Notes

- Note: JCam resolution change
- Course feedback
- Quiz return
Quiz 1 Results

- $E(x) = 21.68$
- $\sigma(x) = 2.88$
- $85\% = 25.5$

An obvious bug

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

Based on Principles of Robot Motion, Choset et al.
The obvious bug has a 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

A better bug
Bug 1

- This bug is special: it's provably complete.
  - Definitions: Let $d()$ be the distance to goal. Let entry, be the point at which we encountered the $i$th obstacle, exit, 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., $d(\text{entry}_i) \leq d(\text{exit}_i)$.

**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 ${d(\text{entry}_1), d(\text{exit}_1), d(\text{entry}_2), d(\text{exit}_2), d(\text{entry}_3), d(\text{exit}_3), \ldots}$ is monotonically decreasing by Lemma 1 and 2.

### 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
  \[ D + \sum_{i} \frac{n_i}{2} P_i \]
Motion planning with complete information

- 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?

A*

- For each node in search tree, compute two quantities:
  - cost-so-far
  - min-cost-to-go

- Expand the node with the minimum value of:  
  cost-so-far + min-cost-to-go

- If min-cost-to-go is admissible, A* is optimal and complete
  - In fact, A* is optimally efficient: no algorithm can guarantee optimal solutions while expanding fewer nodes
A* Algorithm

```
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.min-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 suboptimal answer x. This means that cost(x) > 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 cost-so-far(p) + cost-to-go(p) <= 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 suboptimal node.
A* Heuristic Function

- As we see, optimality depends on heuristic function never over-estimating cost-to-go.

- Tighter estimates $\rightarrow$ fewer nodes expanded

- Bad news: If heuristic function under-estimates true cost by more than log(actual cost), A* has exponential complexity.
  - Is this the common case or not?

A* for high-level planning

- Finding routes in road networks
A* Applications

A* for low-level planning

- Vanilla A* not a great choice for most low-level planning problems, unless:
  - Optimal results are required
  - Relatively small number of actions
  - Relatively large number of possible states

- Variants on A*
  - IDA*
  - MDA
**IDA***

- **Cost-limited-A**(problem, cost-limit)
  - Just like A*, except does not expand any nodes whose cost underestimate exceeds cost-limit.
  - If failure, returns cost-limit of smallest un-expanded node

**IDA**(problem)

```plaintext
limit = \epsilon

do forever
  \{solution, next-cost\} = Cost-Limited-A*(problem, limit)
  if solution not null
    return solution
  limit = max(next-cost, limit + \epsilon)
end do
```

**How can this not be a terrible idea?**
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 Planning: Algorithm

wavefront = { goal }
for all nodes n
    dist(n) = infinity
end for
dist(goal) = 0

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

Wavefront Planning: Algorithm 2

- Previous algorithm computes minimum distance from every point to goal.

- How do we get the solution from these distances?
Wavefront: Summary

- Pros
  - Complete
  - Optimal
  - Extremely fast when state space is small
    - Action space affects performance only modestly.
  - Only have to recomputed when goal point changes.

- Cons
  - What happens if vehicle state is multi-dimensional or continuous?
We’ve assumed our robot is a point so far

But real robots take up space!
- In PS4 we solve this problem by checking for collisions in multiple locations corresponding to the size of the robot. This takes a lot of CPU time!

Can we pre-compute a set of states that are collision-free?
- Constrain our search to these states

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 \(\theta\)?

- No problem!
  - Expand configuration space into third dimension
    - Pre-compute collision for all \((x,y,\theta)\) 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

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.

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.
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: \( \text{child} = \text{propagate}(\text{parent}, a) \)
    - If action is safe
      - Add node to tree
  - return best plan found so far

Planning Variants

- We'll consider three basic variants
  - Random: Pick a parent node at random. Pick an action at random.
  - RRT: Pick a destination at random. Find parent node that is closest to destination. Compute action that goes towards destination
  - RRT-Biased: Pick the destination randomly, but in a biased way (i.e., prefer directions that are heuristically likely to work out)
Random Policy

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

Random Policy: Analysis

- Complete?
- Optimal?
- Practical?
RRT Policy

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

---

RRT Policy (Biased Sampling)

- **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 for bicycle: Mini-Quiz

- What is the action space for the bicycle?

- What are the kinematic constraints?
  - How do we satisfy them?

- What are the dynamic constraints?
RRT: Analysis

- Complete?
- Optimal?
- Practical?