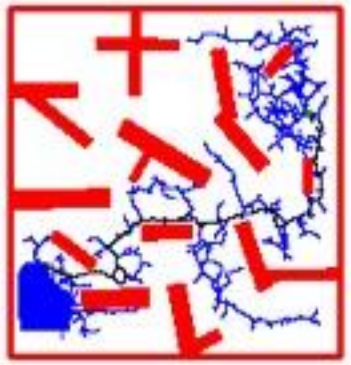
Robotic Path Planning

McGill COMP 417

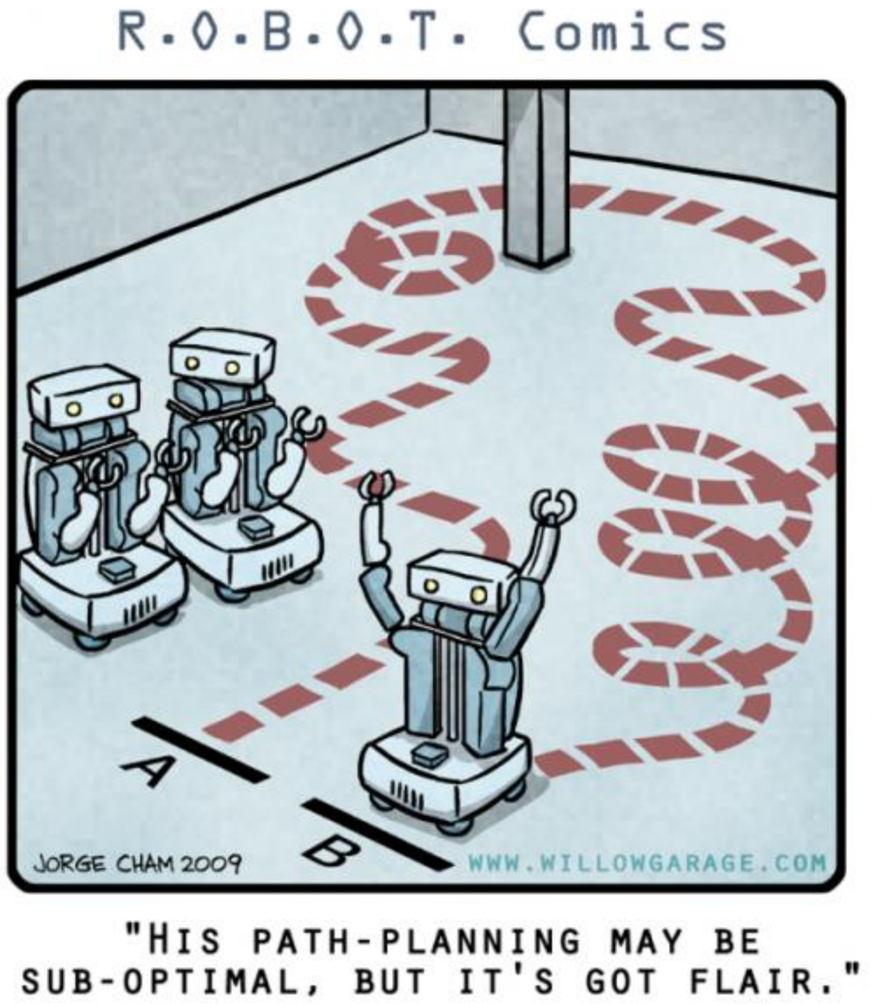
Sept 12th, 2019



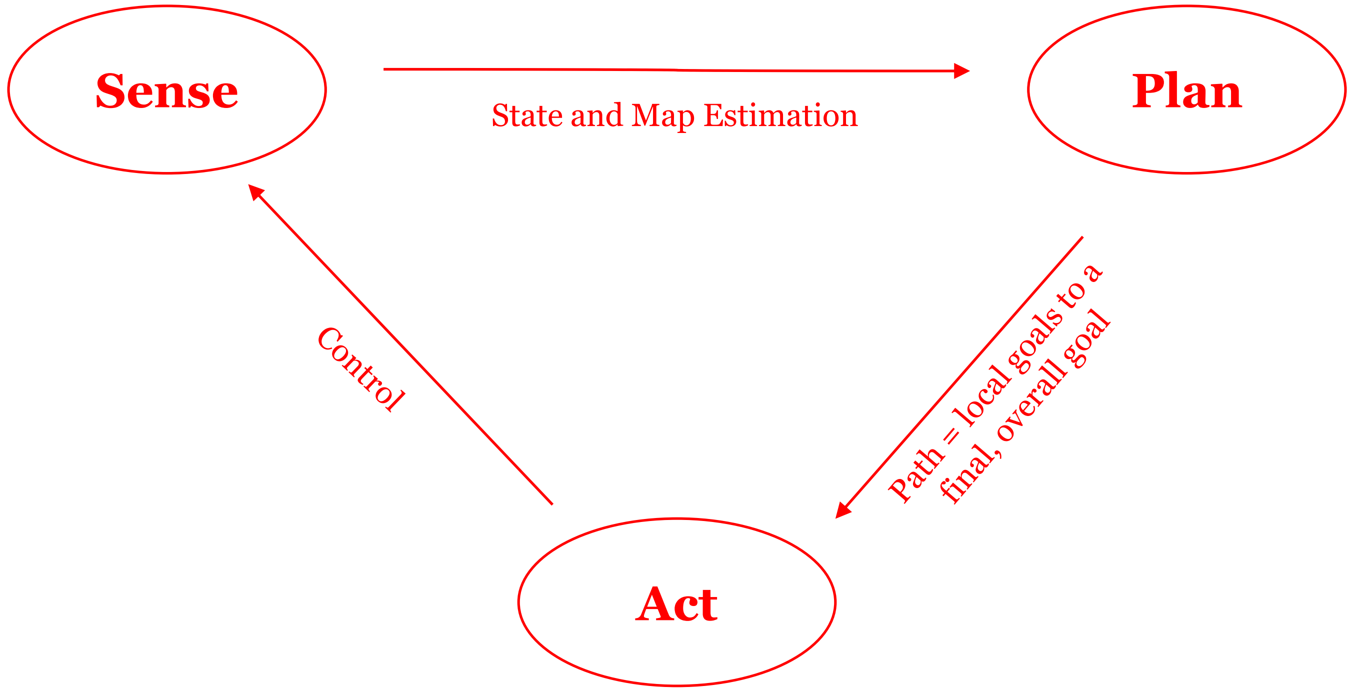
Outline

- Planning definition and philosophy
- Interfaces to kinematics and maps
- Graph-based planning
- Planning with Dynamic Programming

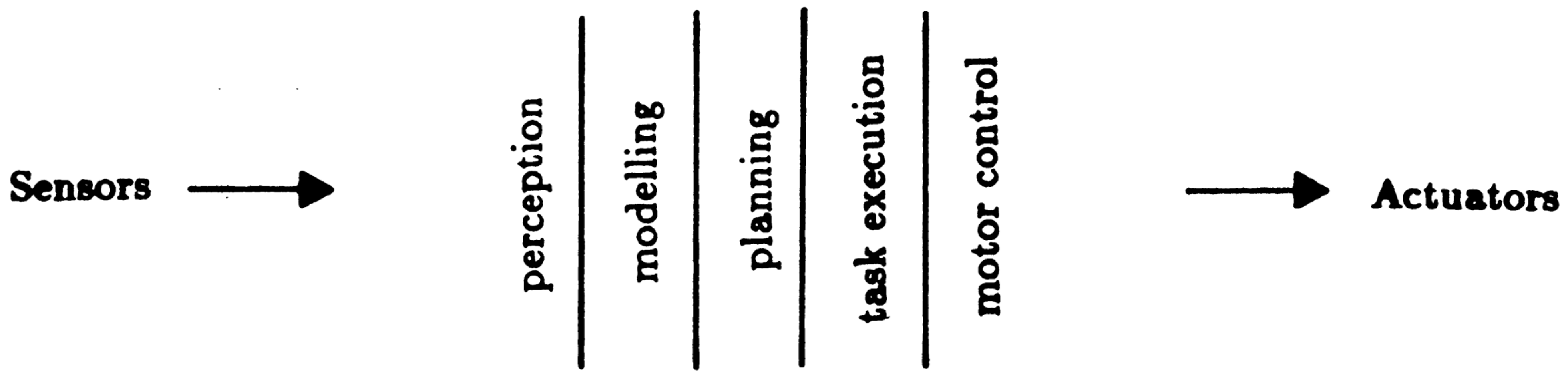
Why Bother Planning?



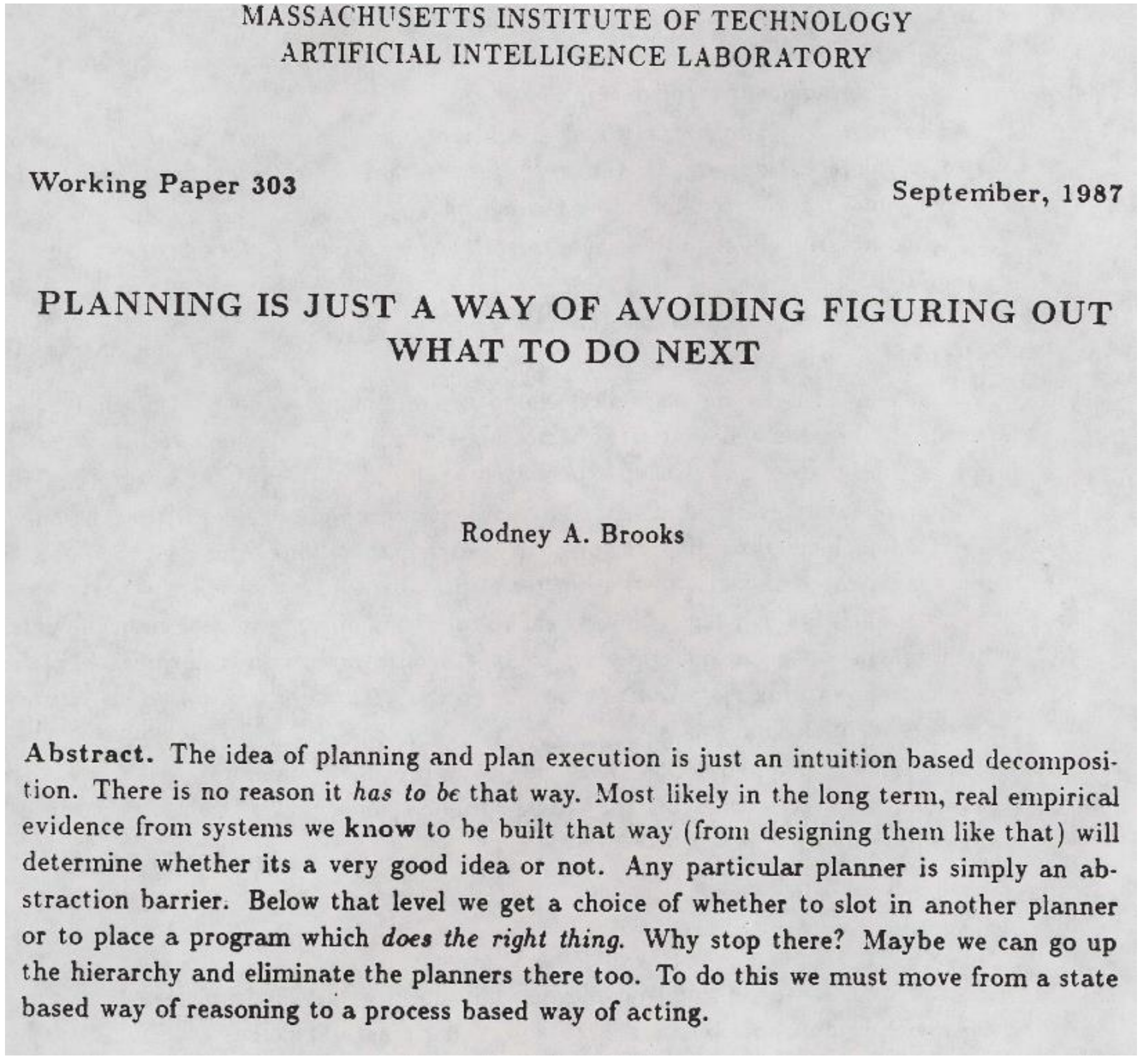
Sense-Plan-Act Paradigm: Planning Is Necessary



Planning-based or Deliberative Architecture



Subsumption Architecture: Planning Is Not Necessary



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

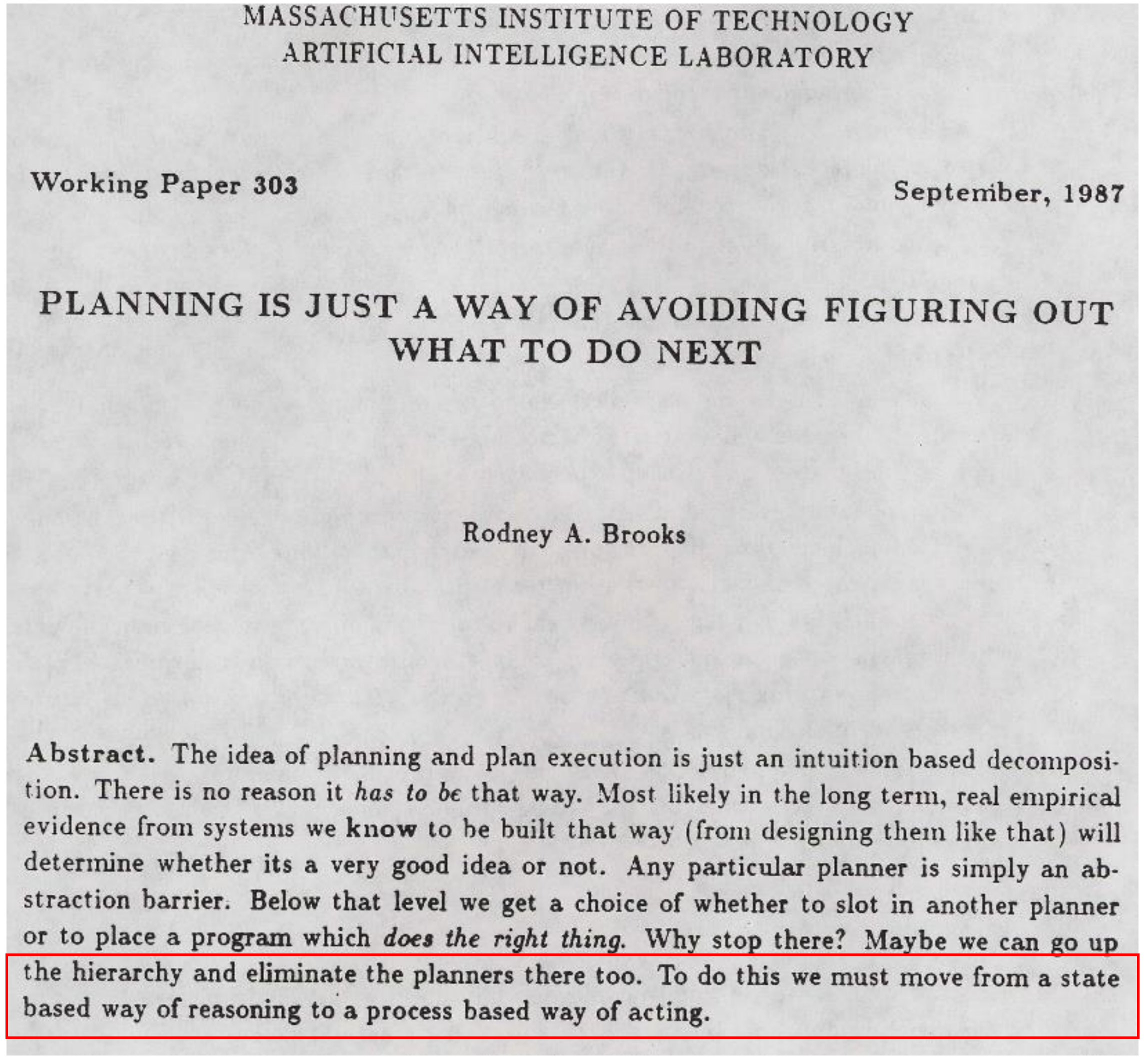
Working Paper 303 September, 1987

PLANNING IS JUST A WAY OF AVOIDING FIGURING OUT WHAT TO DO NEXT

Rodney A. Brooks

Abstract. The idea of planning and plan execution is just an intuition based decomposition. There is no reason it *has to be* that way. Most likely in the long term, real empirical evidence from systems we **know** to be built that way (from designing them like that) will determine whether its a very good idea or not. Any particular planner is simply an abstraction barrier. Below that level we get a choice of whether to slot in another planner or to place a program which *does the right thing.* Why stop there? Maybe we can go up the hierarchy and eliminate the planners there too. To do this we must move from a state based way of reasoning to a process based way of acting.

Subsumption Architecture: Planning Is Not Necessary



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 303 September, 1987

PLANNING IS JUST A WAY OF AVOIDING FIGURING OUT WHAT TO DO NEXT

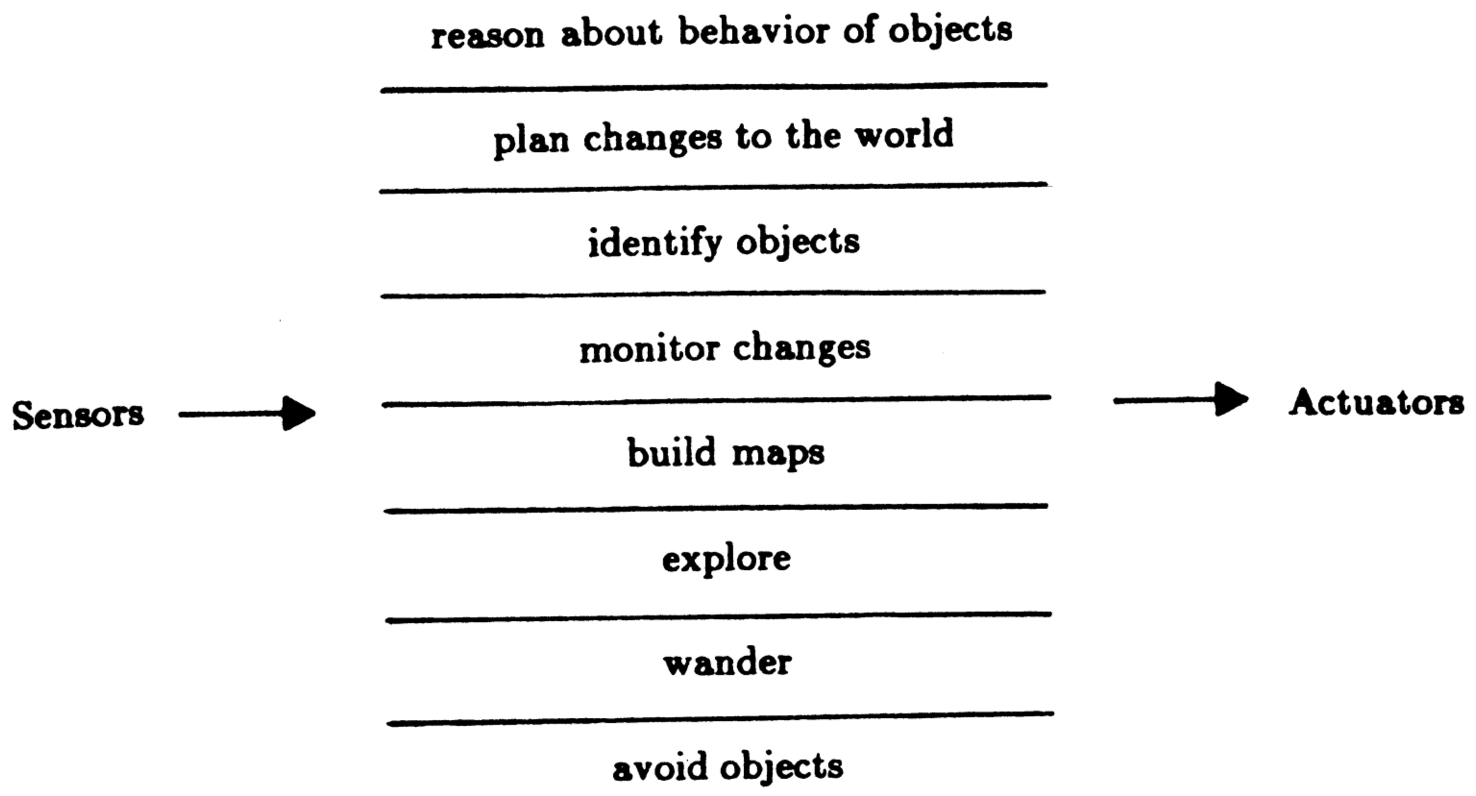
Rodney A. Brooks

Abstract. The idea of planning and plan execution is just an intuition based decomposition. There is no reason it *has to be* that way. Most likely in the long term, real empirical evidence from systems we know to be built that way (from designing them like that) will determine whether its a very good idea or not. Any particular planner is simply an abstraction barrier. Below that level we get a choice of whether to slot in another planner or to place a program which *does the right thing*. Why stop there? Maybe we can go up the hierarchy and eliminate the planners there too. To do this we must move from a state based way of reasoning to a process based way of acting.

He means: why bother estimating state and planning? It's too much work and could be error-prone. Why not only have a hierarchy of reactive behaviors/controllers?

One possibility:
instead of u(state)=...
use u(sensory observation)=...

Subsumption Architecture: Planning Is Not Necessary



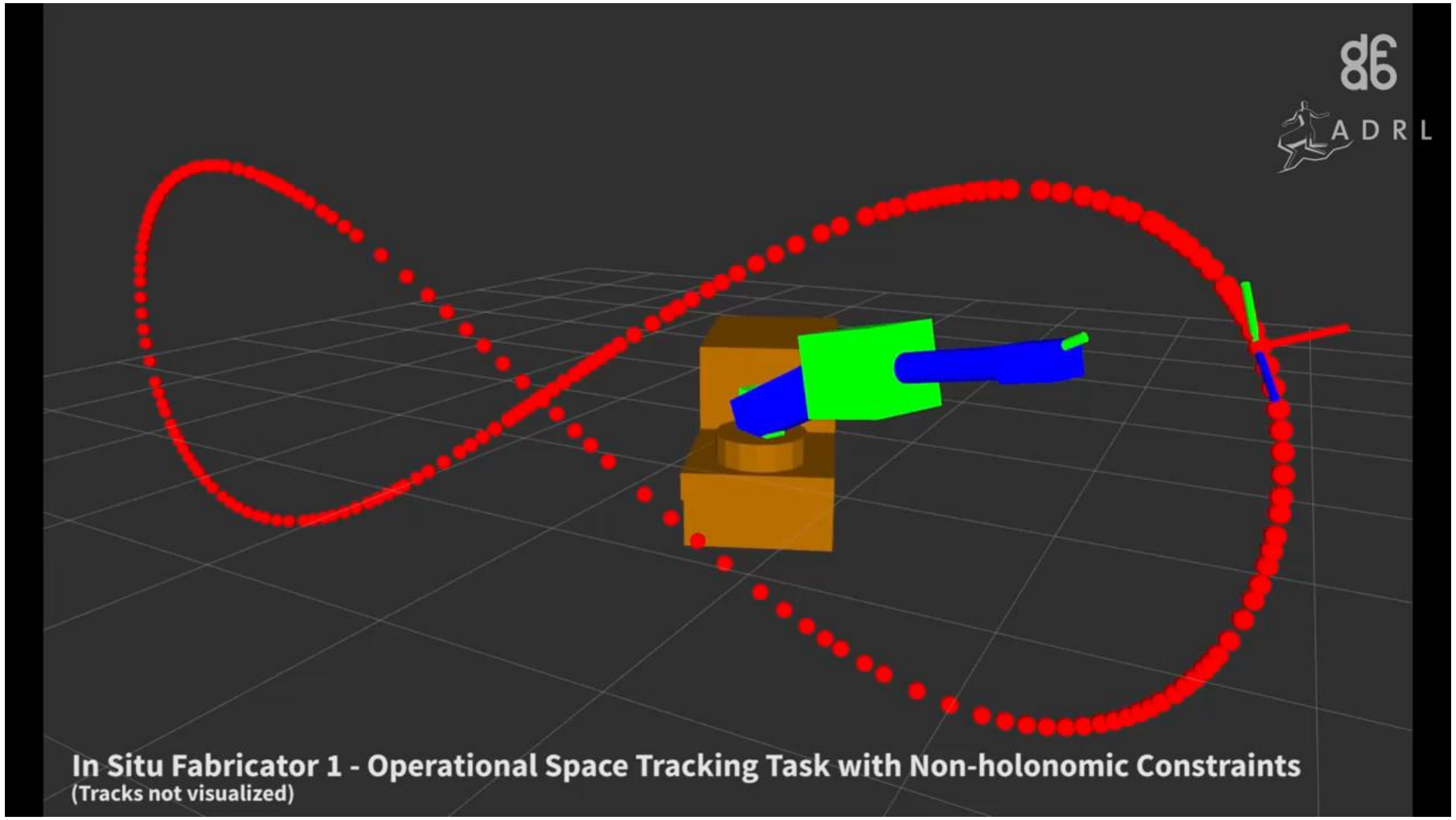
Planning Philosophy

- Somewhere in between full deliberation and sub-sumption is typical for robot systems today

- Do build plans, construct with careful algorithms, try your best to act upon them, BUT:
  - Be willing to re-plan because no model is perfect and the world changes
  - Make planning aware of sensing and action considerations. Leads to more complex planners, but more robust behavior

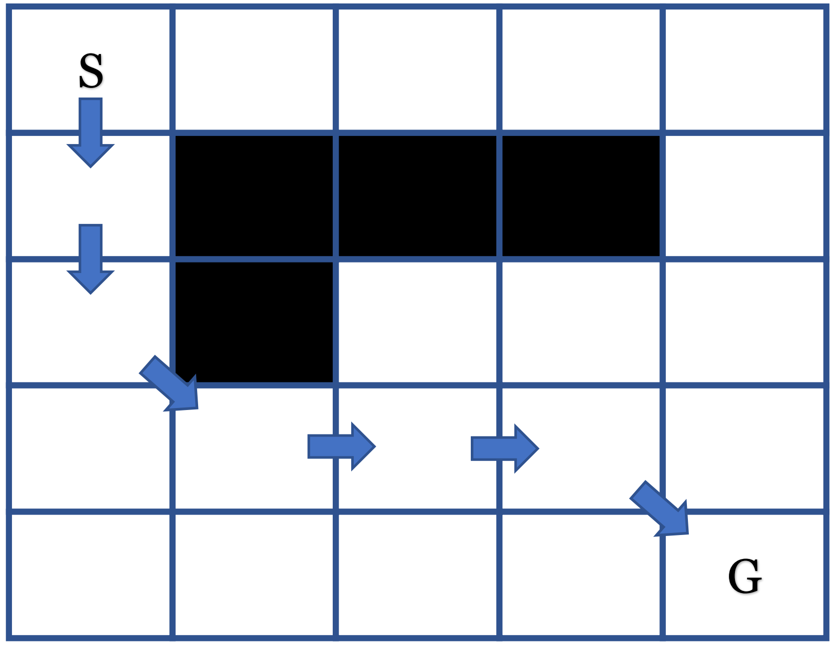
- For this unit, we consider the full Deliberative mode. Forget physics and sensors for a while (we'll get back to them).

Deliberative Planning: Think Ahead


ADRL
In Situ Fabricator 1 - Operational Space Tracking Task with Non-holonomic Constraints
(Tracks not visualized)

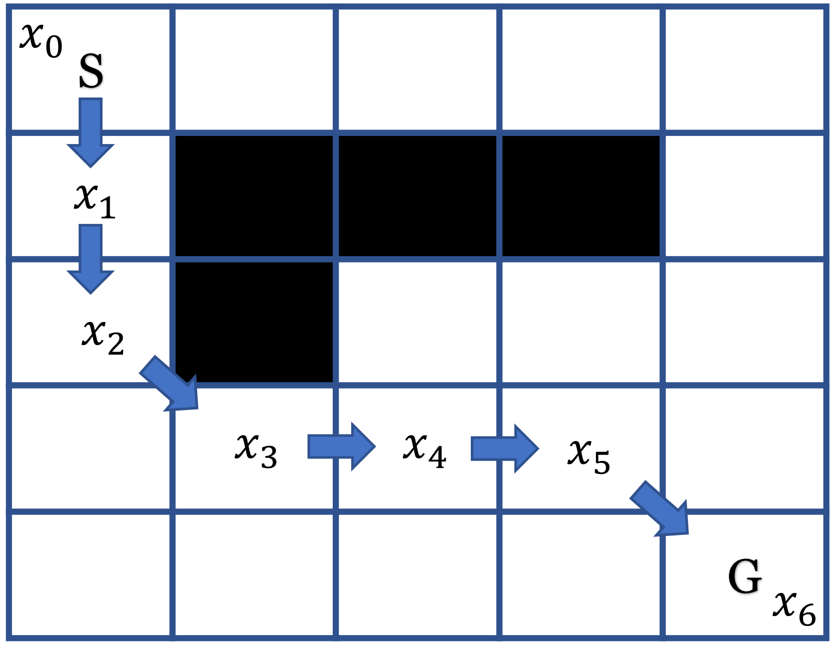
What is a plan?

- Intuitively, a path from start to goal
- The robot should be able to follow this path under its kinematics
- The path should not take the robot through any obstacles in the map



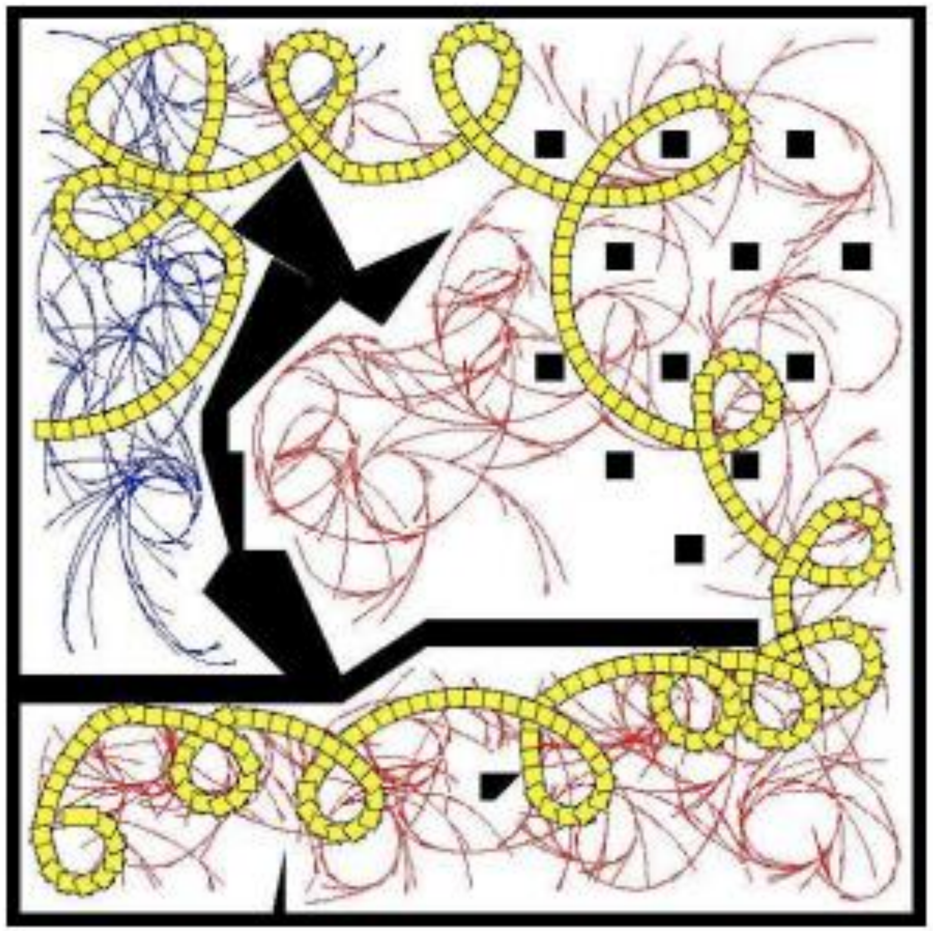
What is a plan, more formally?

- A plan is a sequence of states:
  $$P = \{x_0, x_1, \dots, x_t, \dots x_n\}$$
- A ***feasible*** plan is one where:
  1. $x_0 = S$ (it starts at the start)
  2. $x_n = G$ (it ends at the goal)
  3. each state x_t is feasible
  4. each transition between subsequent states, (x_t, x_{t+1}) is feasible
- Feasibility of states and transitions involves kinematics and maps



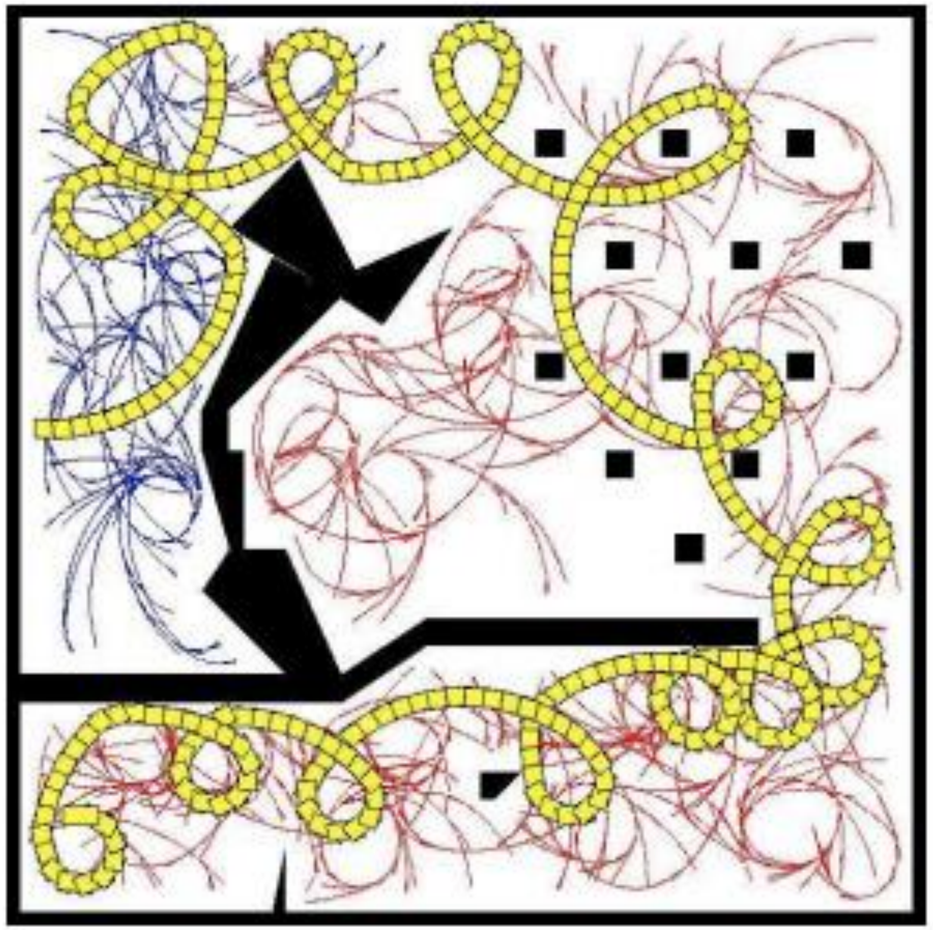
Planning Algorithms

- Take as input a start, a goal, kinematic constraints and a map
- Output a path/plan, or "impossible"

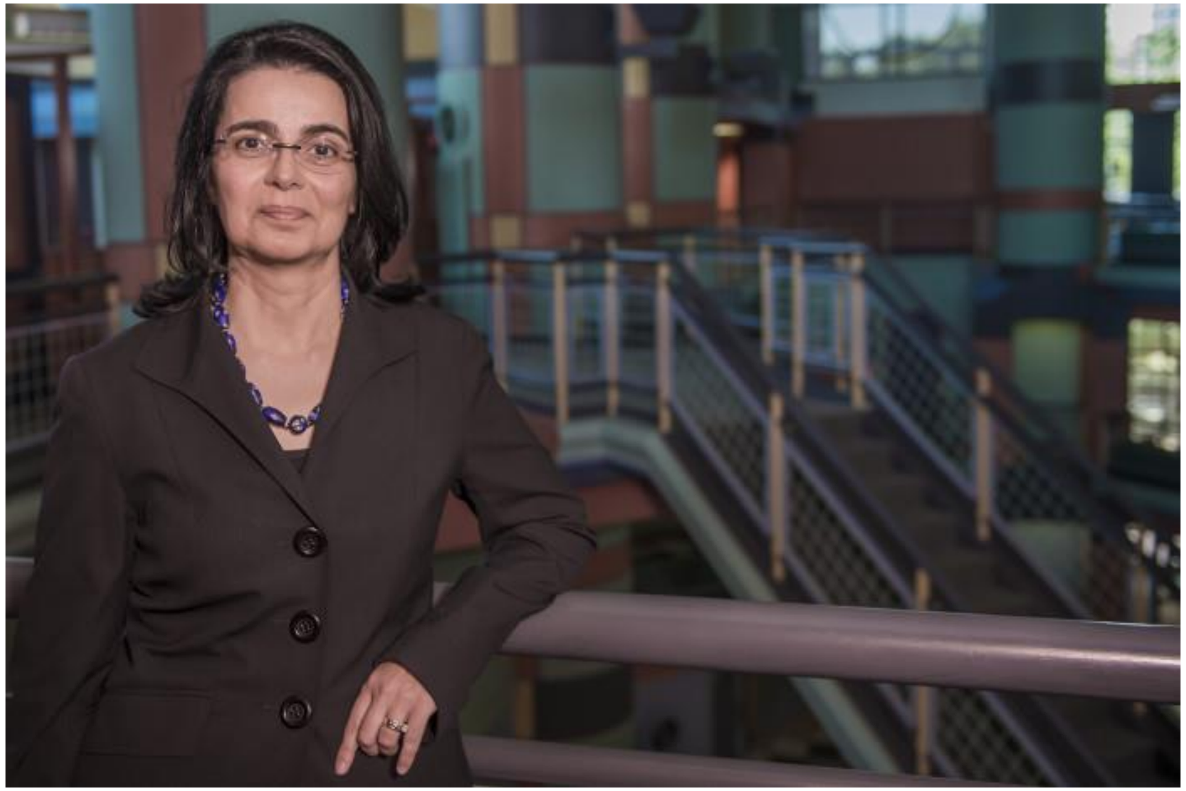


Planner Properties

- Planning algorithms are:
  - ***Correct*** if every plan they output is feasible
  - ***Complete*** if they find a plan whenever one exists
  - ***Terminating*** if they execute in a finite time (guaranteed)
- ***Complexity*** is measured by the worst-case memory and run-time in the map/robot dimensions, path length, etc.

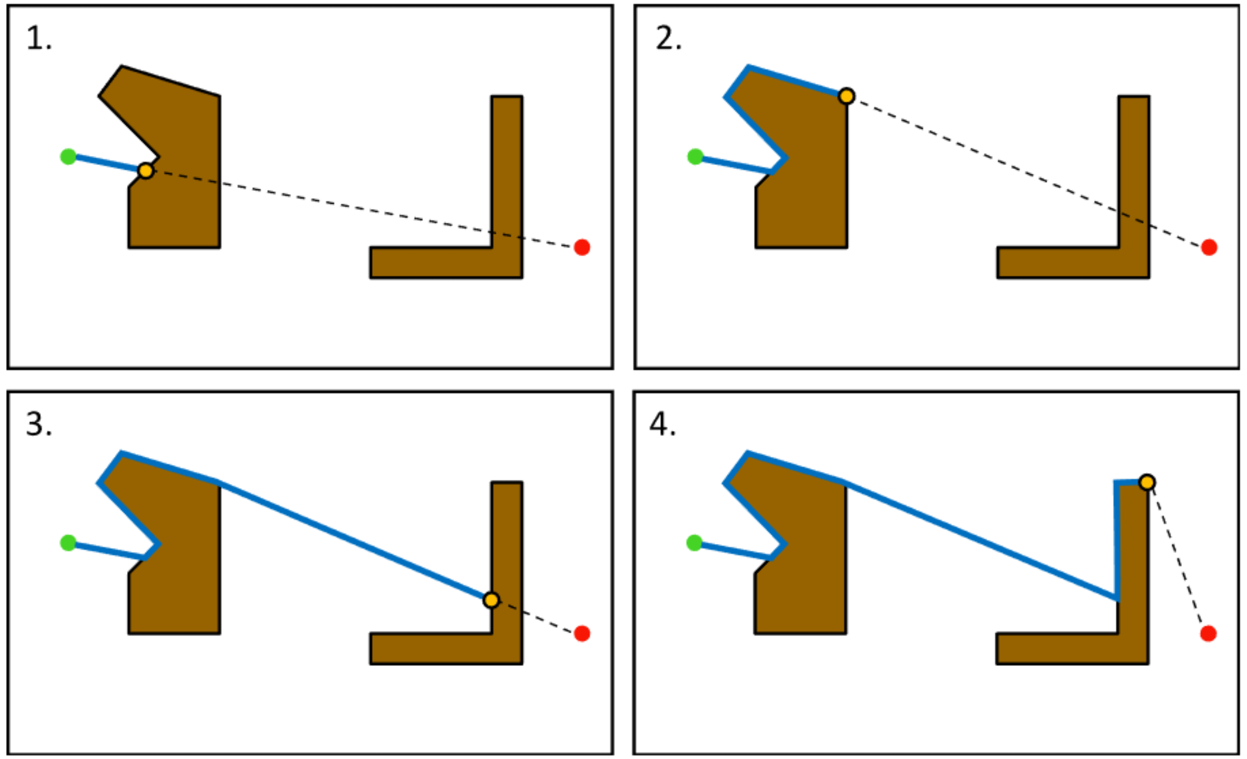


Name that Roboticist



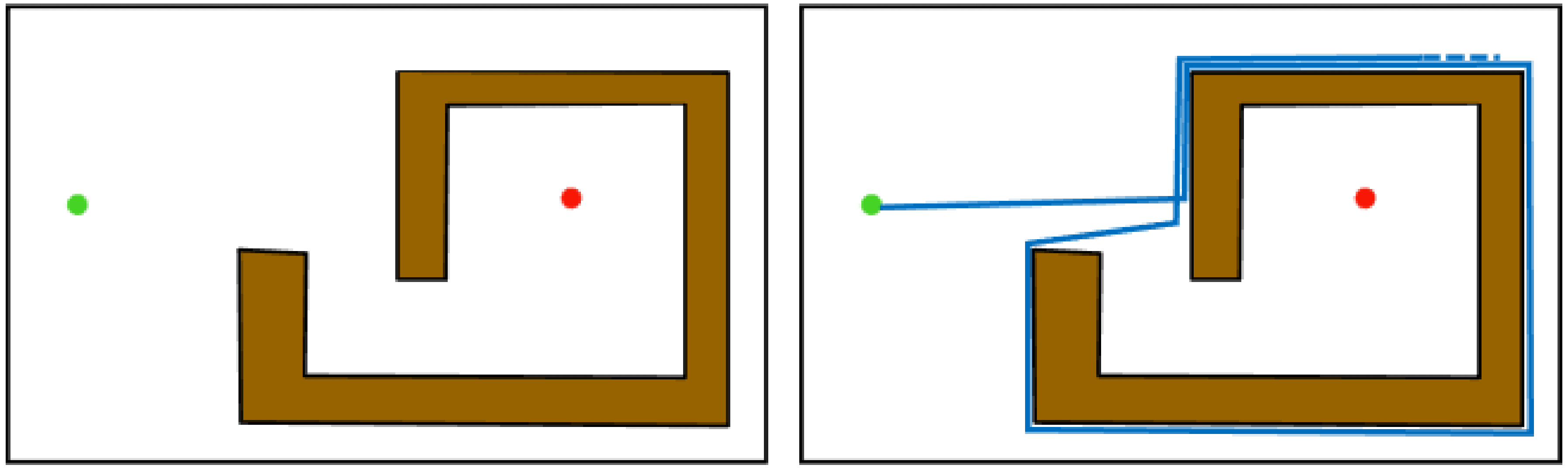
Our first "planner": The Bug Algorithm

- Intuition of human labyrinth solving: "Always turn left"
- Does this work?



1. Move in a straight line path toward G.
2. If the robot reaches G, we are done.
3. If the robot hits an obstacle, then follow the right-hand rule and traverse its boundary
4. This continues until the wall no longer is pointing against the straight line from X toward G. Once this occurs, continue from Step 1.

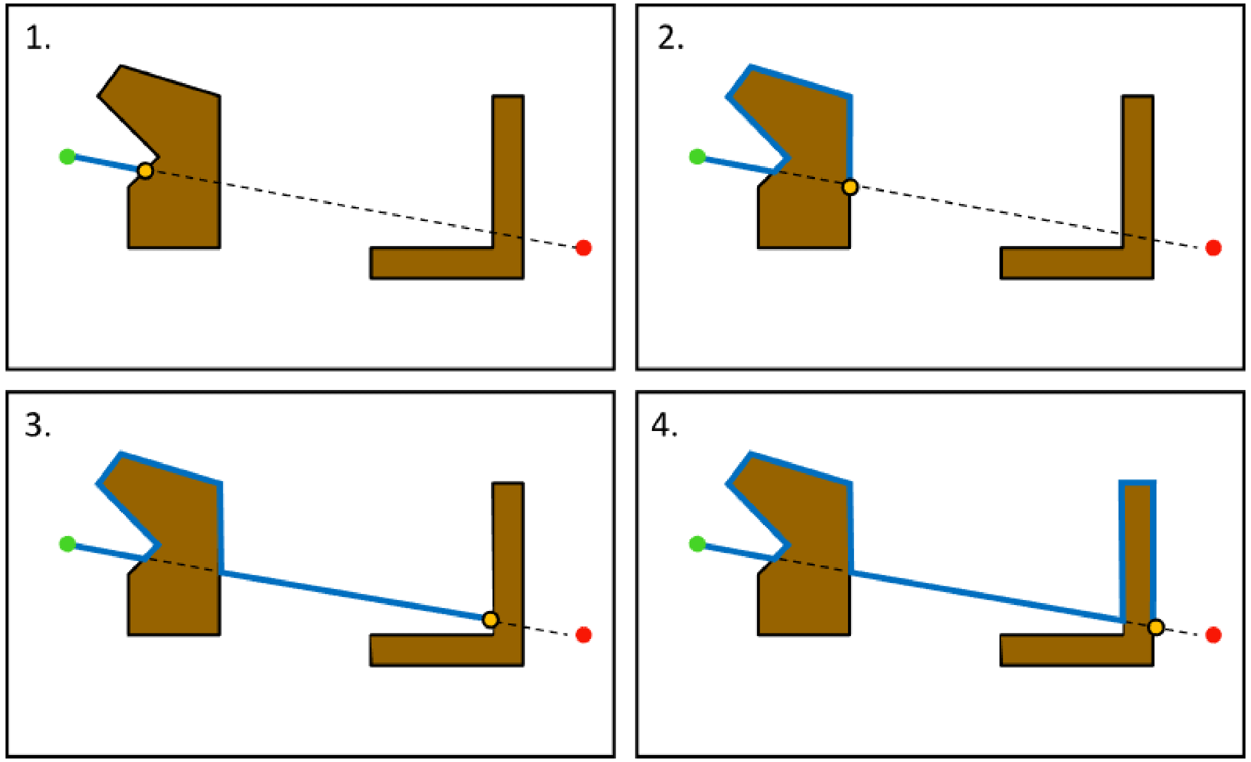
Bug Alg. #1 Counter-example



1. Move in a straight line path toward G.
2. If the robot reaches G, we are done.
3. If the robot hits an obstacle, then follow the right-hand rule and traverse its boundary
4. This continues until the wall no longer is pointing against the straight line from X toward G. Once this occurs, continue from Step 1.

Our second "Planner": Bug #2

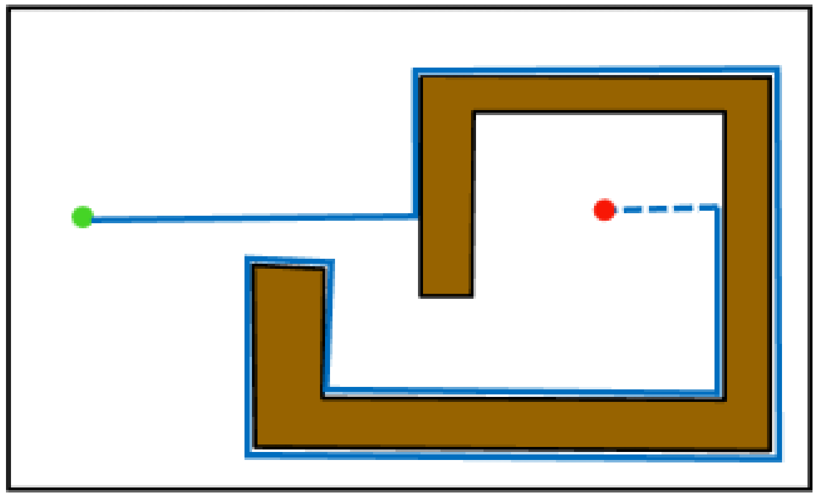
- Intuition: we needed to follow the boundary a bit farther to ensure we maintain progress



1. Move in a straight line path toward G.
2. If the robot reaches G, we are done.
3. If the robot hits an obstacle, remember the point P at which it is hit. Follow the right hand-rule.
4. If the robot crosses the line $\overline{PG}$ as it walks along the wall, determine whether $\|X - G\| < \|P - G\|$ and the wall is not pointing against the straight line from X toward G. If both conditions are satisfied, continue from step 1. Otherwise, keep walking.
5. If P is reached again, return "No path"

Can you prove Bug #2 is complete?

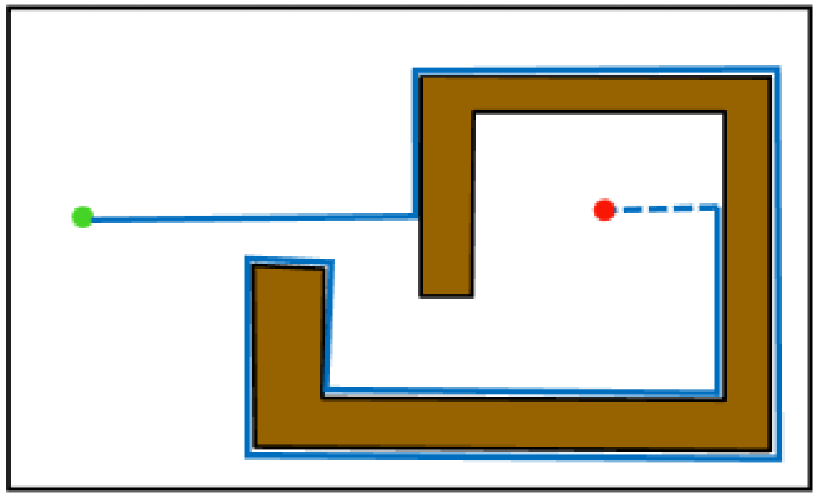
- Recall complete means a feasible path is always found if there is one.

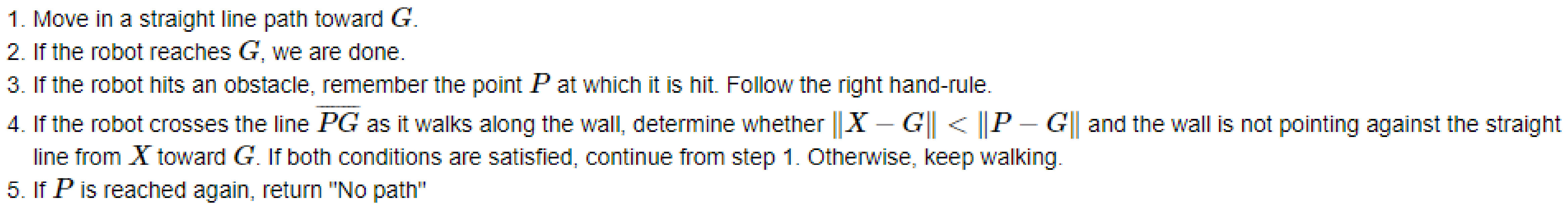


1. Move in a straight line path toward G.
2. If the robot reaches G, we are done.
3. If the robot hits an obstacle, remember the point P at which it is hit. Follow the right hand-rule.
4. If the robot crosses the line $\overline{PG}$ as it walks along the wall, determine whether $\|X - G\| < \|P - G\|$ and the wall is not pointing against the straight line from X toward G. If both conditions are satisfied, continue from step 1. Otherwise, keep walking.
5. If P is reached again, return "No path"

Yes, Bug #2 is complete

- Proof of constantly reducing distance to goal:
  - A. For each obstacle, Bug 2 will break away from it at some point if there is a path to the goal.
  - B. The sequence of hit points decreases in distance to the goal.
  - C. If there is no path to the goal, then Bug 2 will terminate correctly with failure.





1. Move in a straight line path toward G.
2. If the robot reaches G, we are done.
3. If the robot hits an obstacle, remember the point P at which it is hit. Follow the right hand-rule.
4. If the robot crosses the line $\overline{PG}$ as it walks along the wall, determine whether $\|X - G\| < \|P - G\|$ and the wall is not pointing against the straight line from X toward G. If both conditions are satisfied, continue from step 1. Otherwise, keep walking.
5. If P is reached again, return "No path"

Planning as graph search

- Graph nodes represent discrete states
- Edges represent transitions/actions
- Edges have weights
  - E.g., formed by lookup into the map's grid: the distance to cross the grid cell if we can traverse, infinite if an obstacle

- Potential queries:
  - Shortest path from node a to node b, that does not hit obstacles
  - Is b reachable from a?

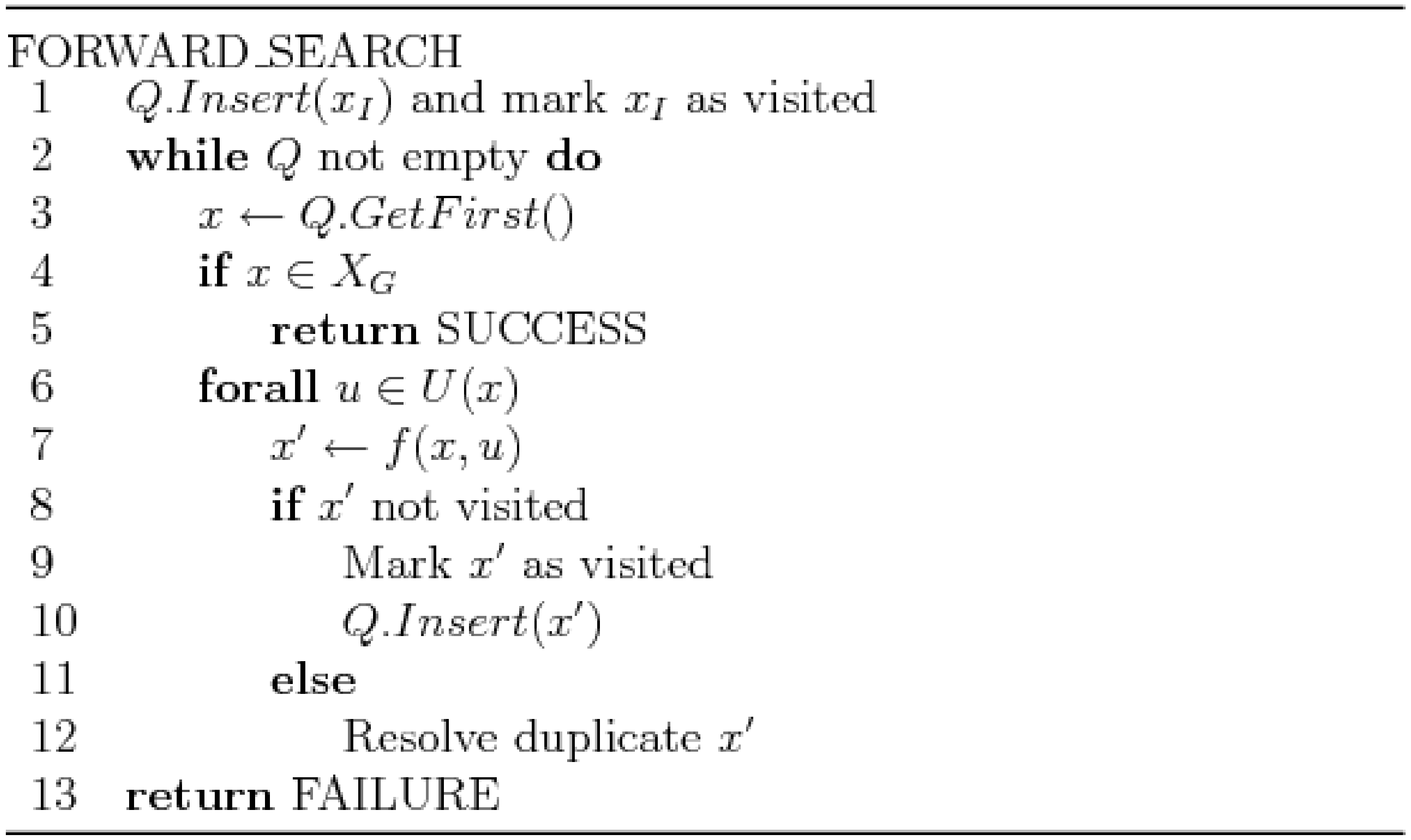
Planning as graph search

- Graph nodes represent discrete states
- Edges represent transitions/actions
- Edges have weights

- Potential queries:
  - Shortest path from node a to node b, that does not hit obstacles
  - Is b reachable from a?

- Typical assumptions:
  - Current state is known
  - Map is known
  - Map is mostly static

Graph planning as forward search



```
FORWARD_SEARCH
$Q.Insert(x_I)$ and mark $x_I$ as visited
while $Q$ not empty do
    $x \leftarrow Q.GetFirst()$
    if $x \in X_G$
        return SUCCESS
    forall $u \in U(x)$
        $x' \leftarrow f(x, u)$
        if $x'$ not visited
            Mark $x'$ as visited
            $Q.Insert(x')$
        else
            Resolve duplicate $x'$
return FAILURE
```

Figure 2.4: A general template for forward search.

Dynamic Programming

$$D(v) = \min_{u \in N(v)} [d(v, u) + D(u)]$$

$$D(v_{\text{dest}}) = 0$$

Cost-to-go to destination starting from node v

Instantaneous transition cost needs to be non-negative

Dynamic Programming

$$D(v) = \min_{u \in N(v)} \left[d(v,u) + D(u) \right]$$

$$D(v_{\text{dest}}) = 0$$

Neighbors of v.
i.e. available actions

Instantaneous transition
cost needs to be non-negative

Cost-to-go to destination
starting from node v

Base case

Dynamic Programming

Worst-Case Complexity:

$O(|V|^2)$

In 2D grid world

$O(|V|)$

$$D(v) = \min_{u \in N(v)} [d(v,u) + D(u)]$$

$$D(v_{\text{dest}}) = 0$$

Cost-to-go to destination node starting from node v. Could also have denoted it $D(v, v_{\text{dest}})$

Base case

Instantaneous transition cost for adjacent nodes needs to be non-negative

Dijkstra's algorithm

- Let $D(v)$ denote the length of the optimal path from the source node to node v (i.e. cost-to-come, not cost-to-go like before)
- Set $D(v) = \infty$ for all nodes except the source: $D(v_{\mathrm{src}}) = 0$
- Add all nodes to priority queue Q with cost-to-come as priority
- While Q is not empty:

Dijkstra's algorithm

- Let $D(v)$ denote the length of the optimal path from the source node to node v (i.e. cost-to-come, not cost-to-go like before)
- Set $D(v) = \infty$ for all nodes except the source: $D(v_{\text{src}}) = 0$
- Add all nodes to priority queue Q with cost-to-come as priority
- While Q is not empty:
  - Extract the node v with minimum cost-to-come from the queue Q
  - If found goal then done
  - Remove v from the queue

    The cost-to-come of v is final at this point.
    Need to check if we can reduce the cost-to-come of its neighbors.

Dijkstra's algorithm

- Let $D(v)$ denote the length of the optimal path from the source node to node v (i.e. cost-to-come, not cost-to-go like before)
- Set $D(v) = \infty$ for all nodes except the source: $D(v_{\text{src}}) = 0$
- Add all nodes to priority queue Q with cost-to-come as priority
- While Q is not empty:
  - Extract the node v with minimum cost-to-come from the queue Q
  - If found goal then done
  - Remove v from the queue

    The cost-to-come of v is final at this point.

    Need to check if we can reduce the cost-to-come of its neighbors.
  - For u in neighborhood of v :
    - If $d(u,v) + D(v) < D(u)$ then
      - Update priority of u in Q to be $d(u,v) + D(v)$

Dijkstra's algorithm

- Let $D(v)$ denote the length of the optimal path from the source node to node v (i.e. cost-to-come, not cost-to-go like before)
- Set $D(v) = \infty$ for all nodes except the source: $D(v_{\text{src}}) = 0$
- Add all nodes to priority queue Q with cost-to-come as priority
- While Q is not empty:
    - Extract the node v with minimum cost-to-come from the queue Q
    - If found goal then done
    - Remove v from the queue

    The cost-to-come of v is final at this point.
    Need to check if we can reduce the cost-to-come of its neighbors.
    - For u in neighborhood of v :
        - If $d(u,v) + D(v) < D(u)$ then
            - Update priority of u in Q to be $d(u,v) + D(v)$

$O(1)$

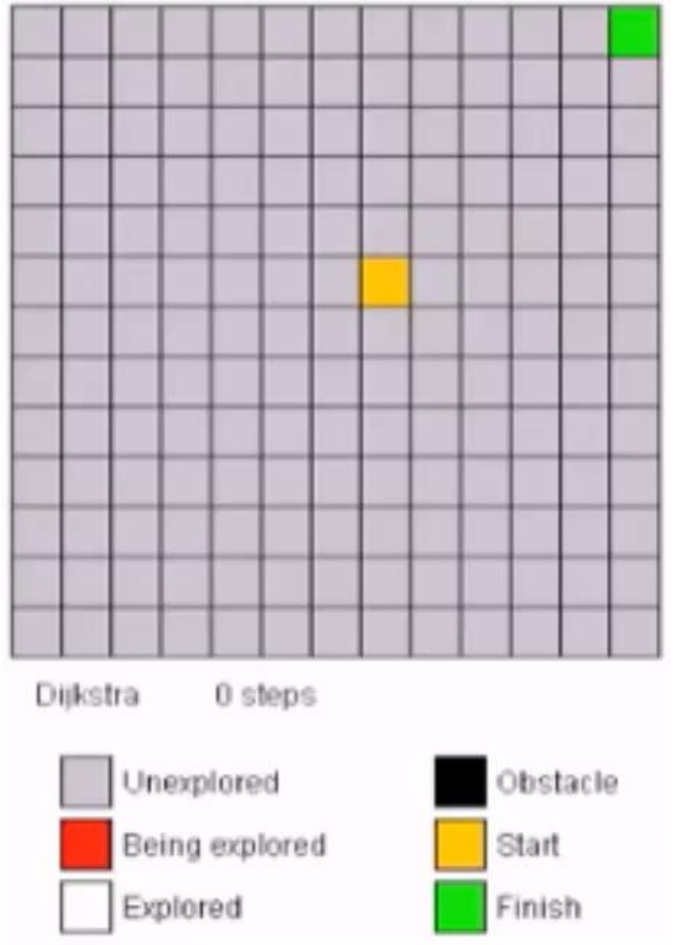
$O(\log|V|)$

$O(1)$

For Fibonacci heaps

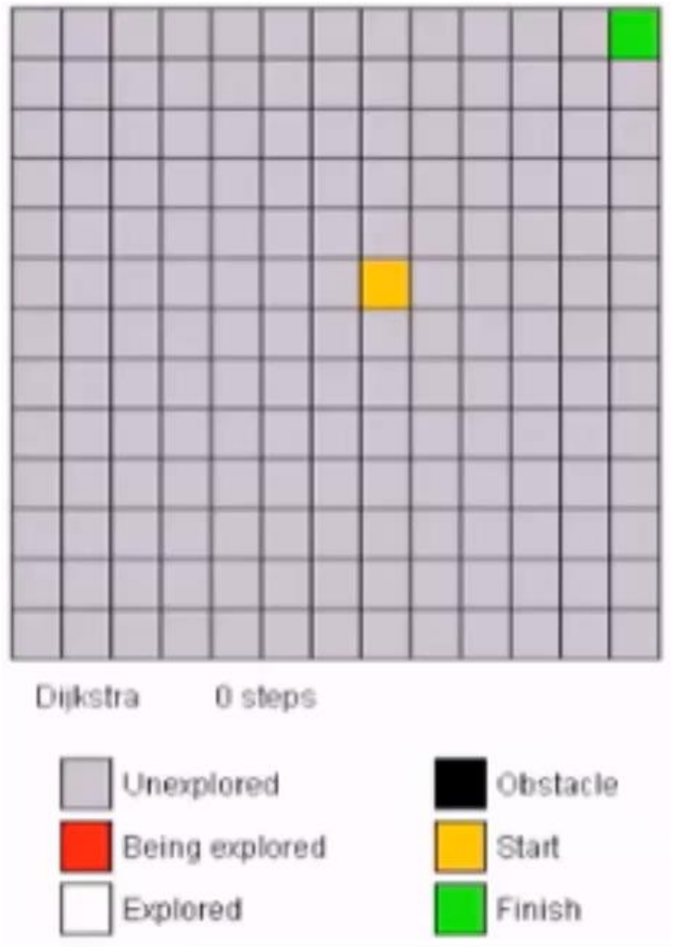
$$O(|E|T_{\text{update priority}} + |V|T_{\text{remove min}}) = O(|E| + |V|\log|V|)$$

Dijkstra’s algorithm: example runs



Dijkstra’s algorithm: example runs

Many nodes are explored unnecessarily. We are sure that they are not going to be part of the solution.



A* Search: Main Idea

- Modifies Dijkstra's algorithm to be more efficient
- Expands fewer nodes than Dijkstra's by using a heuristic

- While Dijkstra prioritizes nodes based on cost-to-come
- A* prioritizes them based on:

cost-to-come to v + lower bound on cost-to-go from v to v_{dest}

$$f(v) = \underbrace{g(v)}_{\text{cost-to-come}} + \underbrace{h(v)}_{\text{lower bound on cost-to-go}}$$

Lower bound on cost of path from source to destination that passes through v $\rightarrow f(v)$

Optimistic search: explore node with smallest f(v) next

A* Search: Main Idea

- Modifies Dijkstra's algorithm to be more efficient
- Expands fewer nodes than Dijkstra's by using a heuristic

- While Dijkstra prioritizes nodes based on cost-to-come
- A* prioritizes them based on:

cost-to-come to v + lower bound on cost-to-go from v to v_{dest}

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

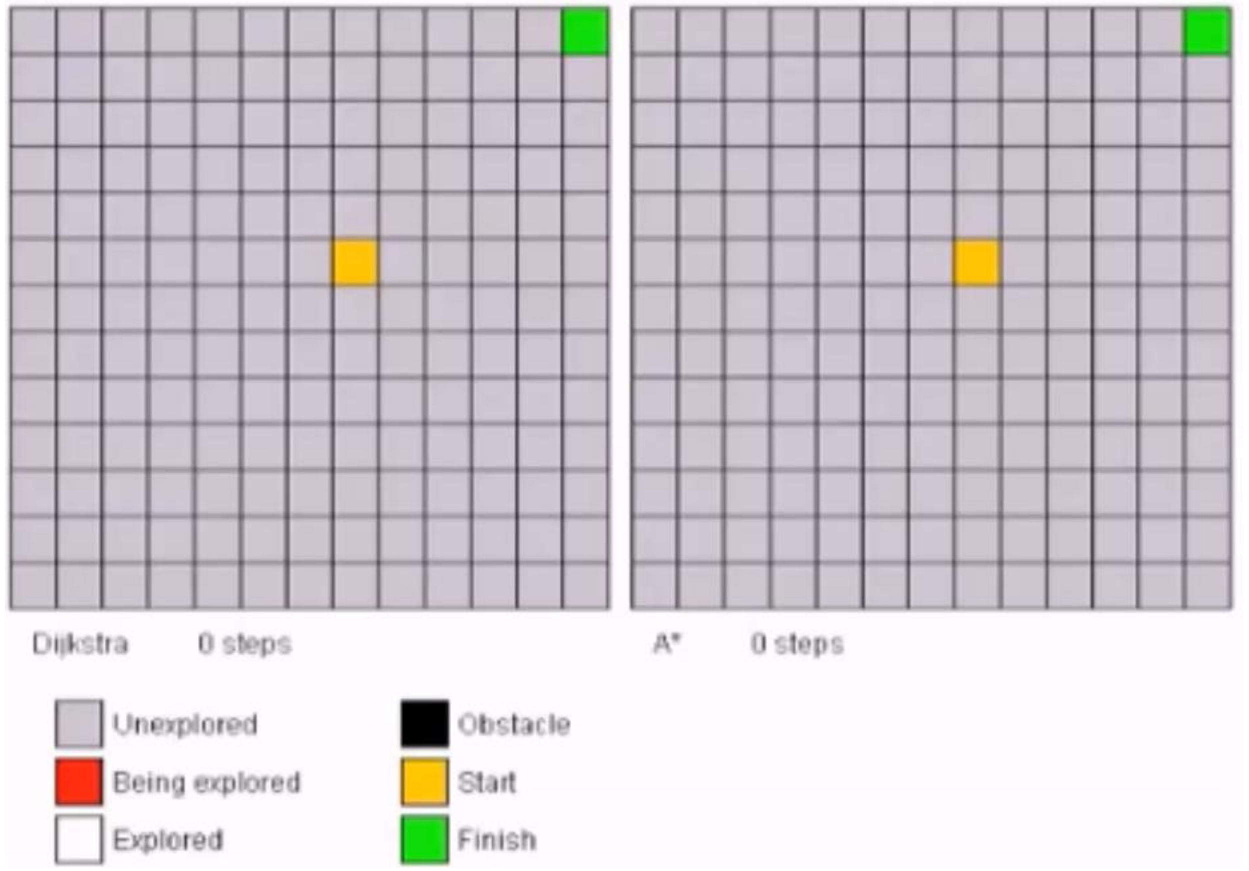
Lower bound on cost of path from source to destination that passes through v

h() is called a heuristic. h() must be **admissible,** i.e. underestimate the cost-to-go from v to destination. h() must also be **monotonic,** i.e. satisfy the triangle inequality.

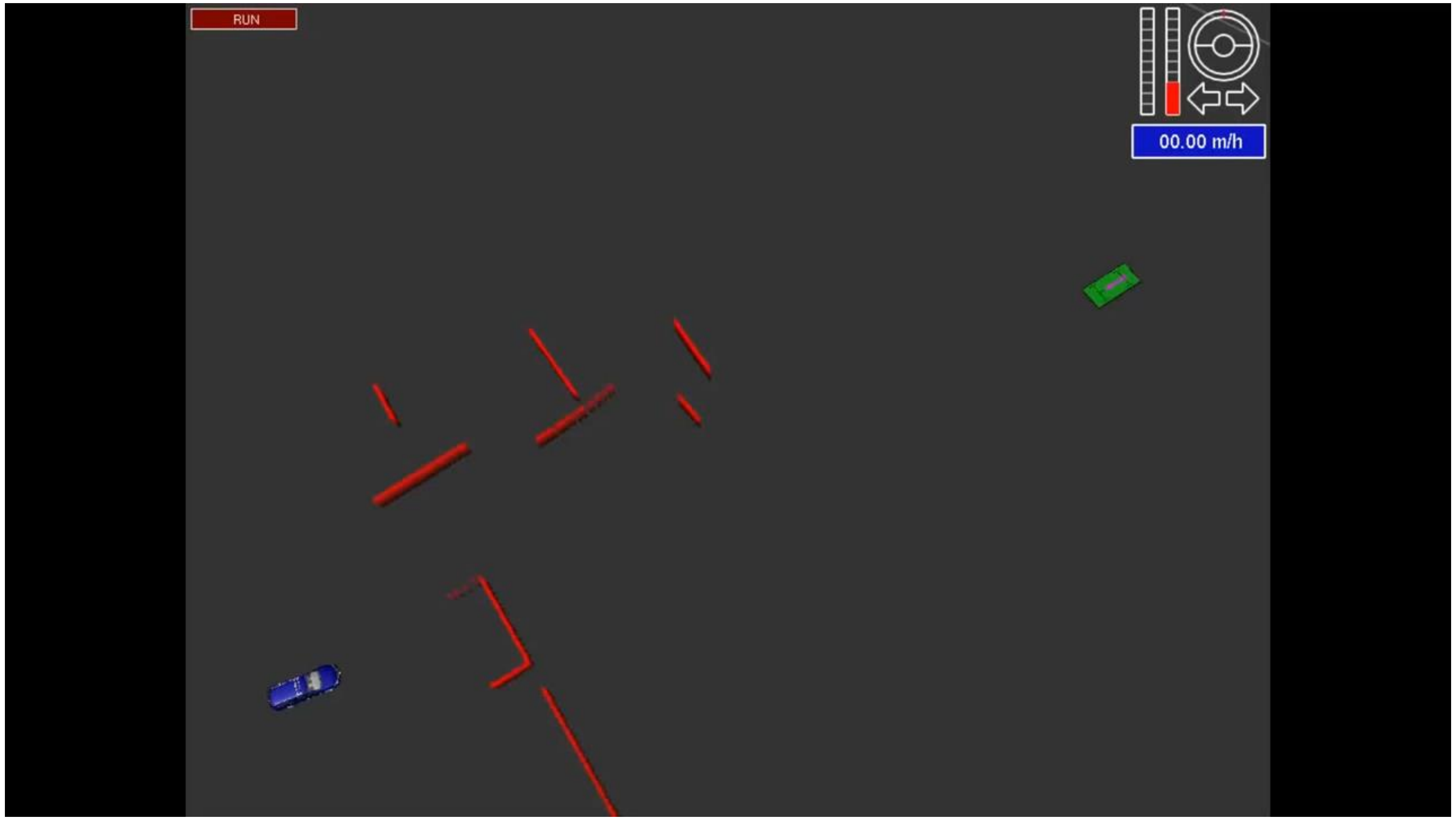
A* Search

- Set $g(v) = \infty$ for all nodes except the source: $g(v_{\text{src}}) = 0$
- Set $f(v) = \infty$ for all nodes except the source: $f(v_{\text{src}}) = h(v_{\text{src}})$
- Add v_{src} to priority queue Q with priority $f(v_{\text{src}})$
- While Q is not empty:
    - Extract the node v with minimum $f(v)$ from the queue Q
    - If found goal then done. Follow the parent pointers from v to get the path.
    - Remove v from the queue Q
    - explored(v) = true
    - For u in neighborhood of v if not explored(u):
        - If u not in Q then
            - Add u in Q with cost-to-come $g(u) = g(v) + d(v, u)$ and priority $f(u) = g(u) + h(u)$
            - Set the parent of u to be v
        - Else if $g(v) + d(v, u) < g(u)$
            - Update the cost-to-come and the priority of u in Q
            - Set the parent of u to be v

Dijkstra vs A*



A* for cars

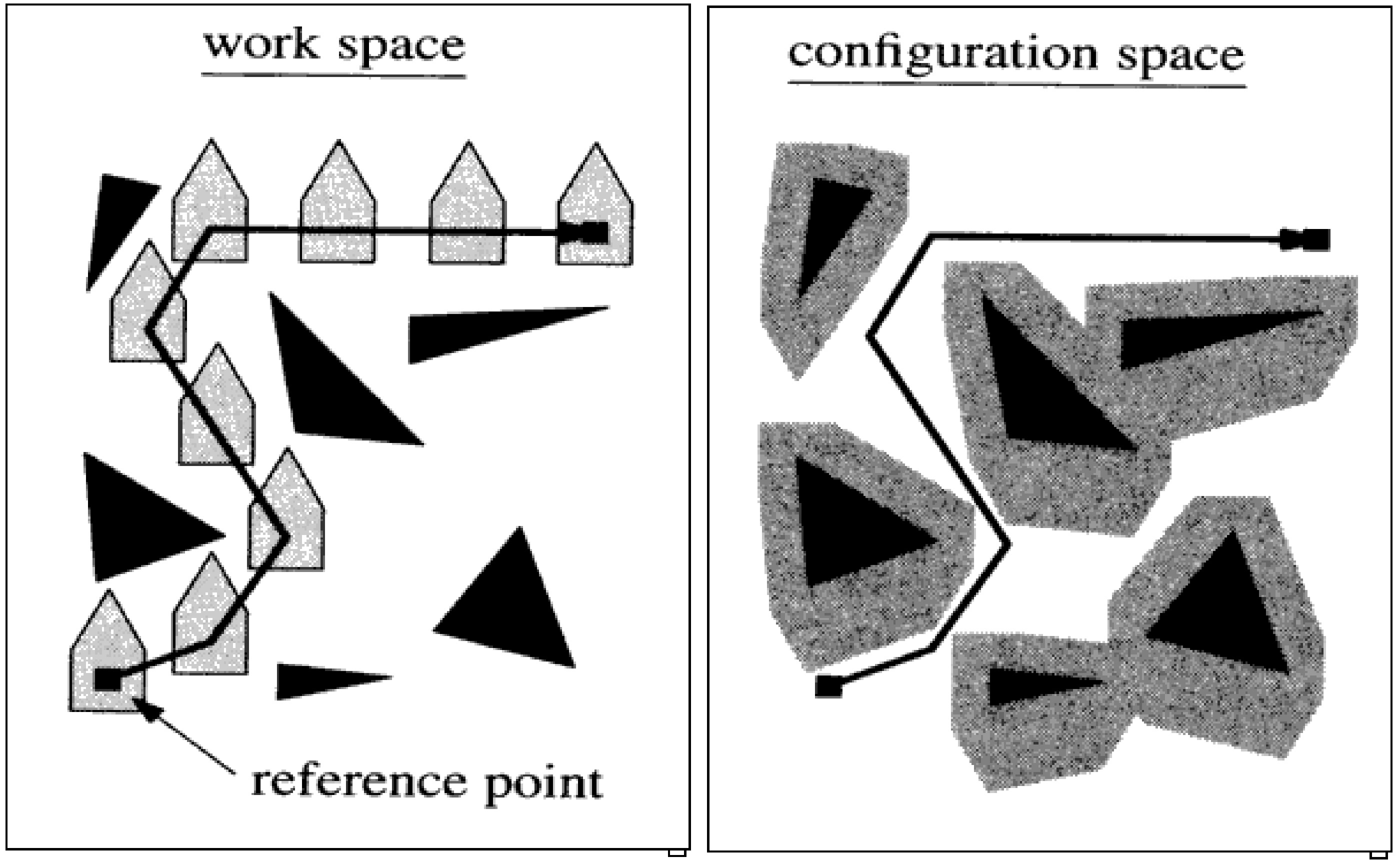


Configuration Space

Idea: dilate obstacles to account for the ways the robot can collide with them.

Why? Instead of planning in the work space and checking whether the robot's body collides with obstacles, plan in configuration space where you can treat the robot as a point because the obstacles are dilated.

This idea is typically not used for robots with high-dimensional states.

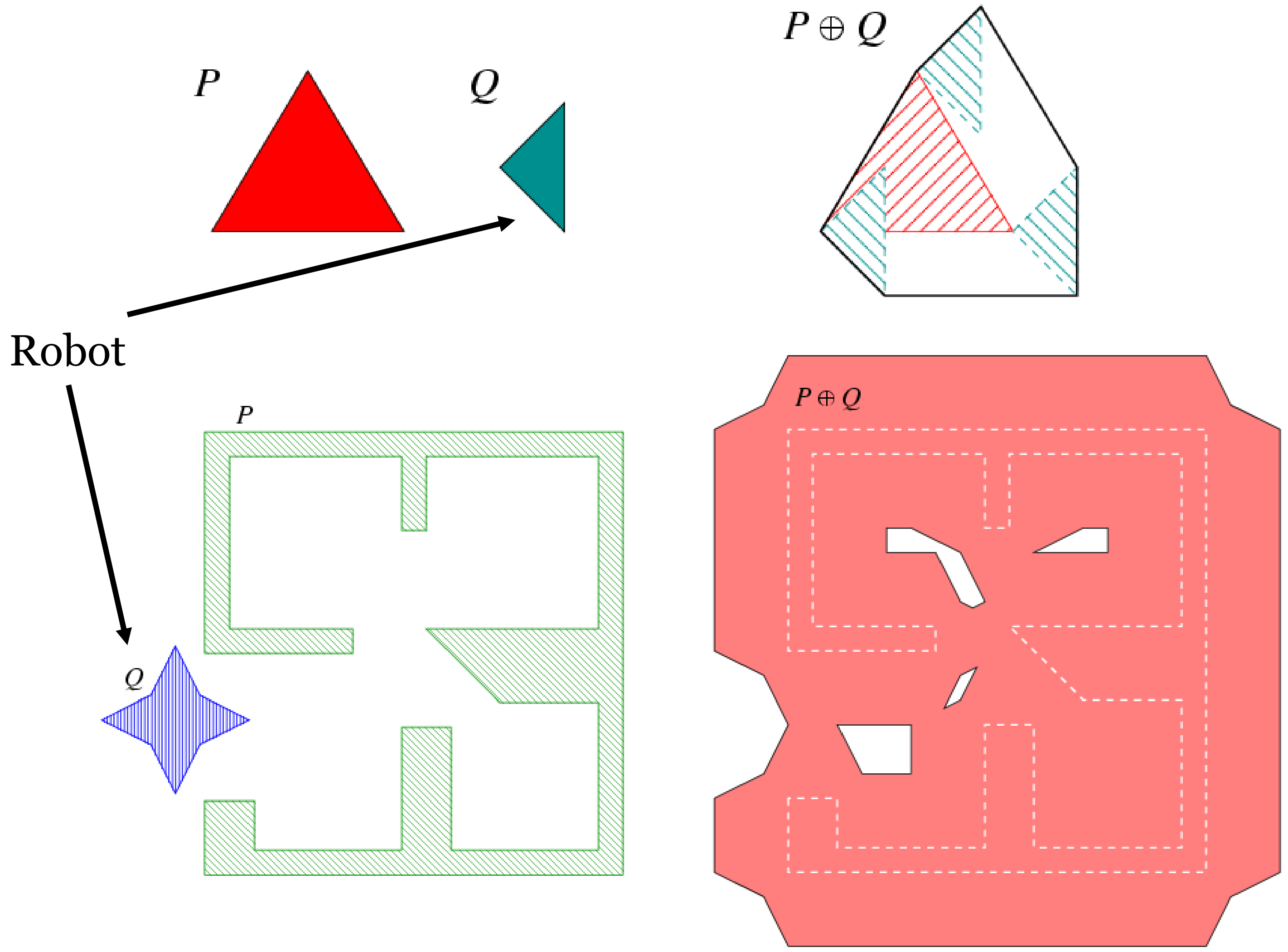


Configuration Space

How do we dilate obstacles?

Minkowski Sum

$$P \oplus Q = \{p + q \mid p \in P,\ q \in Q\}$$


P
Q
P ⊕ Q
Robot
P
Q
P ⊕ Q