cost functions

Obstacle

Infeasible
High cost
Low cost

position

cost

Friday, March 29, 13
Real-world cost maps

- Take all your cost functions, add/max them up!

- Planner minimizes the integral of cost along trajectory
  - Includes a total distance component
  - Includes penalties for “close calls”
Approximate configuration spaces

- Step 1: Build configuration space for a useful shape
  - E.g., a circle whose diameter = width of car

- A collision test now becomes a line test in configuration space
- Only slightly conservative, at the expense of more configuration space testing.
Handling continuous search spaces

- Continuous action space makes exact answer intractable (in general)
- We can sometimes consider only a few discrete actions
  - Consider “nudges” and “veers”

- Shortcomings
  - Algorithm is no longer complete.
  - Paths are not optimal
Non-deterministic Planning

- Consider sequences of random actions
  - Build a tree of actions
    - Node = state
    - Edge = action
  - A plan is a sequence of actions, i.e., a path from the root (initial state) to a leaf near the goal

- Skeleton algorithm:
  - while time remains
    - Select a node (call it parent) in the tree
    - Generate a new action a
    - Create new node: child = propagate(parent, a)
    - If action is safe
      - Add node to tree
  - return best plan found so far

Implementation here is critical
Taking into account kinematic and dynamic constraints

- These forward-search methods can easily take into account kinematic and dynamic constraints
  - Action propagation can correspond to arbitrary non-linear transformation
  - Typical: run differential equations forward
Random Tree Growth

- Random: Pick a parent node at random. Pick an action at random.
Random Policy

- Complete?
- Optimal?
- Practical?

Complete: yes, as time -> infinity
Optimal: yes, as time -> infinity
Practical: not a chance.
RRT Policy

- RRT: Pick a destination at random. Find parent node that is closest to destination. Compute action that goes towards destination
RRT-Biased Policy

- RRT-Biased: Pick the destination randomly, but in a biased way (i.e., prefer directions that are heuristically likely to work out)
RRT: Trunking

- State of tree at $t_i$ limits
  state of tree at $t_{i+1}$

- New edges always connect
  closest part of tree: never
  generate a completely new
  route to an area we
  already have a route to

- How could you address
  this?
RRT: Analysis

• Complete?

• Optimal?

• Practical?

Complete: No
Optimal: No
Practical: sure, maybe.
RRT on an Arm