Robotics

Spring 2023

Abhishek Gupta

TAs: Yi Li, Srivatsa GS

Courtesy of Maxim Likhachev, CMU, Dieter Fox, UW

Recap: Course Overview

Filtering/Smoothing

Localization

Mapping

SLAM

Search

Motion Planning

TrajOpt

Stability/Certification

MDPs and RL

Imitation Learning

Solving POMDPs

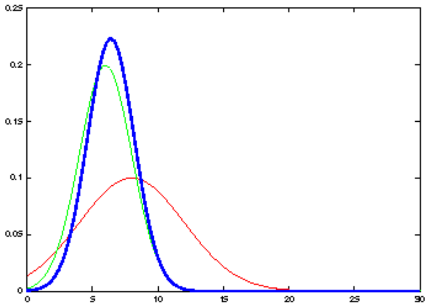
What we have seen so far?

Bayesian Filtering

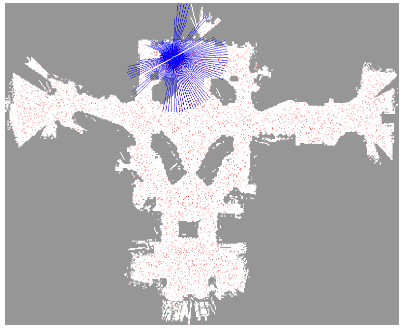
$$Bel(x_t) = P(x_t | u_{0:t-1}, z_{0:t})$$

$$= \eta \, p(z_t|x_t) \int P(x_t|u_{t-1}, x_{t-1}) Bel(x_{t-1}) dx_{t-1}$$

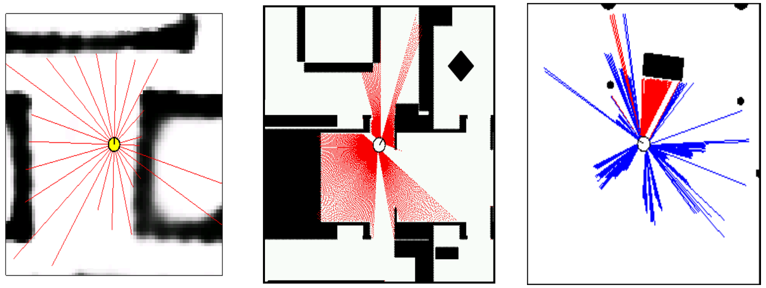
Kalman Filters



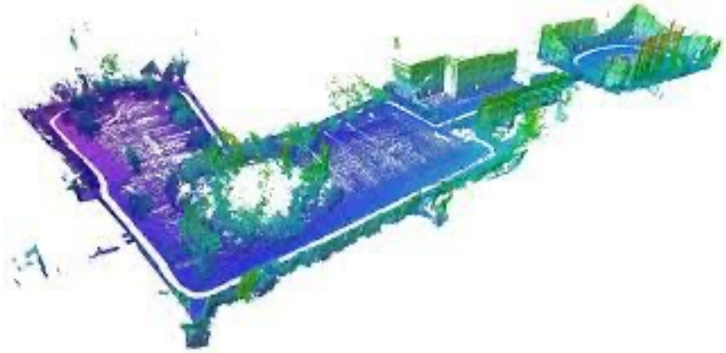
Particle Filters



Localization

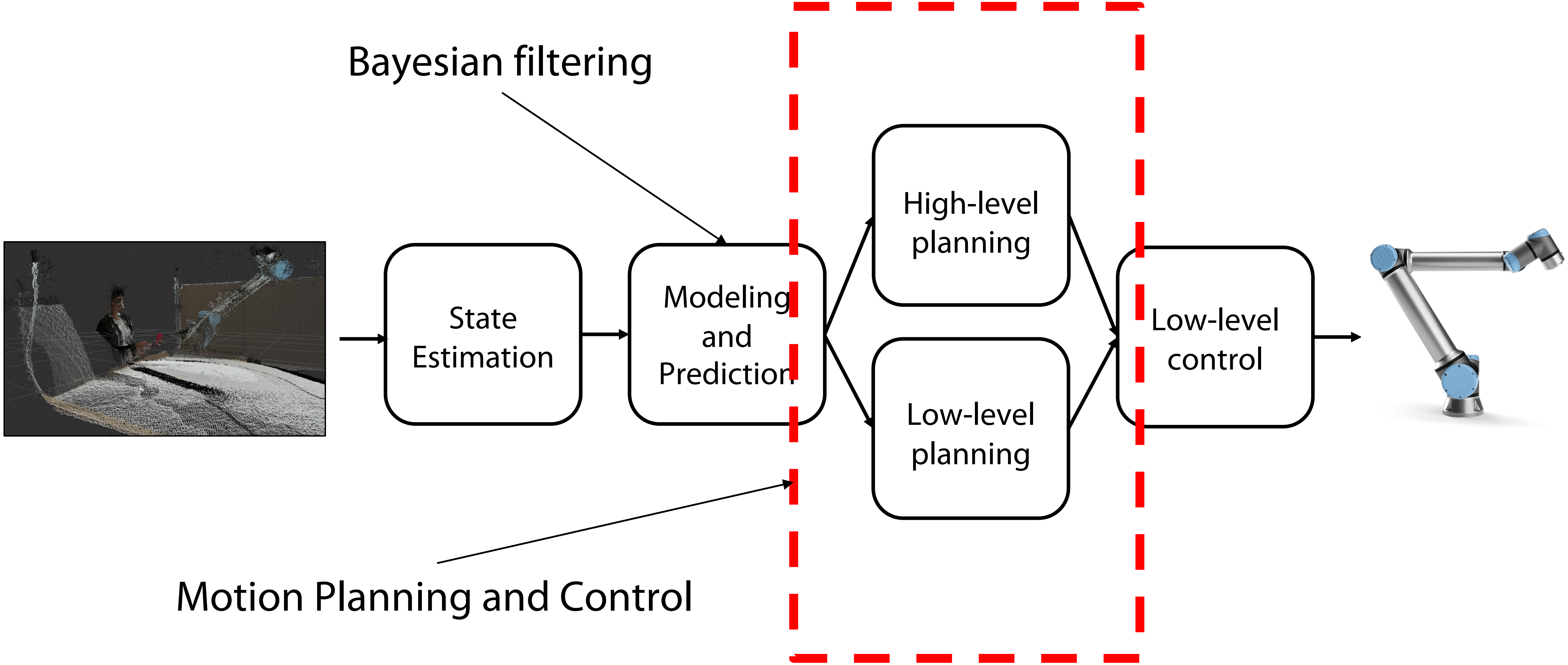


Mapping



SLAM

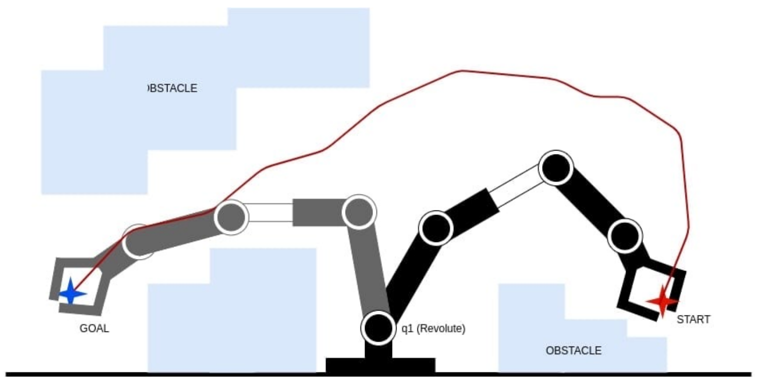
What does full stack robotics involve?



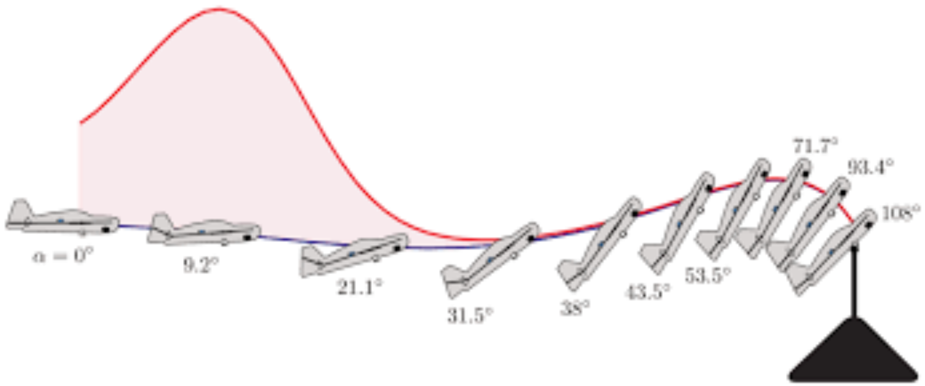
Section 2 of this course

Given an accurate estimate of the state – how do we decide what actions to take?

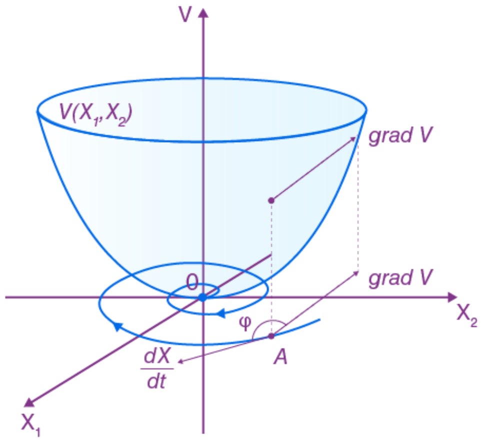
Motion Planning



Trajectory Optimization



Certification



Motion/Path Planning

Task:

- find a feasible (and cost-minimal) path/motion from the current configuration of the robot to its goal configuration (or one of its goal configurations)

Two types of constraints:

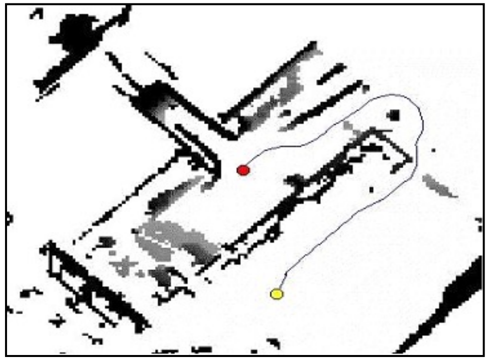
- environmental constraints (e.g., obstacles)
- dynamics/kinematics constraints of the robot

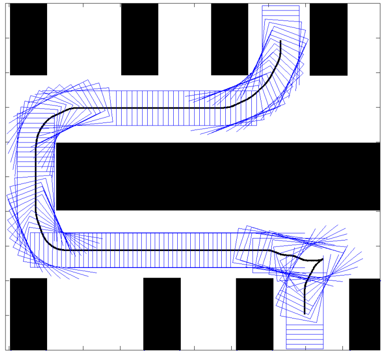
Generated motion/path should (objective):

- be any feasible path
- minimize cost such as distance, time, energy, risk, …

Motion/Path Planning

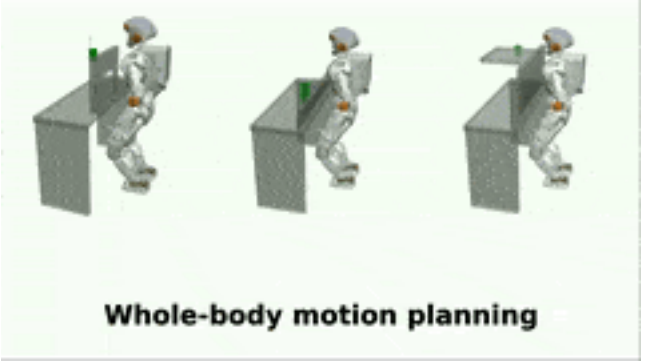
Examples (of what is usually referred to as path planning):

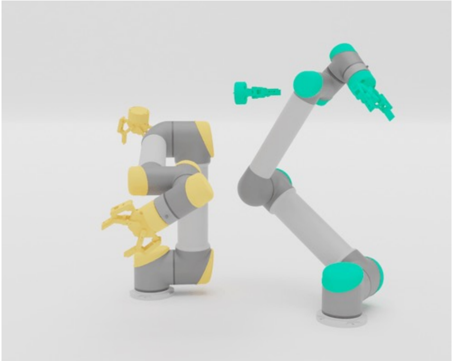


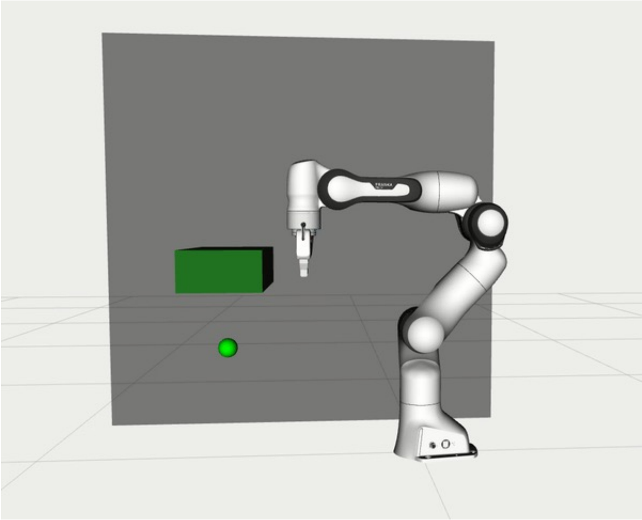


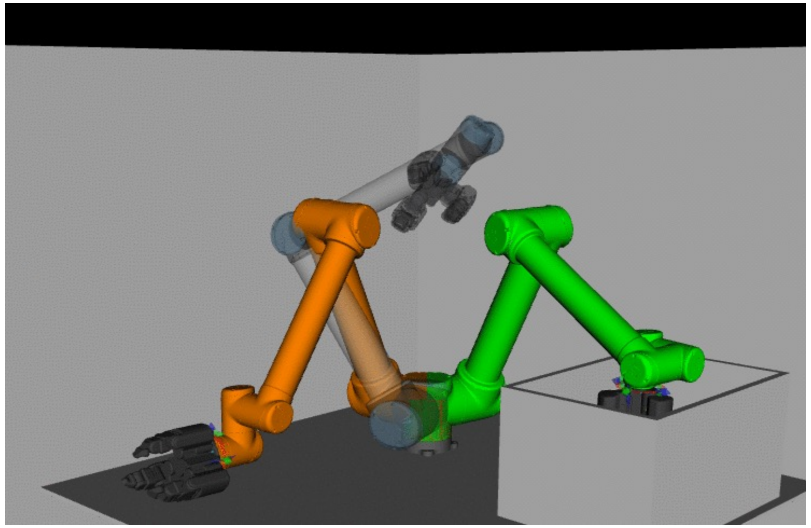
Motion/Path Planning

Examples (of what is usually referred to as motion planning):



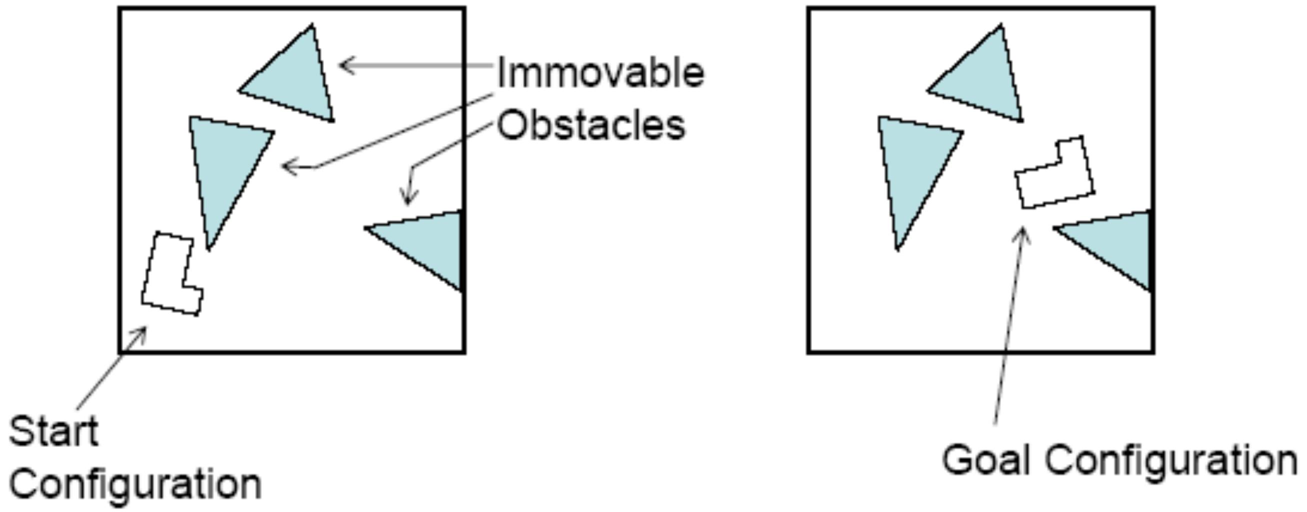






Motion/Path Planning

Examples (of what is usually referred to as motion planning):

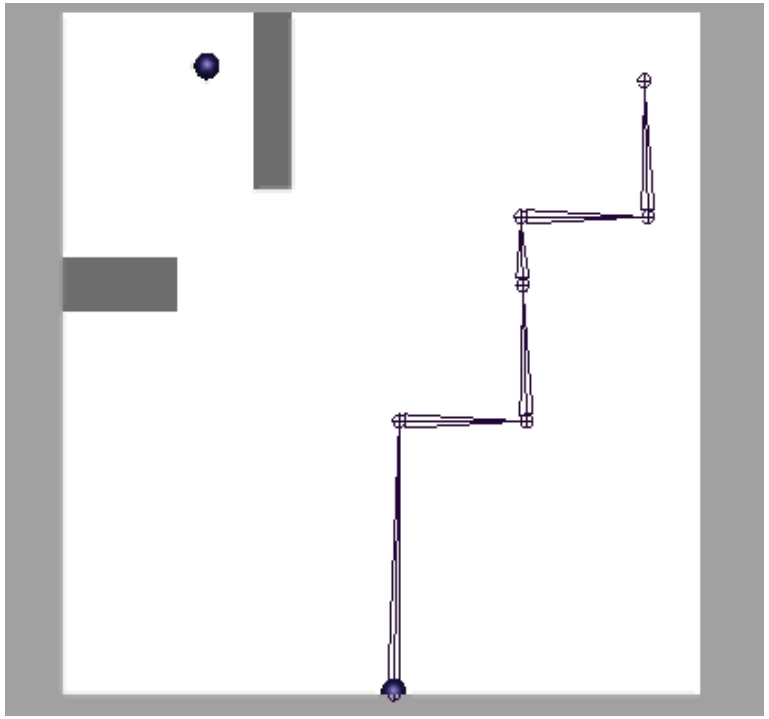


Piano Movers' problem

the example above is borrowed from www.cs.cmu.edu/~awm/tutorials

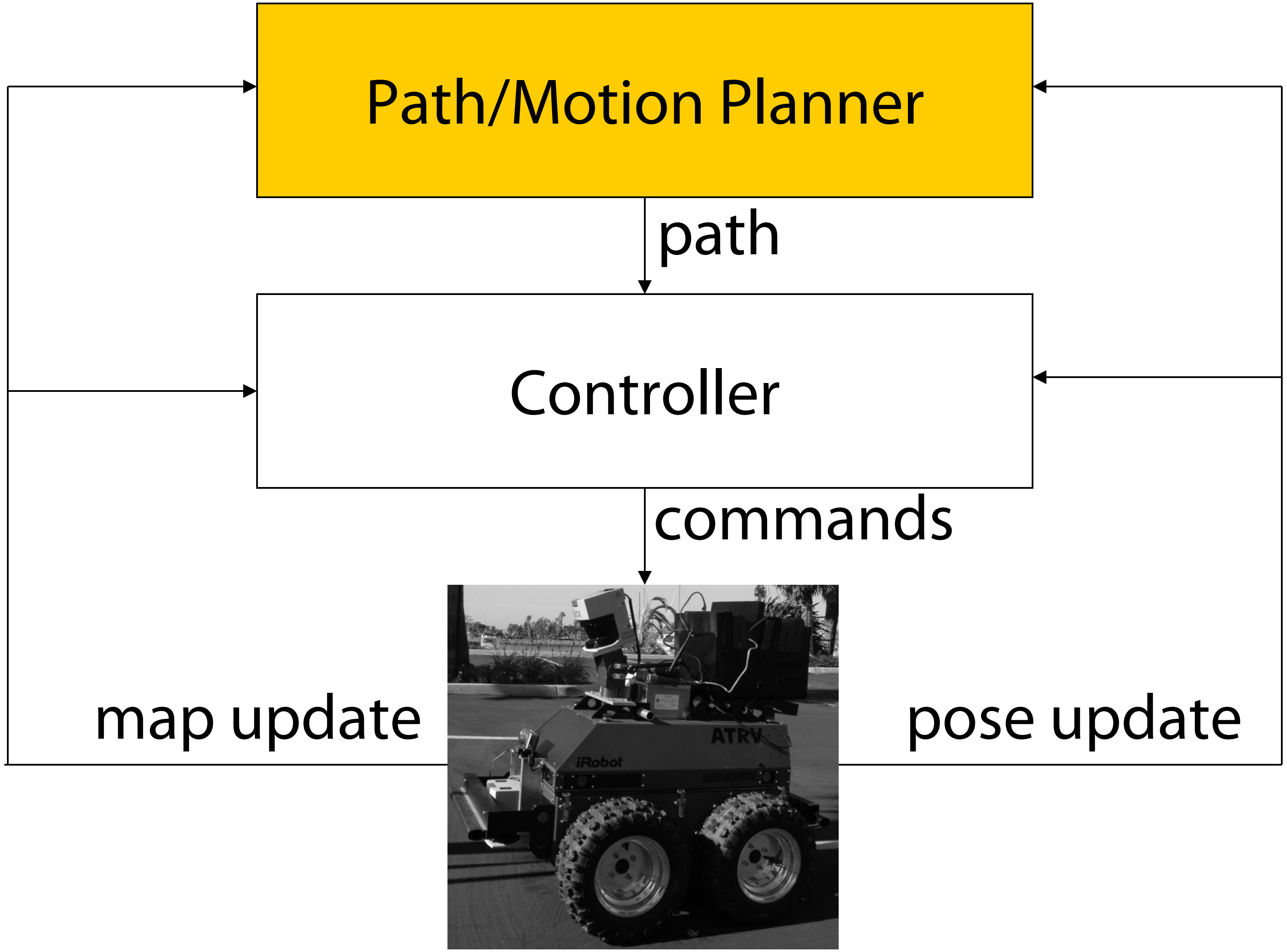
Motion/Path Planning

Examples (of what is usually referred to as motion planning):

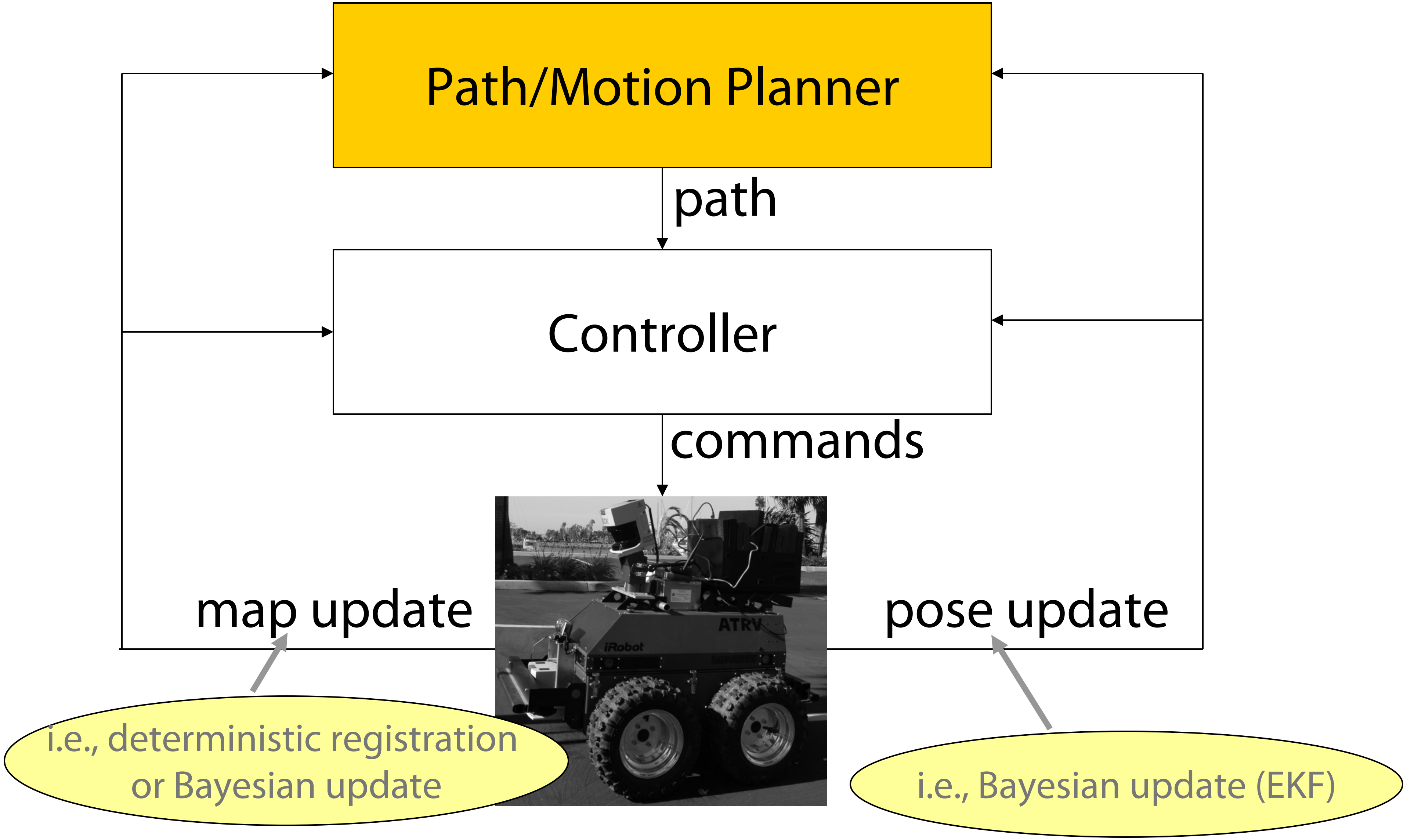


Planned motion for a 6DOF robot arm

Motion/Path Planning

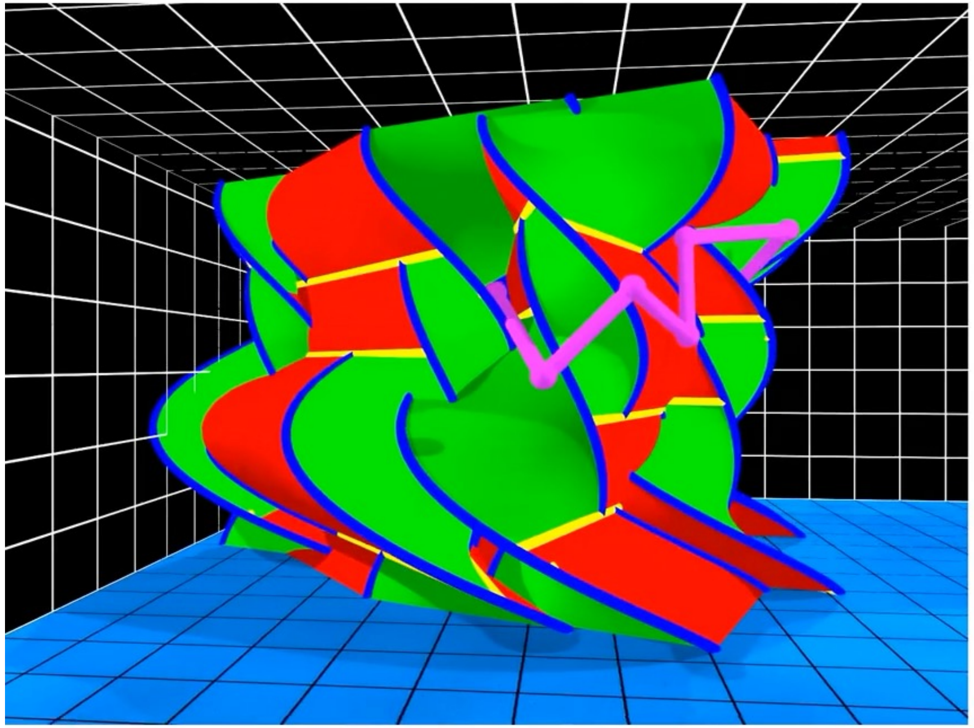


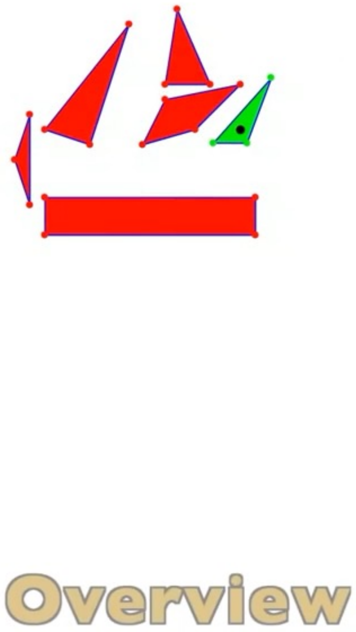
Motion/Path Planning



Why is motion planning non-trivial?

- Searching/Optimization through a complex non-convex space
- Combination of discrete/continuous optimization





Scales poorly with dimensionality of space and number of obstacles – PSPACE complete

Uncertainty and Planning

- Uncertainty can be in:
  - prior environment (i.e., door is open or closed)
  - execution (i.e., robot may slip)
  - sensing environment (i.e., seems like an obstacle but not sure)
  - pose

- Planning approaches:
  - deterministic planning:
    - assume some (i.e., most likely) environment, execution, pose
    - plan a single least-cost trajectory under this assumption
    - re-plan as new information arrives
  - planning under uncertainty:
    - associate probabilities with some elements or everything
    - plan a policy that dictates what to do for each outcome of sensing/action and minimizes expected cost-to-goal
    - re-plan if unaccounted events happen

Uncertainty and Planning

- Uncertainty can be in:
  - prior environment (i.e., door is open or closed)
  - execution (i.e., robot may slip)
  - sensing environment (i.e., seems like an obstacle but not sure)
  - pose

re-plan every time sensory data arrives or robot deviates off its path

- Planning approaches:
  - deterministic planning:
    - assume some (i.e., most likely) environment, execution, pose
    - plan a single least-cost trajectory under this assumption
    - re-plan as new information arrives

re-planning needs to be FAST

  - planning under uncertainty:
    - associate probabilities with some elements or everything
    -plan a policy that dictates what to do for each outcome of sensing/action and minimizes expected cost-to-goal
    - re-plan if unaccounted events happen

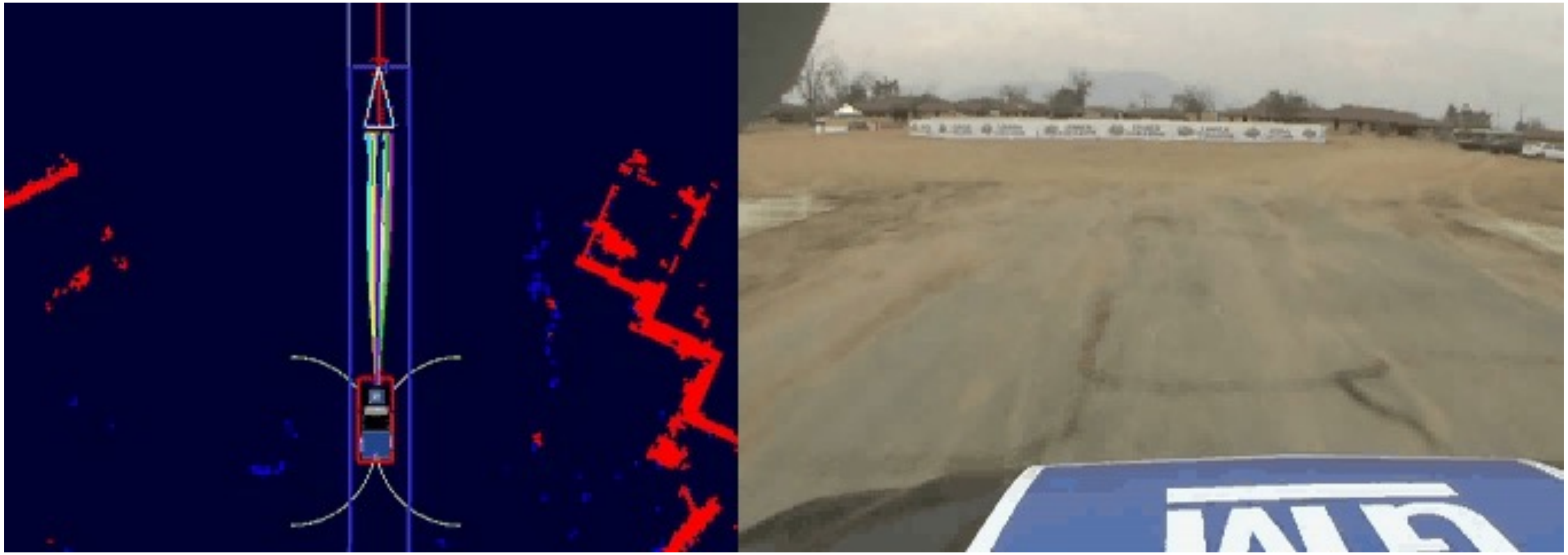
Uncertainty and Planning

- Uncertainty can be in:
  - prior environment (i.e., door is open or closed)
  - execution (i.e., robot may slip)
  - sensing environment (i.e., seems like an obstacle but not sure)
  - pose

- Planning approaches:
  - deterministic planning:
    - assume some (i.e., most likely) environment, execution, pose
    - plan a single least-cost trajectory under this assumption
    - re-plan as new information arrives
  - planning under uncertainty:
    - associate probabilities with some elements or everything
    - plan a policy that dictates what to do for each outcome of sensing/action and minimizes expected cost-to-goal
    - re-plan if unaccounted events happen

computationally MUCH harder

Example



Urban Challenge Race, CMU team, planning with Anytime D*

Lecture Outline

Casting motion planning as a search problem

↓

Motion Planning via A* search

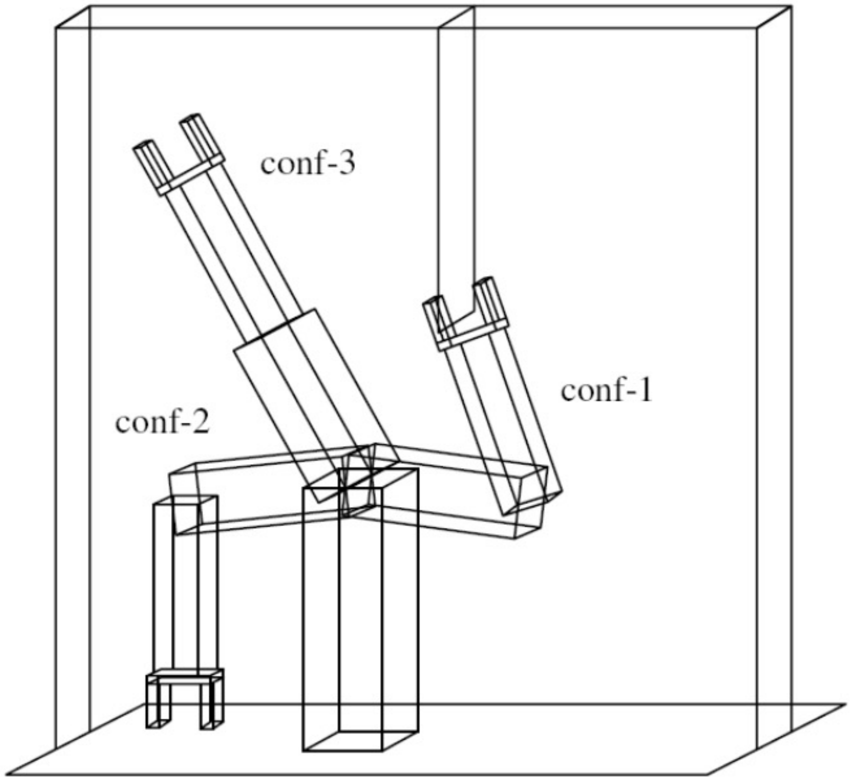
↓

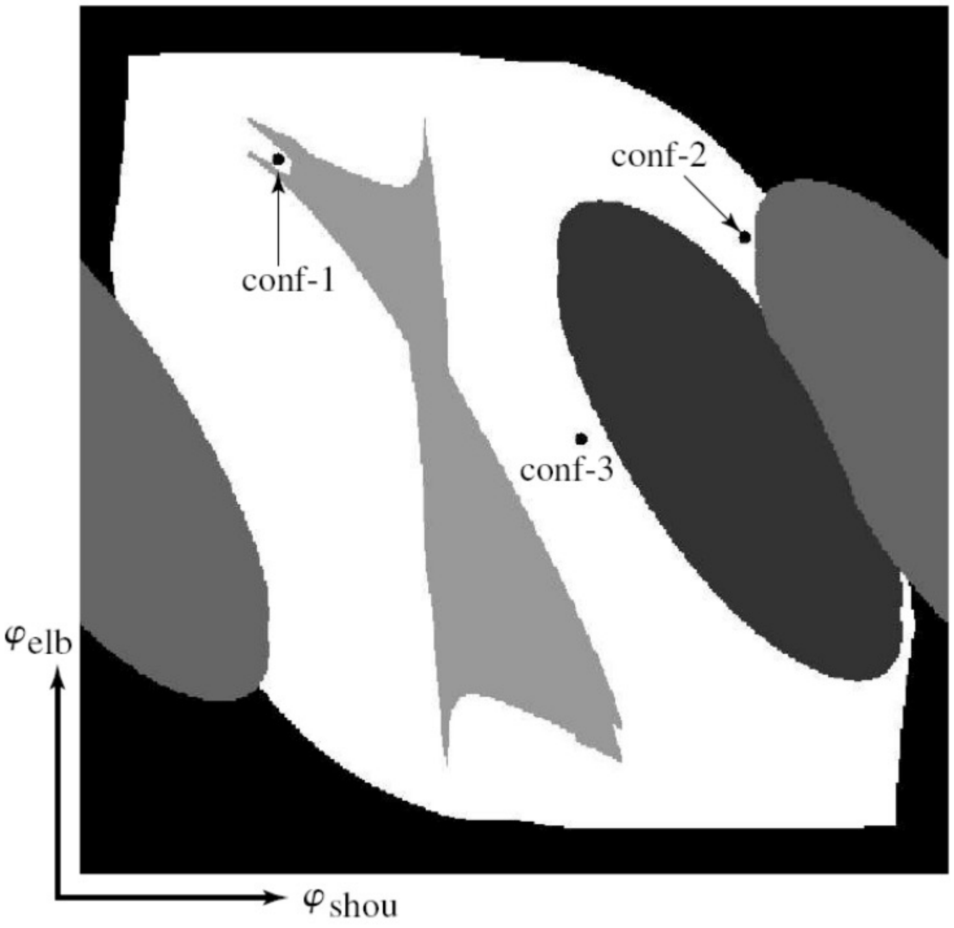
Incremental Search for Replanning

Defining the Motion Planning Problem

- **Problem:**
  - Given start state xS, goal state xG
  - Asked for: a sequence of control inputs that leads from start to goal
- **Why tricky?**
  - Need to avoid obstacles
  - For systems with underactuated dynamics: can't simply move along any coordinate at will
    - E.g., car, helicopter, airplane, but also robot manipulator hitting joint limits

Configuration Space



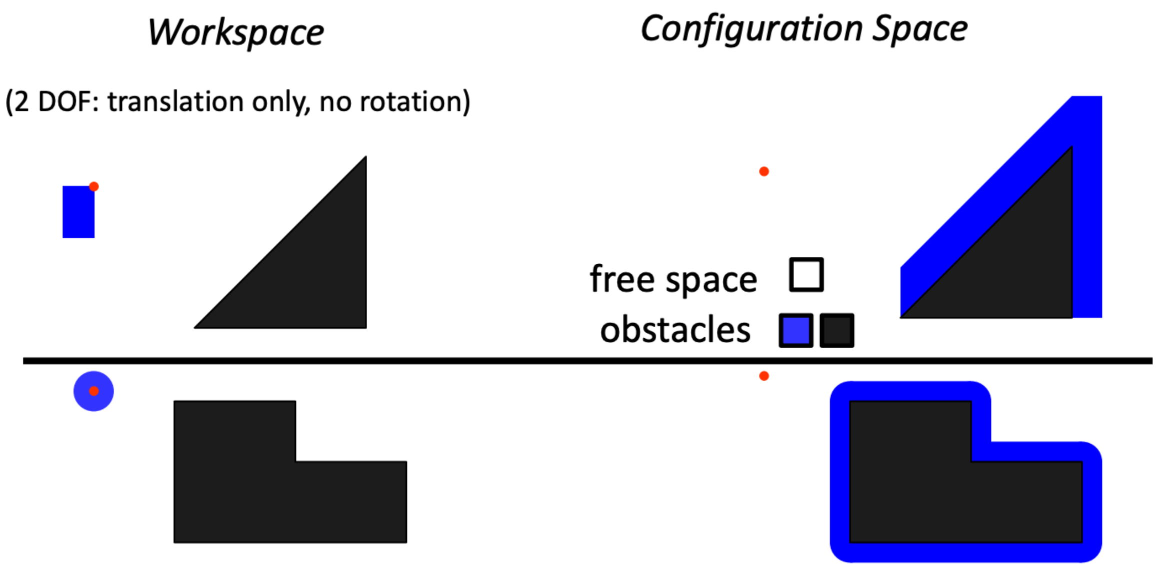


Configuration Space

Configuration space: space of joint configurations of the robot

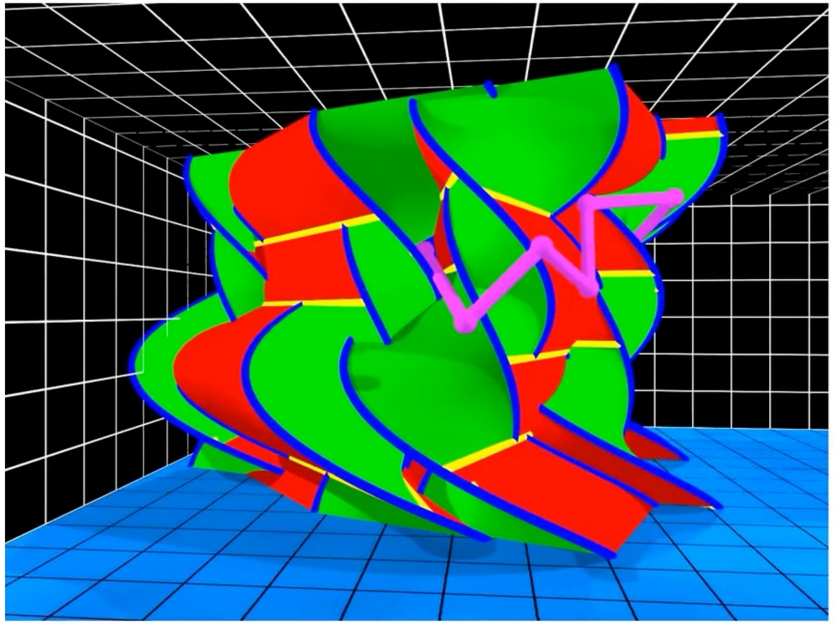
Obstacles/constraints do not live in the joint space of the robot but in the world space
→ non-convex when projected into configuration space

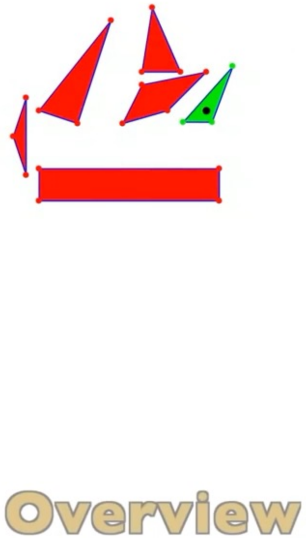
Finding collision free paths is a non-trivial search problem.



Motion Planning in Configuration Space

Cannot directly use optimization techniques like gradient descent,
must solve a non-convex optimization problem.

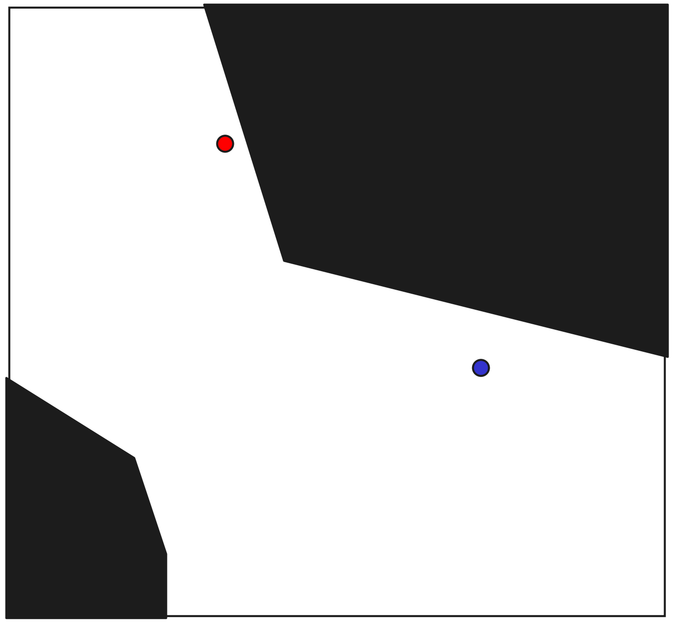




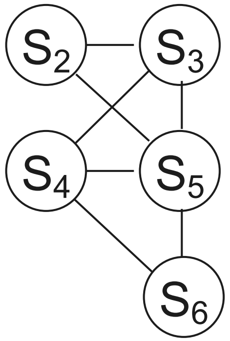
Idea 1: Modeling as discrete search

Idea 2: Sequential convexification of non-convex problems

Planning as Search



Convert into a search problem



planning map

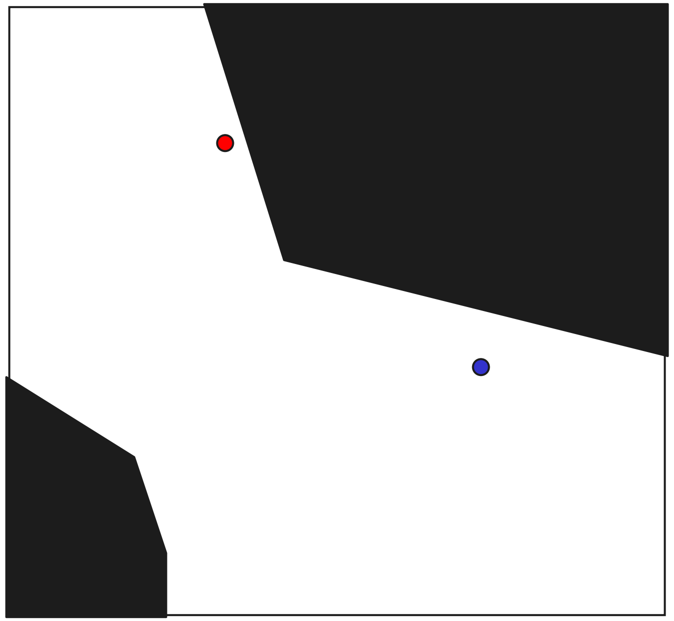
search the graph for a least-cost path from s_{start} to s_{goal}

Can use efficient techniques for <u>**discrete**</u> graph search

Deterministic Search

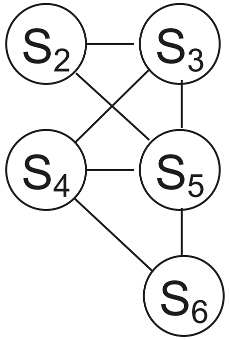
Sampling Based Search

Recasting Planning as Search



Convert into a search problem

How?



planning map

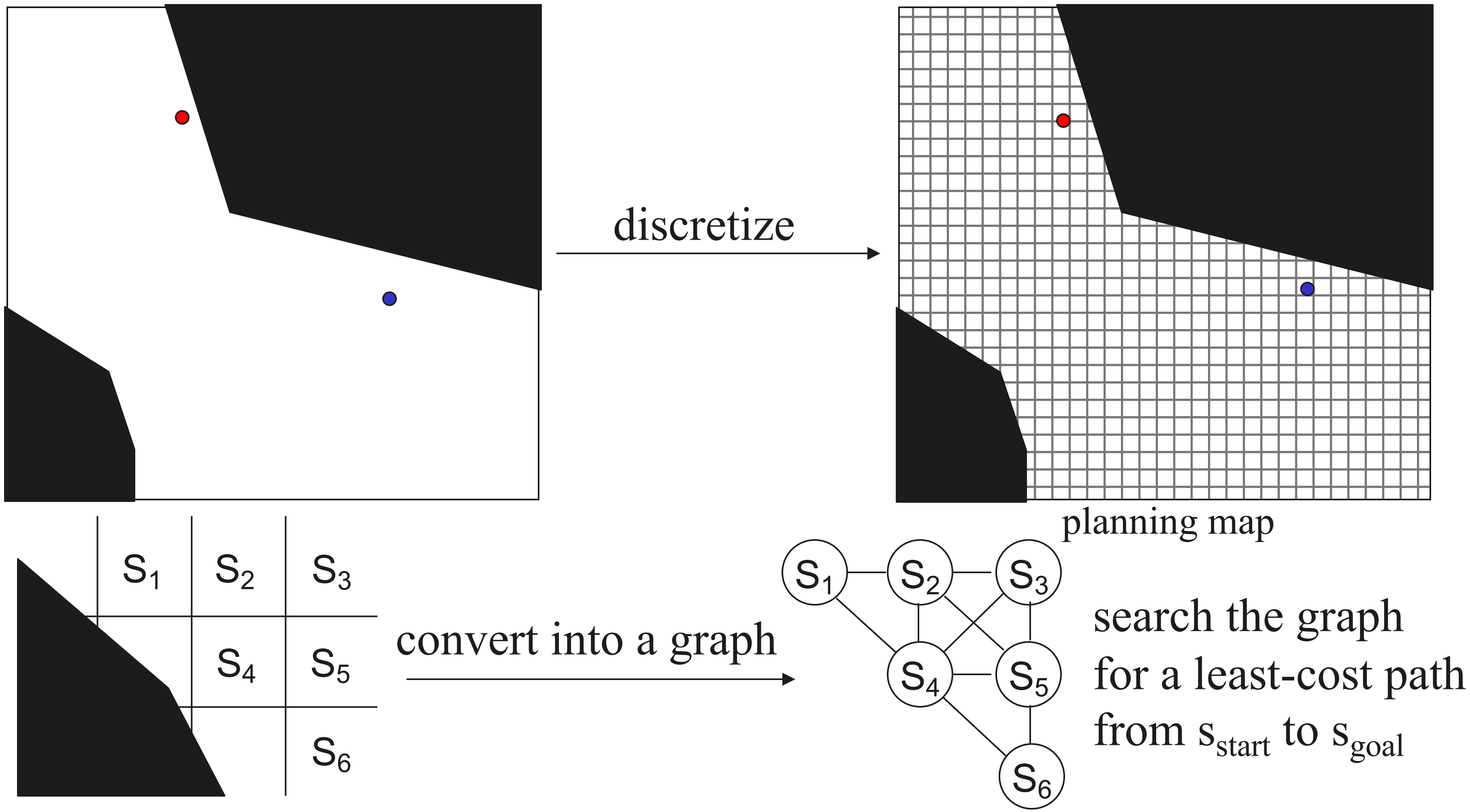
search the graph for a least-cost path from s_{start} to s_{goal}

Can use efficient techniques for **discrete** graph search

Which ones?

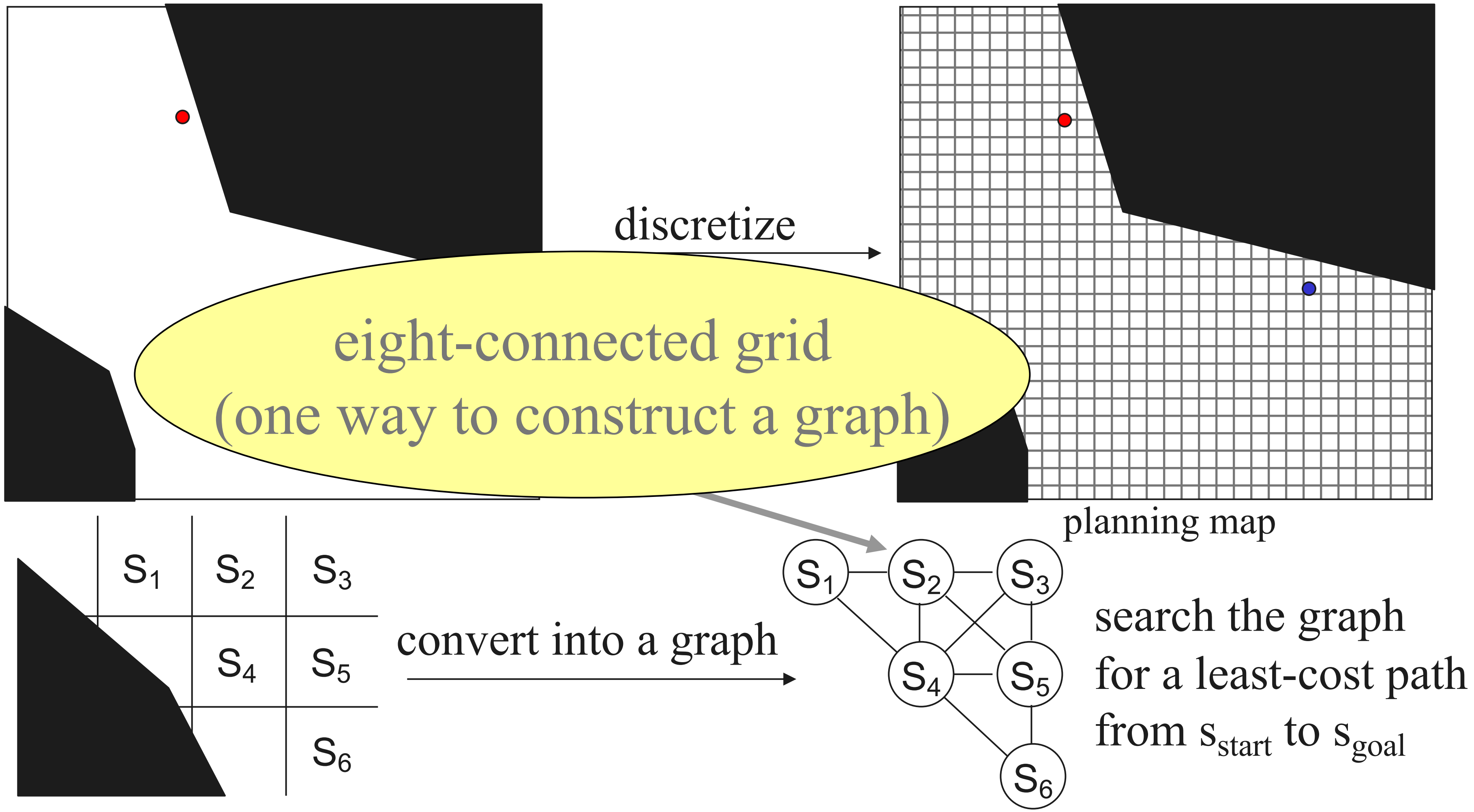
Motion Planning via Cell Decomposition

- Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path



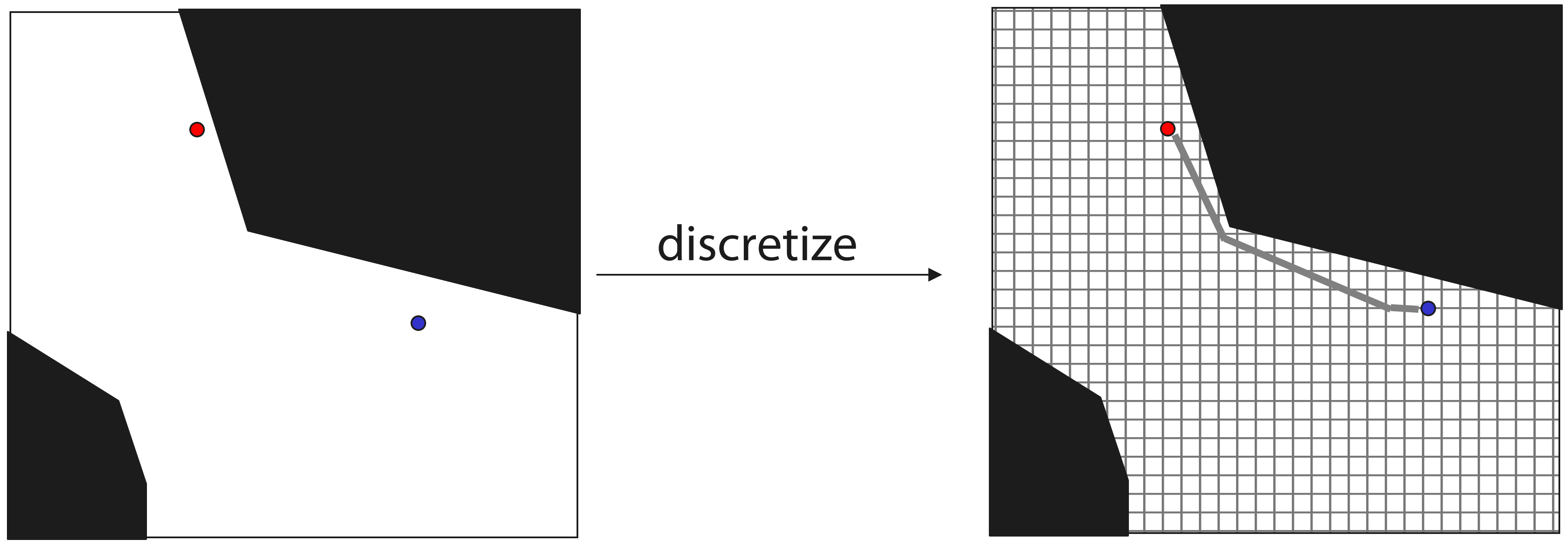
Planning via Cell Decomposition

- Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path



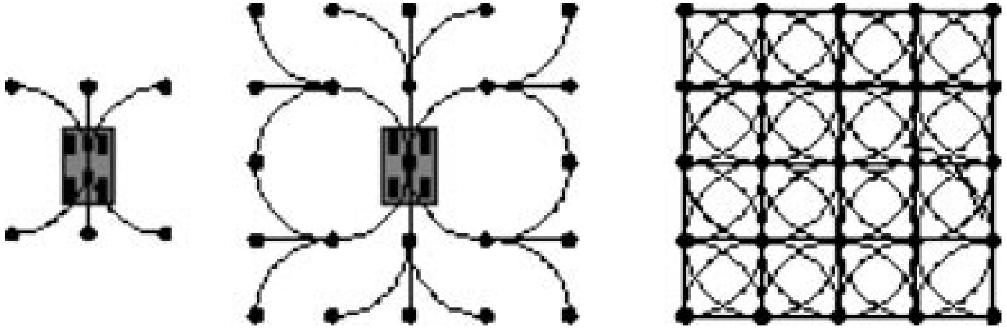
Planning via Cell Decomposition

- Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path
    - VERY popular due to its simplicity and representation of arbitrary obstacles
    - Problem: transitions difficult to execute on non-holonomic robots



How can we connect states for non-holonomic robots?

Requires solving a 2 point boundary value problem on kinematics



s_0 and s_1 are connected if there exists a u such that $|f(s_0, u) - s_1| < \varepsilon$

Differentially Constrained Mobile Robot Motion Planning in State Lattices

Mihail Pivtoraiko, Ross A. Knepper, and Alonzo Kelly
Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
e-mail: mihail@cs.cmu.edu, rak@ri.cmu.edu, alonzo@ri.cmu.edu

Received 6 August 2008; accepted 4 January 2009

On the Reachability of Quantized Control Systems

Antonio Bicchi, Alessia Marigo, Benedetto Piccoli

Can be extended to dynamics systems too!

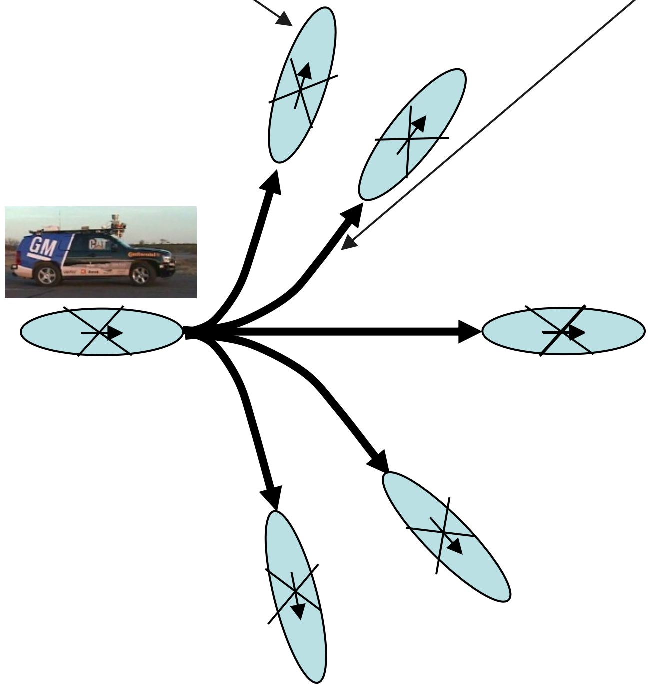
Planning via Cell Decomposition

- Graph construction:
  - lattice graph

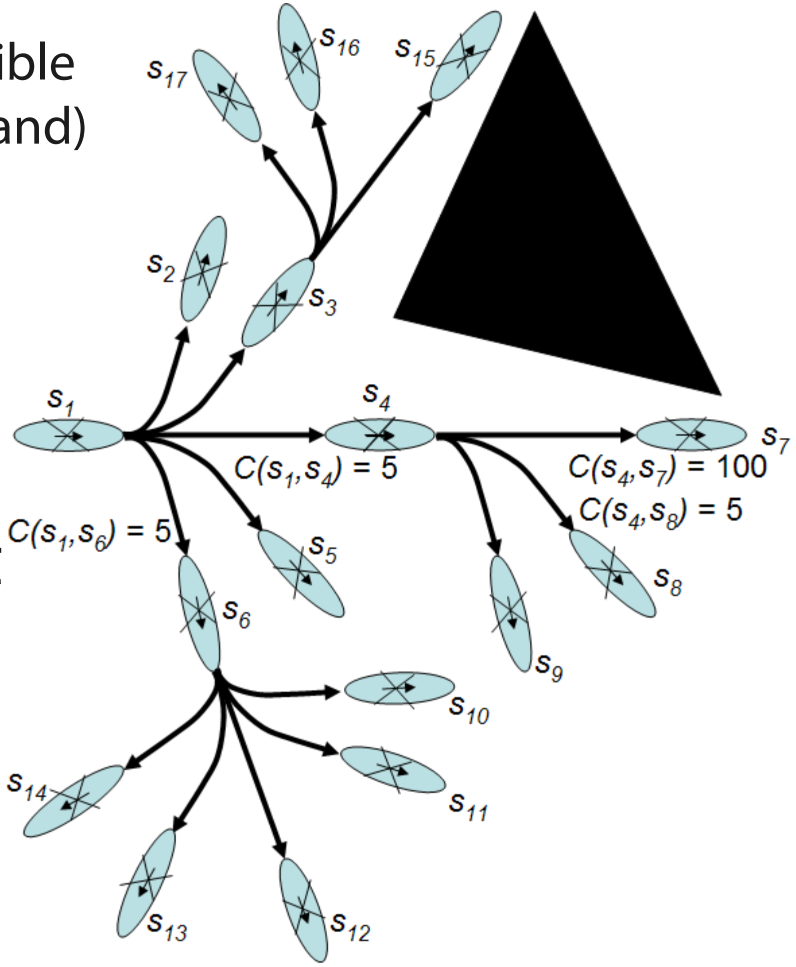
outcome state is the center of the corresponding cell

each transition is feasible (constructed beforehand)

action template



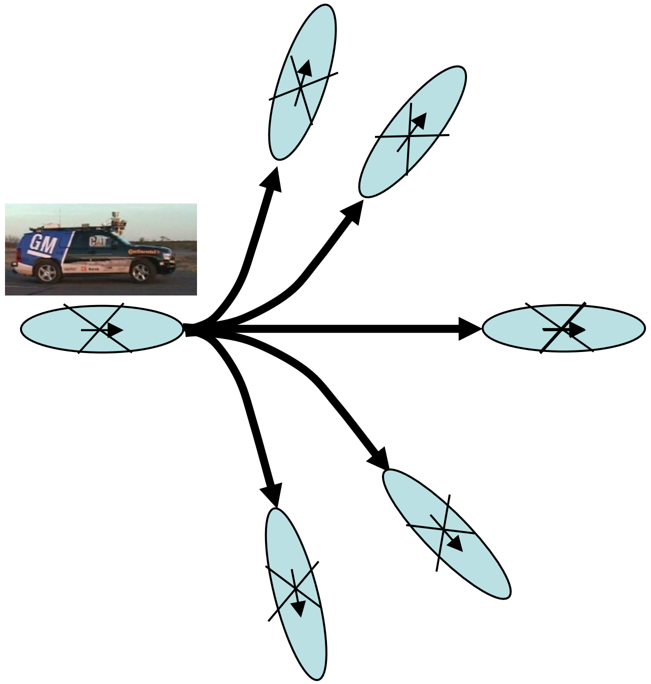
replicate it online



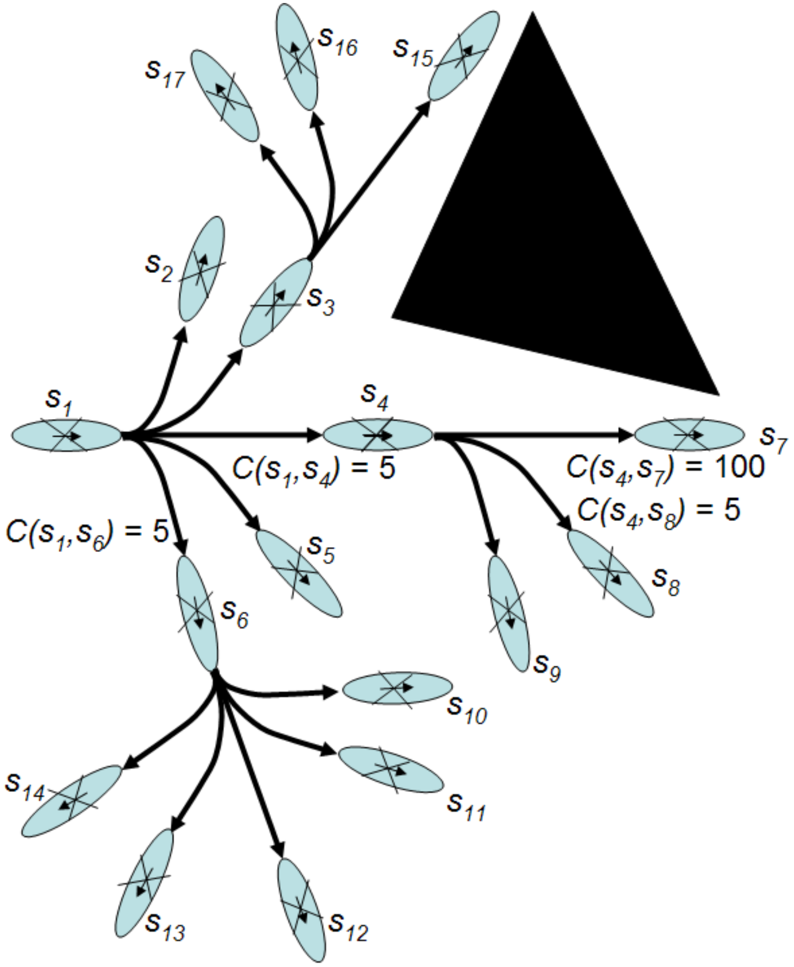
Planning via Cell Decomposition

- Graph construction:
  - lattice graph
  - pros: sparse graph, feasible paths
  - cons: possible incompleteness

action template



replicate it online



Lecture Outline

Casting motion planning as a search problem

↓

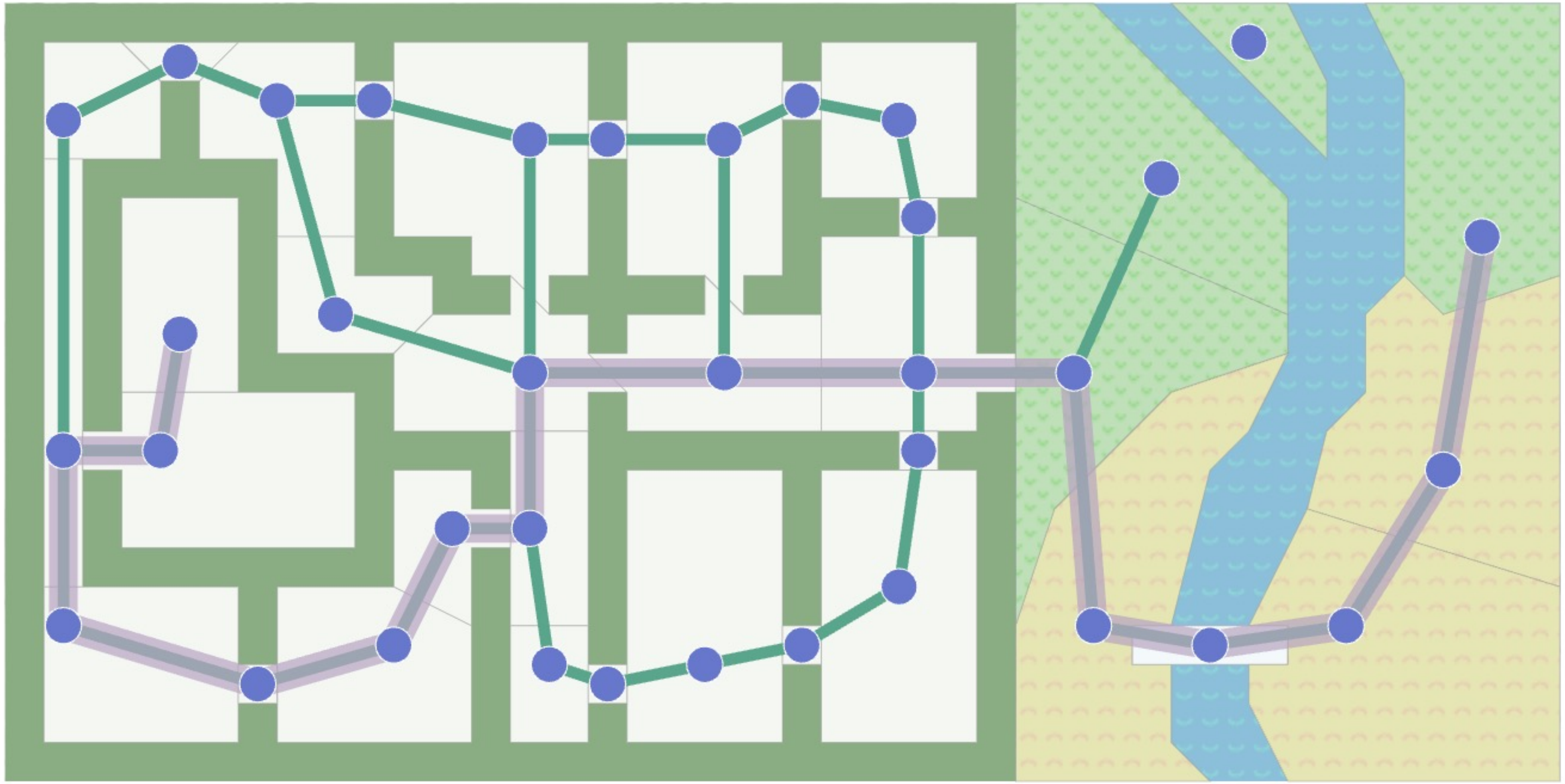
Motion Planning via A* search

↓

Incremental Search for Replanning

Techniques for Search

Start

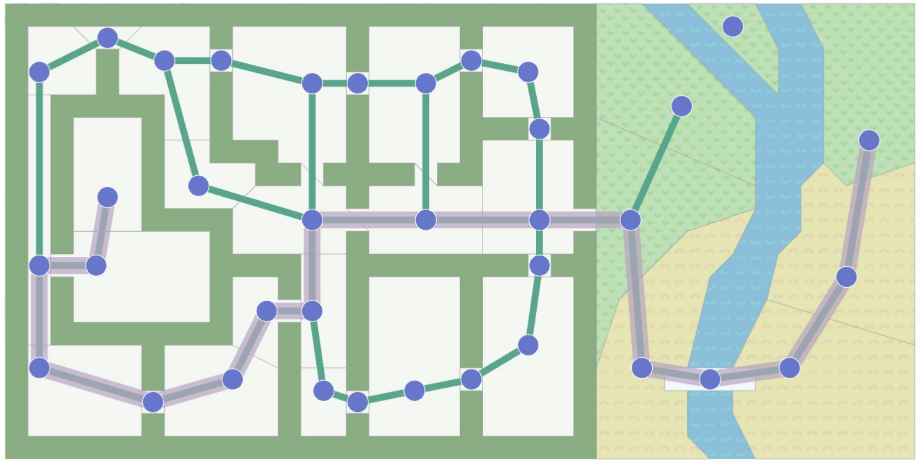


Goal

Goal is to avoid obstacles and reach a particular goal with:

1. As few node expansions as possible
2. Lowest cost path

Techniques for Search



Breadth First Search

Uniform Cost Search

A* Search

Search Attempt 1: Breadth First Search

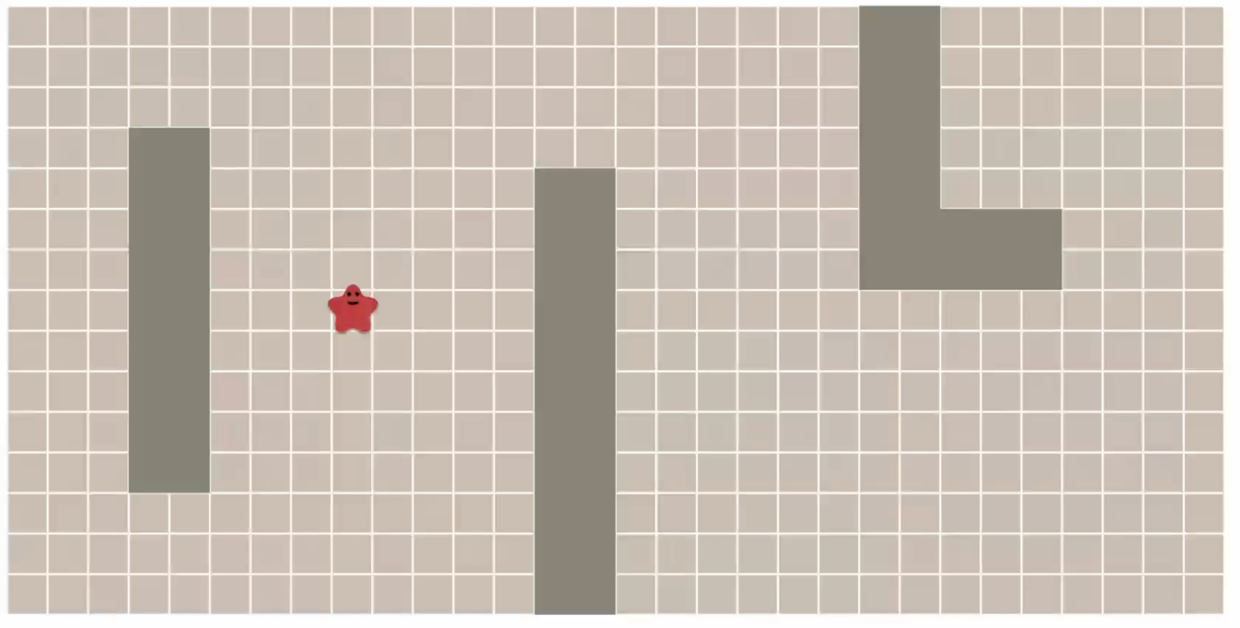
Expand the search uniformly in all directions from start

```
frontier = Queue()
frontier.put(start )
came_from = dict()
came_from[start] = None

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        if next not in came_from:
            frontier.put(next)
            came_from[next] = current
```

Breadth First Search

Search Attempt 1: Breadth First Search

Expand the search uniformly in all directions from start



Pro: Guaranteed to find shortest paths

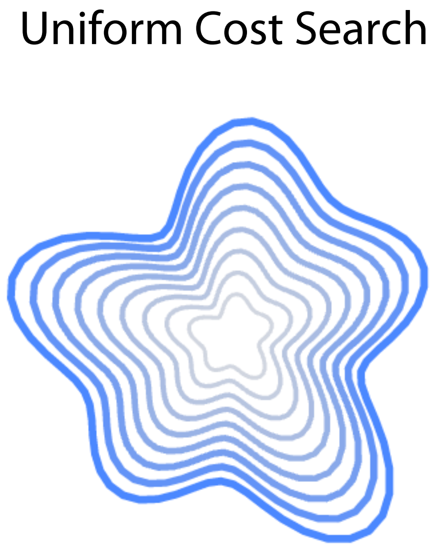
Cons:

1. Doesn't take costs into account
2. May expand way more nodes than necessary

Search Attempt 2: Uniform Cost Search

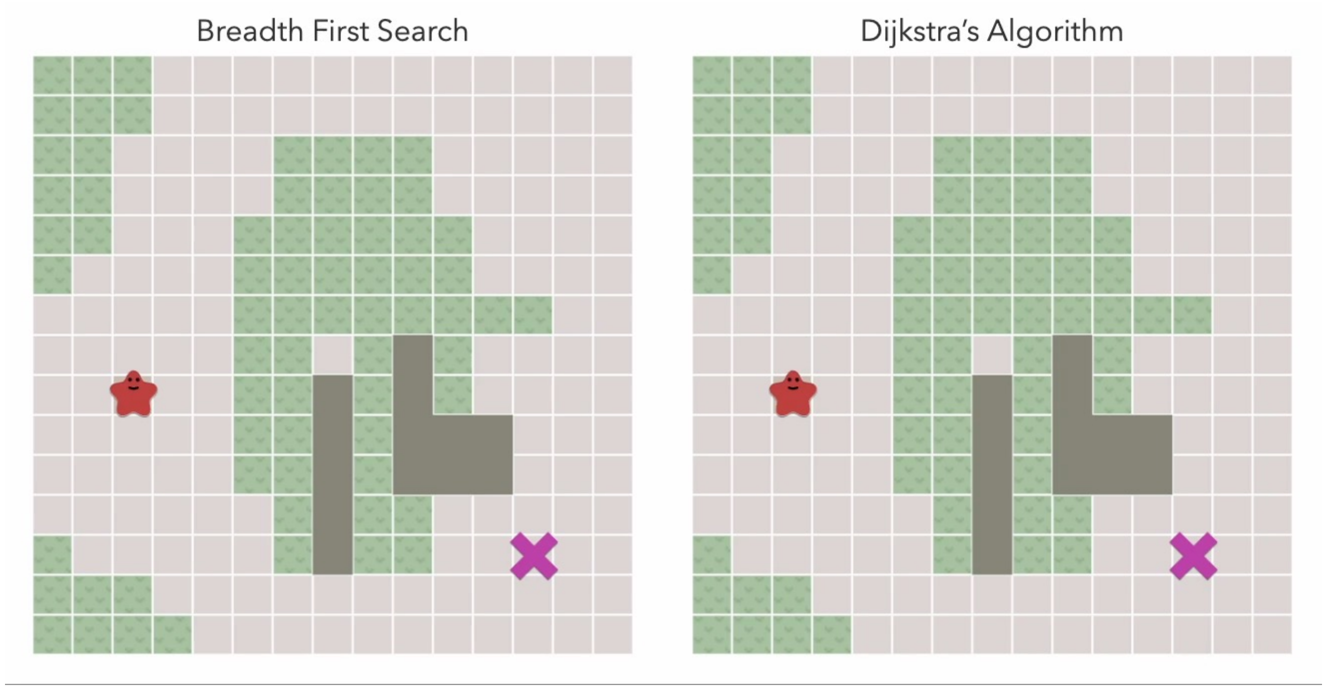
Expand the search according to lowest cost from the start

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```



Search Attempt 2: Uniform Cost Search

Expand the search according to lowest cost from the start



Pro: Guaranteed to find lowest cost paths

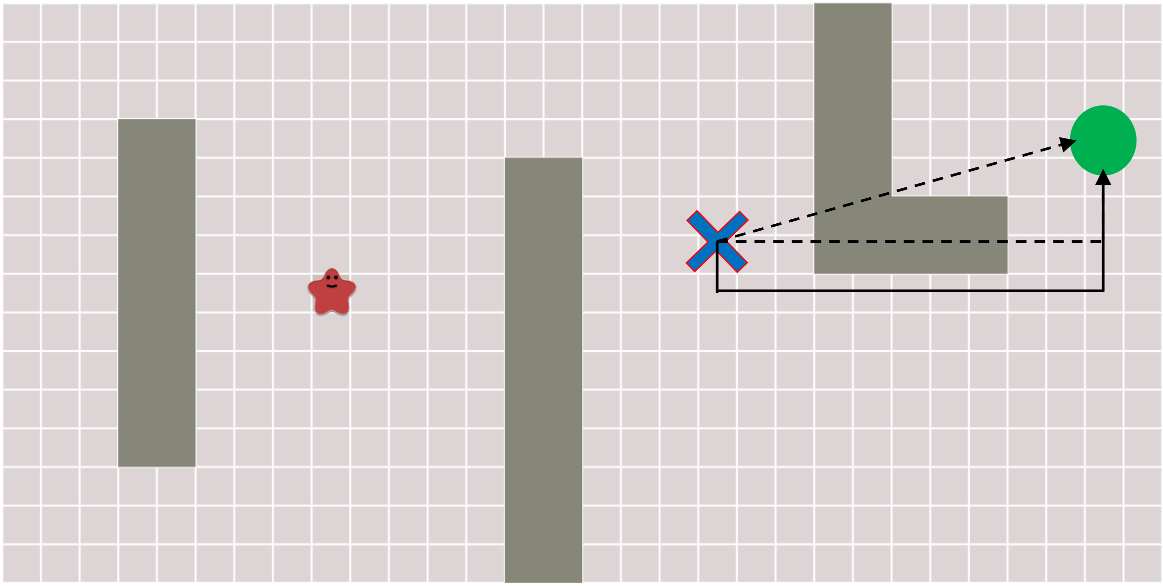
Cons:

1. May expand way more nodes than necessary

Informed Search

What if we knew some (approximate) information about how far a node is from the goal?

→ Heuristics



Example: for shortest path goal reaching around obstacles, reasonable heuristics are:

1. Euclidean distance

2. Manhattan distance

Incorporate domain knowledge while always **underestimating** cost

Admissible heuristic

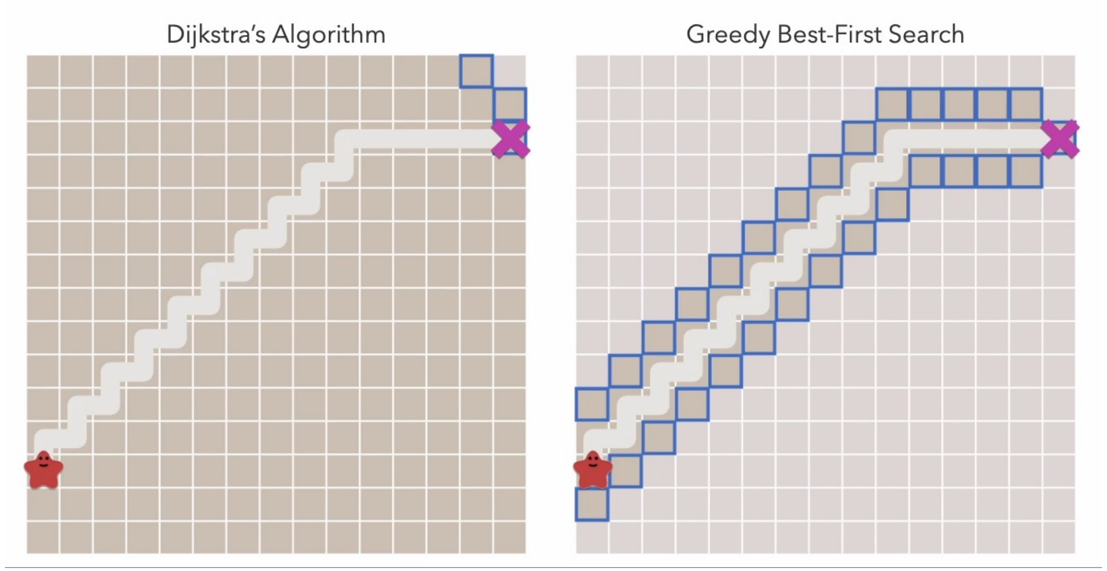
Informed Search Attempt 1: Best-First Search

Choose the next node to expand as the one that has the lowest heuristic – “greedy best first”

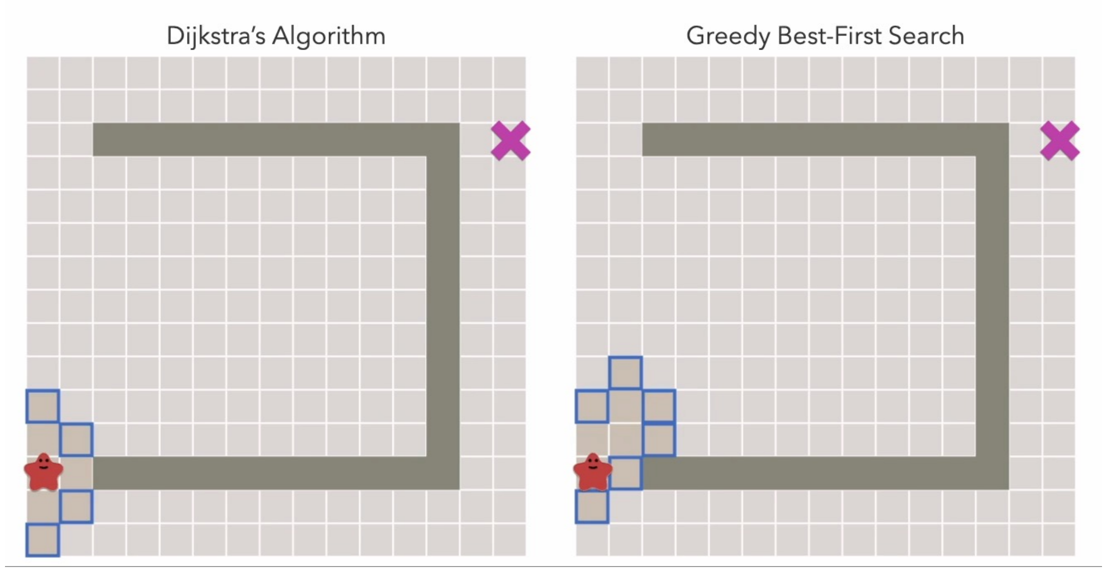
```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      if next not in came_from:
         priority = heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

Informed Search Attempt 1: Best-First Search

Pro: Great without obstacles



Con: Can return suboptimal paths with obstacles



Informed Search Attempt 2: A* Search

Choose the next node to expand as the one that has the lowest heuristic + cost so far

Greedy best first

Uniform cost search

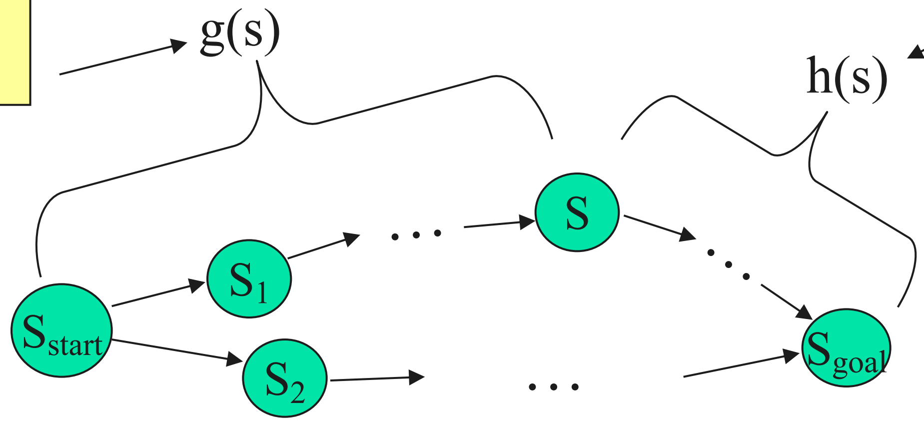
```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

A* Search: Setup

Computes optimal g-values for relevant states at any point of time

Cost-accumulated: the cost of a shortest path from s_{start} to s **found so far**

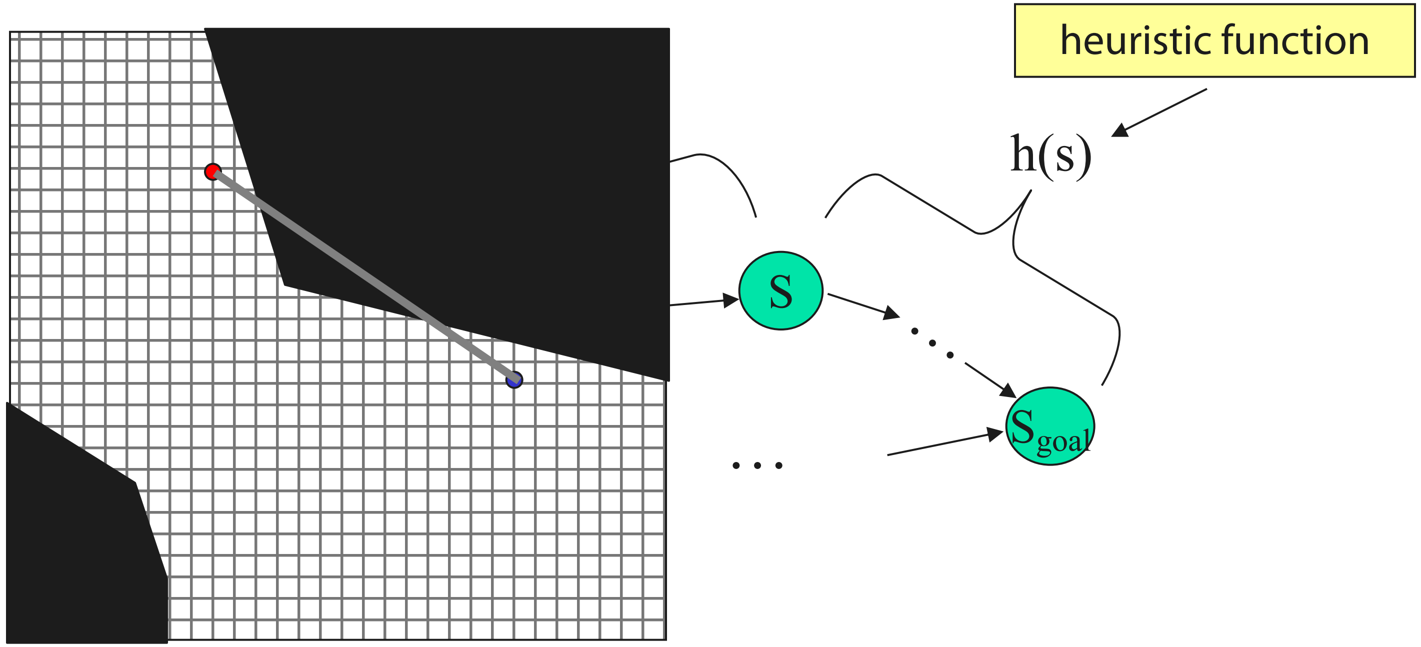
Heuristic: an (under) estimate of the cost of a shortest path from s to s_{goal}



g-value: shortest path so far from the start to a particular state

A* Search: Setup

Computes optimal g-values for relevant states at any point of time

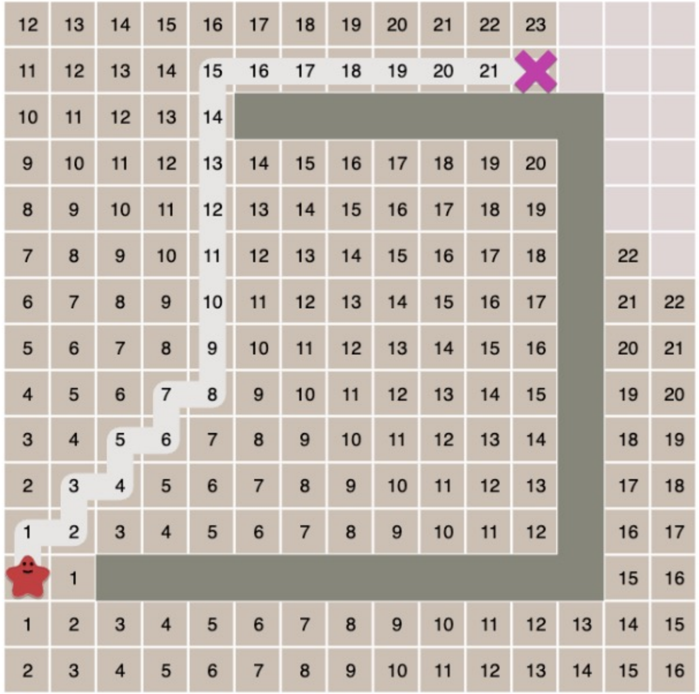


one popular heuristic function – Euclidean distance

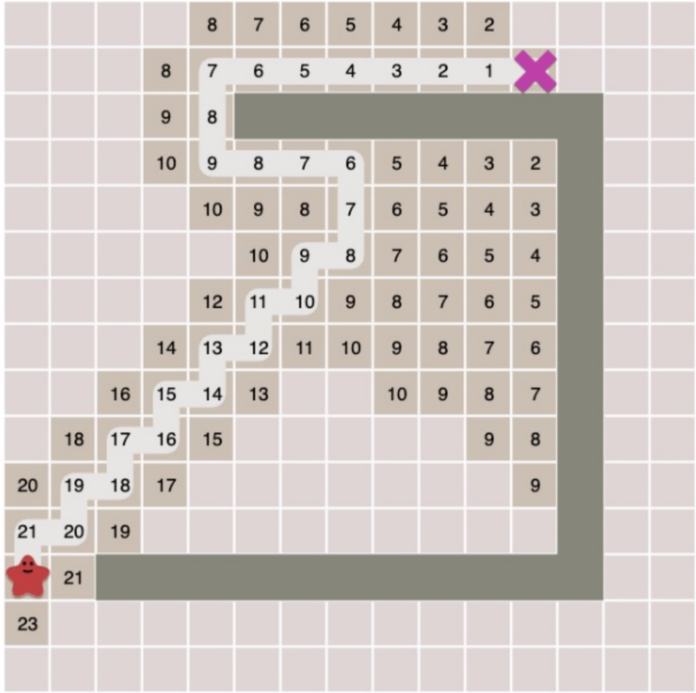
Why A* Search?

Combines the best of both greedy best first search and uniform cost search

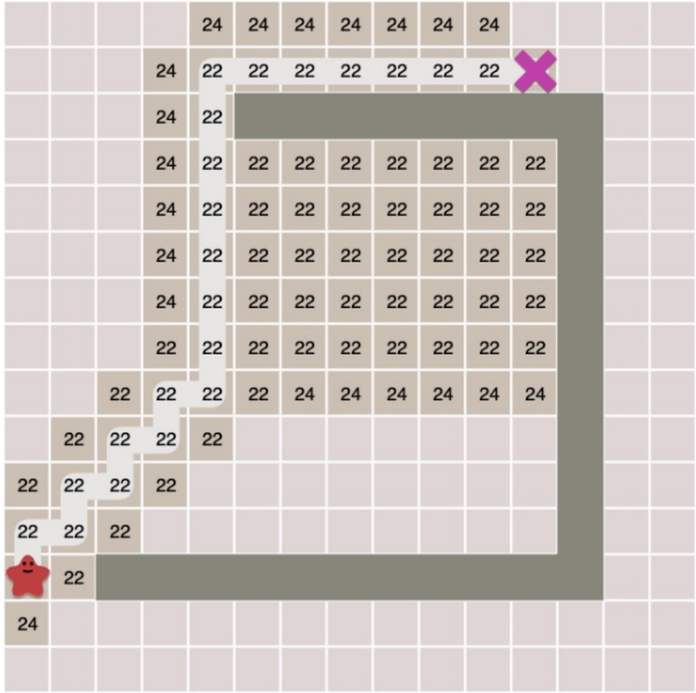
Dijkstra's Algorithm



Greedy Best-First



A* Search



Small number of node expansions

Guaranteed lowest cost path (assuming positive costs)

A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;

insert s into *CLOSED*;

for every successor s' of s such that s' not in *CLOSED*

if $g(s') > g(s) + c(s,s')$

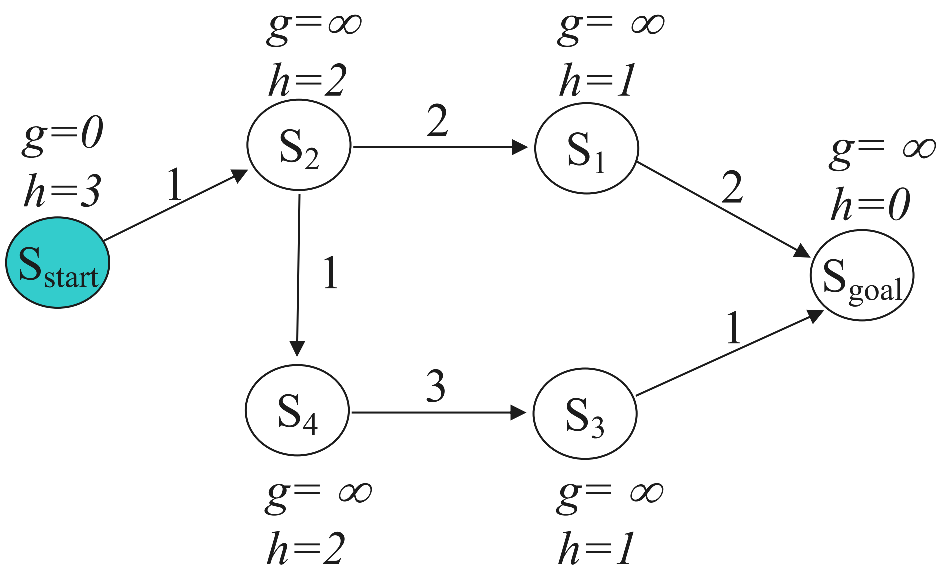
$g(s') = g(s) + c(s,s')$;

insert s' into *OPEN*;

$CLOSED = \{\}$

$OPEN = \{s_{start}\}$

next state to expand: s_{start}



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

  remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

  insert s into $CLOSED$;

  for every successor s' of s such that s' not in $CLOSED$

    if $g(s') > g(s) + c(s,s')$

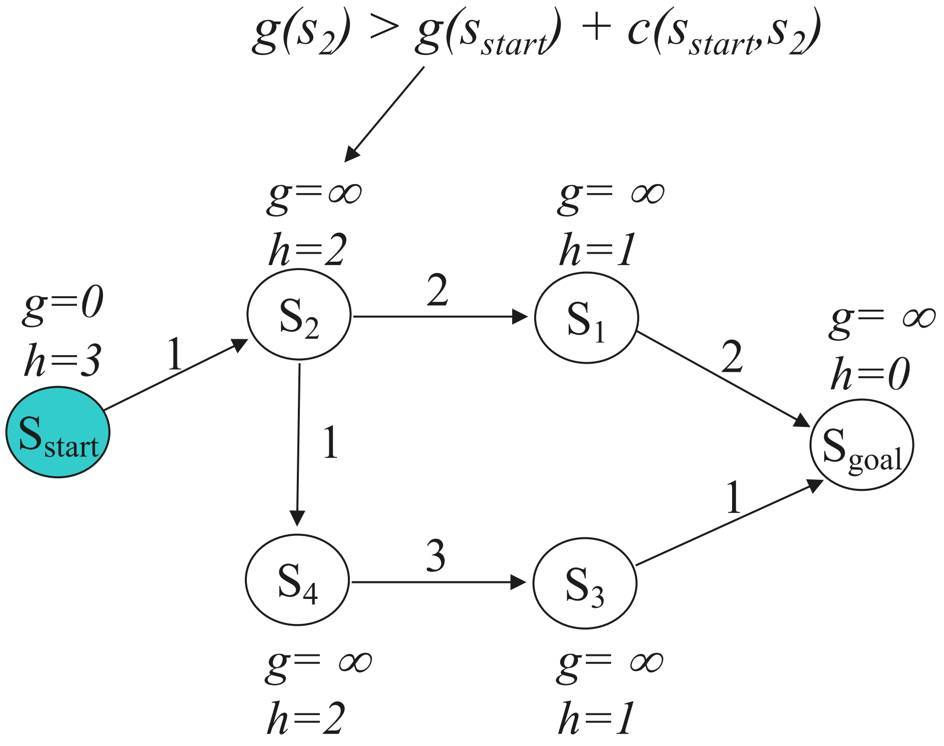
      $g(s') = g(s) + c(s,s')$;

      insert s' into $OPEN$;

$CLOSED = \{\}$

$OPEN = \{s_{start}\}$

next state to expand: s_{start}



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

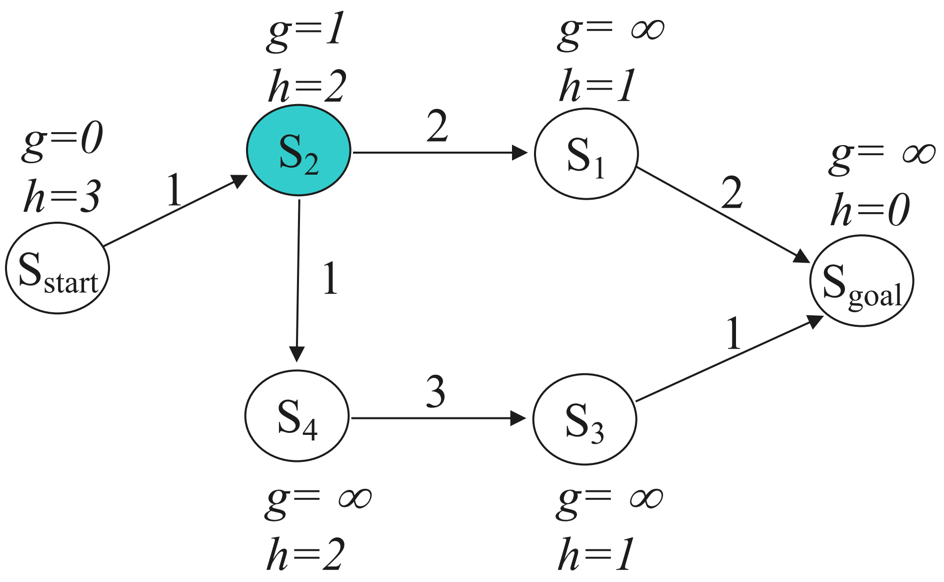
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{s_{start}\}$

$OPEN = \{s_2\}$

next state to expand: s_2



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;

insert s into *CLOSED*;

for every successor s' of s such that s' not in *CLOSED*

if $g(s') > g(s) + c(s,s')$

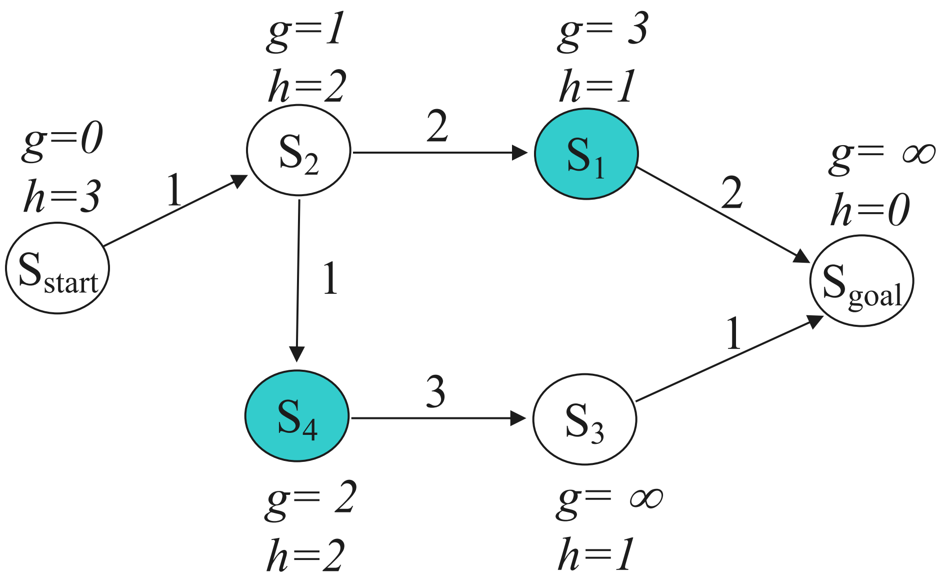
$g(s') = g(s) + c(s,s')$;

insert s' into *OPEN*;

$CLOSED = \{s_{start}, s_2\}$

$OPEN = \{s_1, s_4\}$

next state to expand: s_1



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

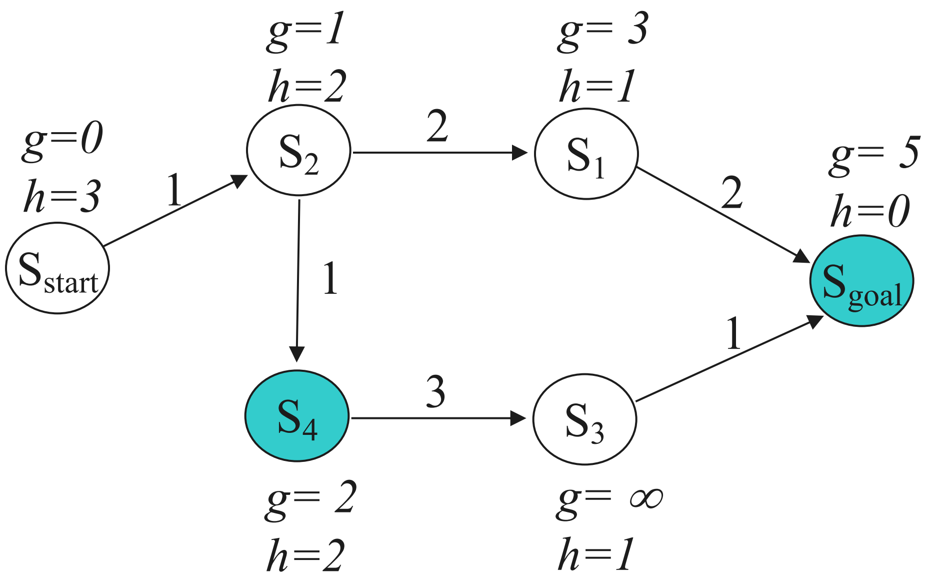
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{s_{start}, s_2, s_1\}$

$OPEN = \{s_4, s_{goal}\}$

next state to expand: s_4



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

  remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

  insert s into $CLOSED$;

  for every successor s' of s such that s' not in $CLOSED$

    if $g(s') > g(s) + c(s,s')$

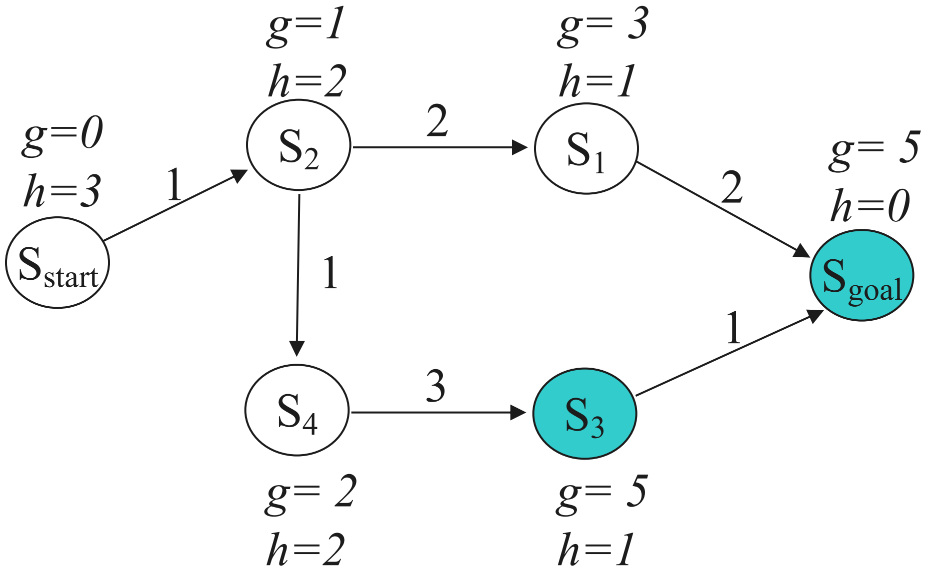
      $g(s') = g(s) + c(s,s')$;

      insert s' into $OPEN$;

$CLOSED = \{s_{start}, s_2, s_1, s_4\}$

$OPEN = \{s_3, s_{goal}\}$

next state to expand: s_{goal}



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

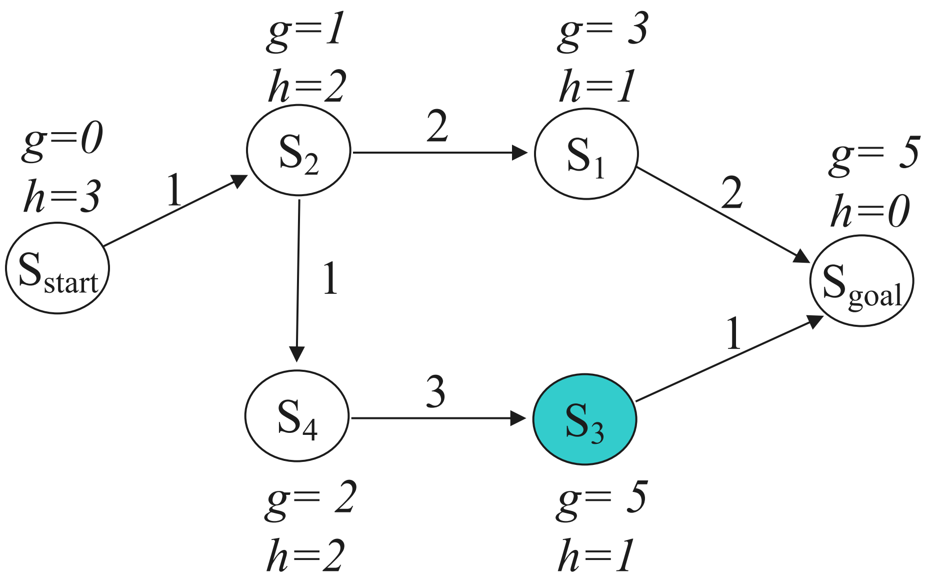
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{s_{start}, s_2, s_1, s_4, s_{goal}\}$

$OPEN = \{s_3\}$

done



A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

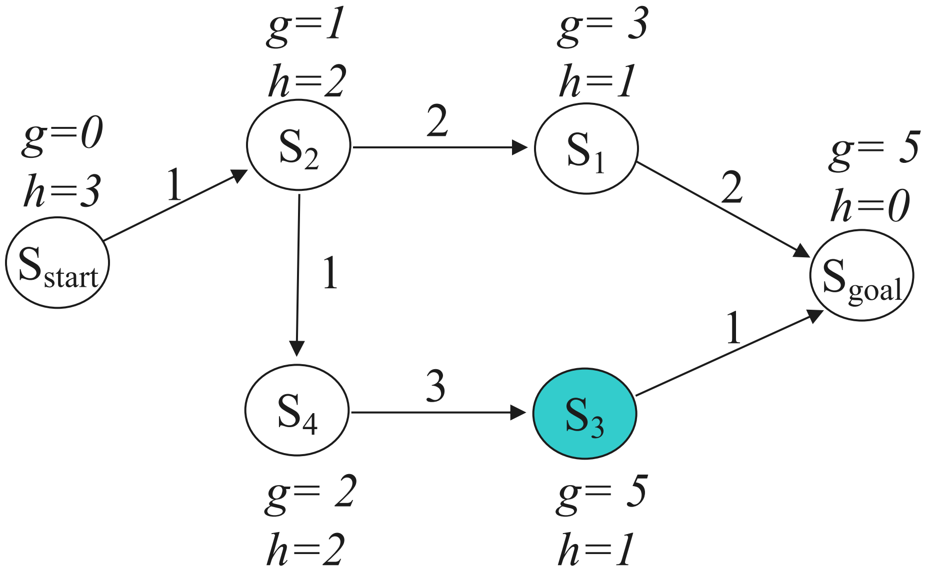
insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;



for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

A* Search

- Computes optimal g-values for relevant states

ComputePath function

while(s_{goal} is not expanded)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

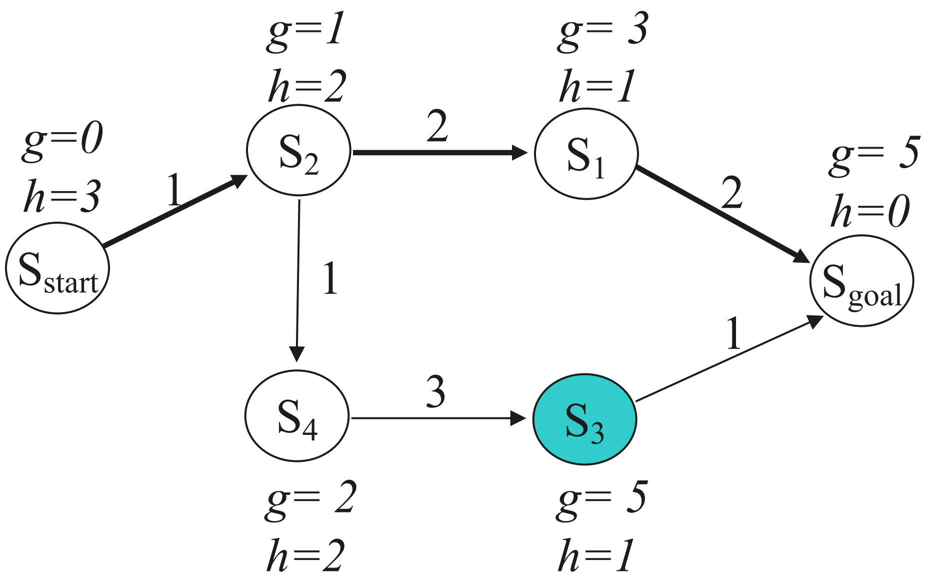
insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

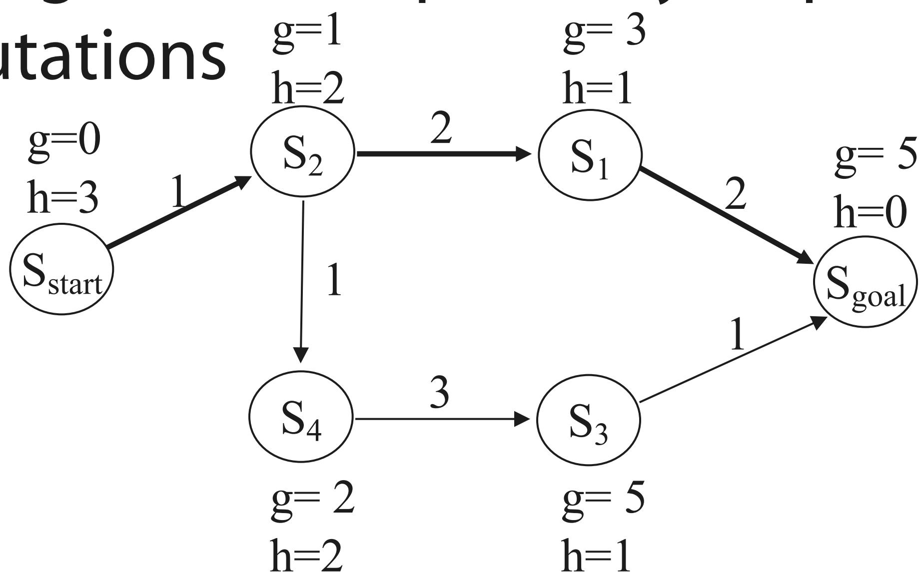


for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

A* Search

- Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution

- Performs provably minimal number of state expansions required to guarantee optimality – optimal in terms of the computations

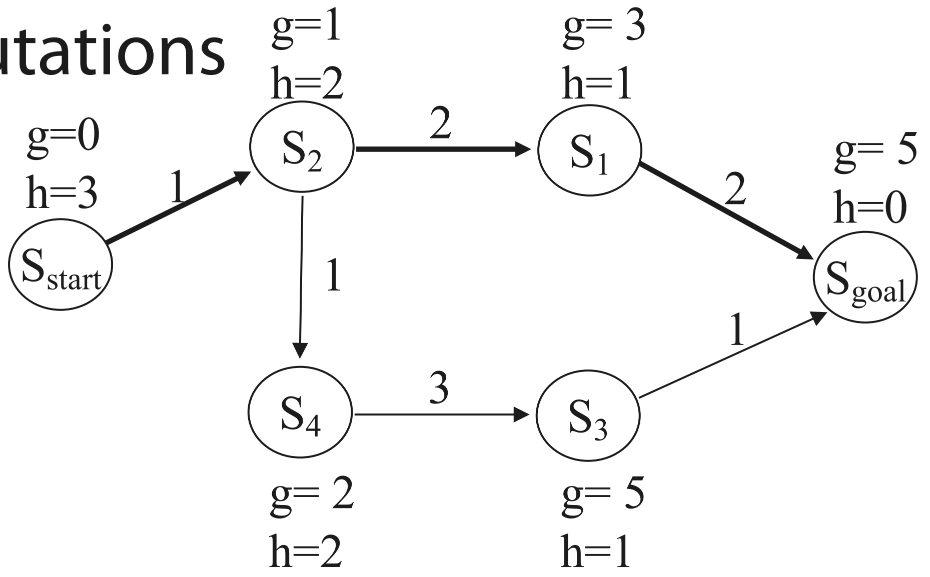


A* Search

- Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution

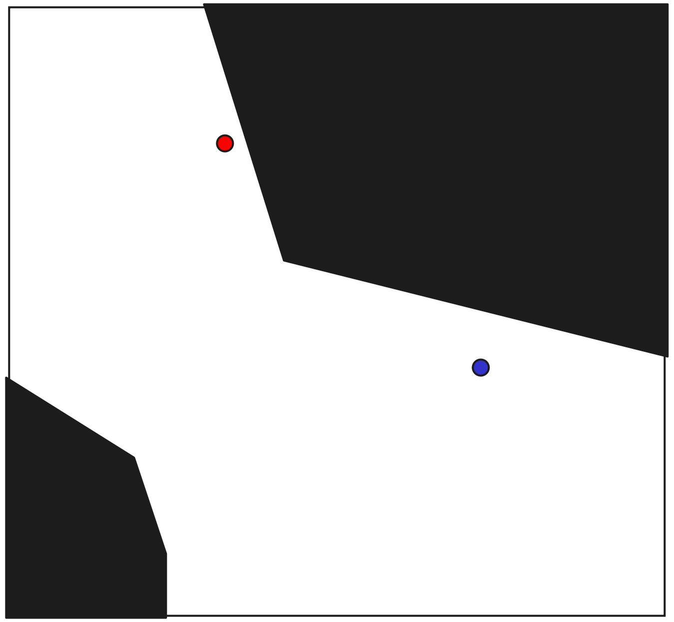
helps with robot deviating off its path if we search with A* backwards (from goal to start)

- Performs provably minimal number of state expansions required to guarantee optimality – optimal in terms of the computations

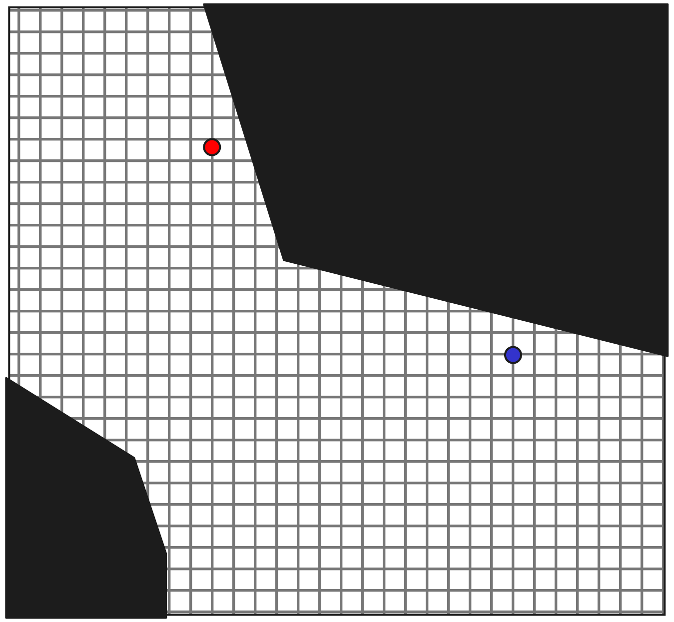


Connecting A* Search back to Motion Planning

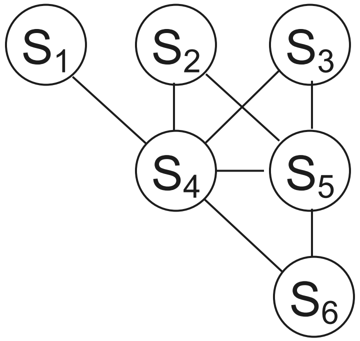
Step 1:
form the graph



discretize

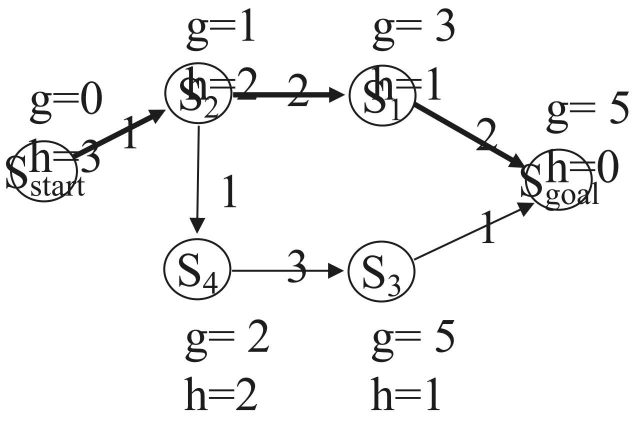


planning map



search the graph
for a least-cost path

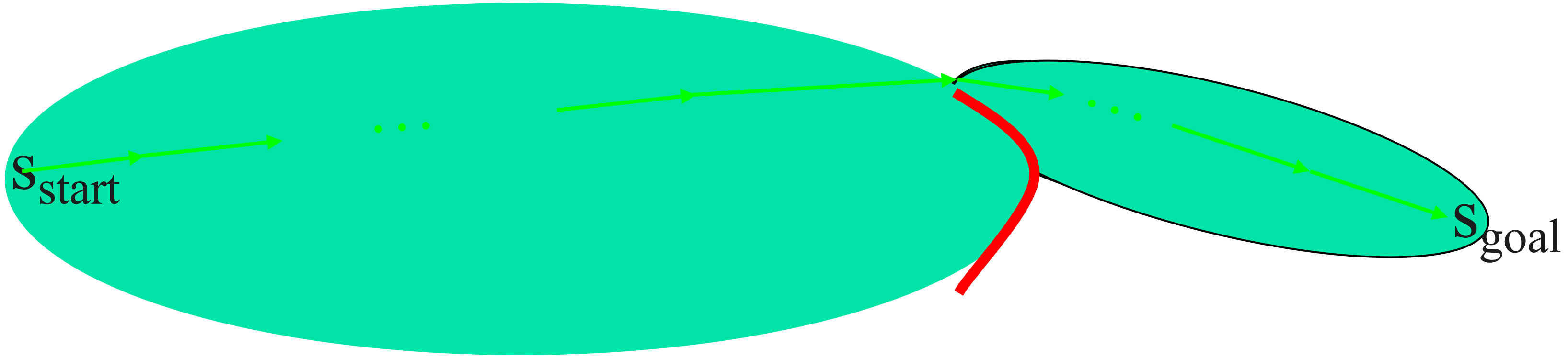
Step 2:
Search the graph



Step 3:
Execute on the robot

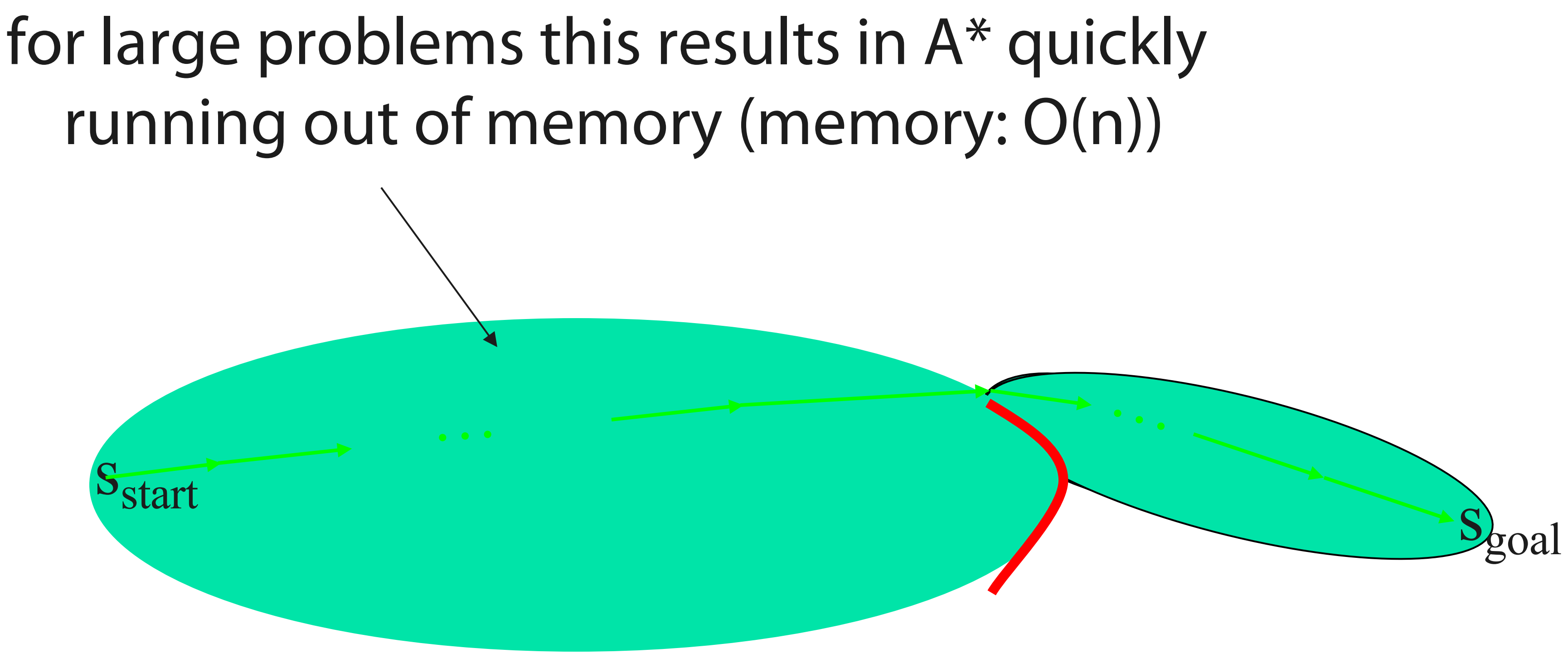
Effect of the Heuristic Function

A* Search: expands states in the order of ***f = g+h*** values



Effect of the Heuristic Function

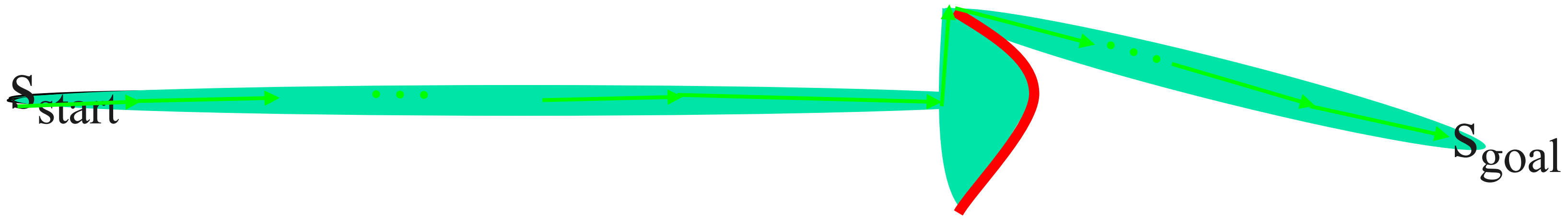
A* Search: expands states in the order of ***f* = *g*+*h*** values



Effect of the Heuristic Function

- Weighted A* Search: expands states in the order of ***f* = *g*+ε*h*** values, ***ε* > 1** = bias towards states that are closer to goal

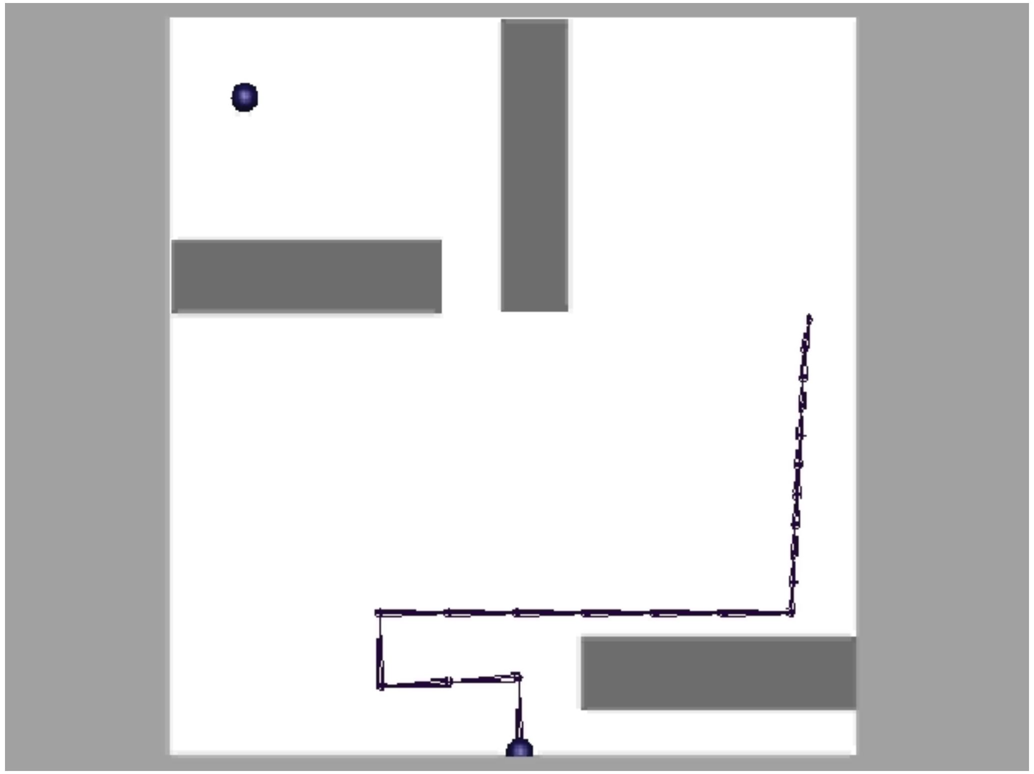
solution is always ε-suboptimal:
cost(solution) ≤ ε·cost(optimal solution)



Effect of the Heuristic Function

Weighted A* Search: expands states in the order of $f = g + \varepsilon h$ values, $\varepsilon > 1$ = bias towards states that are closer to goal

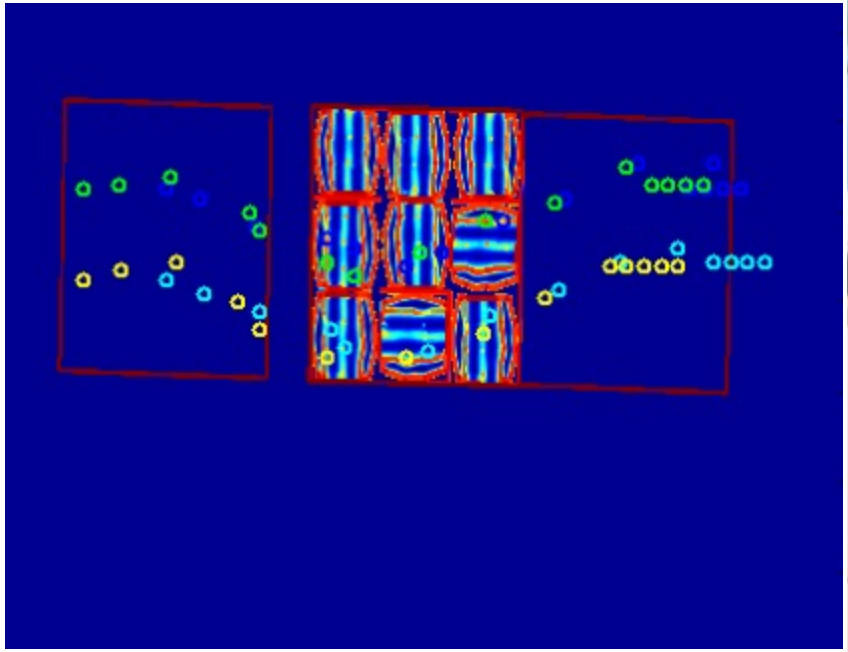
20DOF simulated robotic arm
state-space size: over 10^{26} states



planning with ARA* (anytime version of weighted A*)

Effect of the Heuristic Function

- planning in 8D (<x,y> for each foothold)
- heuristic is Euclidean distance from the center of the body to the goal location
- cost of edges based on kinematic stability of the robot and quality of footholds




planning with R* (randomized version of weighted A*)

joint work with Subhrajit Bhattacharya, Jon Bohren, Sachin Chitta, Daniel D. Lee, Aleksandr Kushleyev, Paul Vernaza

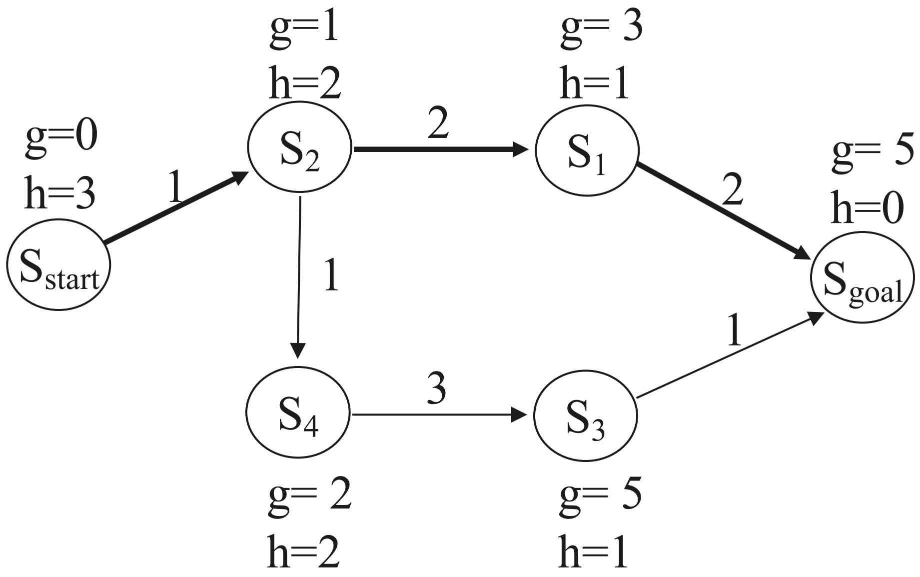
Is A* always optimal for all heuristics?

Admissible → underestimate

$h(s) < h^*(s)$

Consistent → monotone

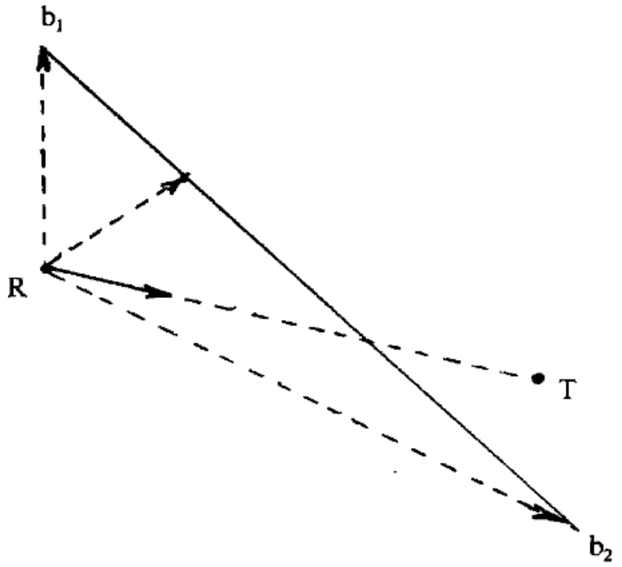
$h(s) < c(s, s') + h(s')$

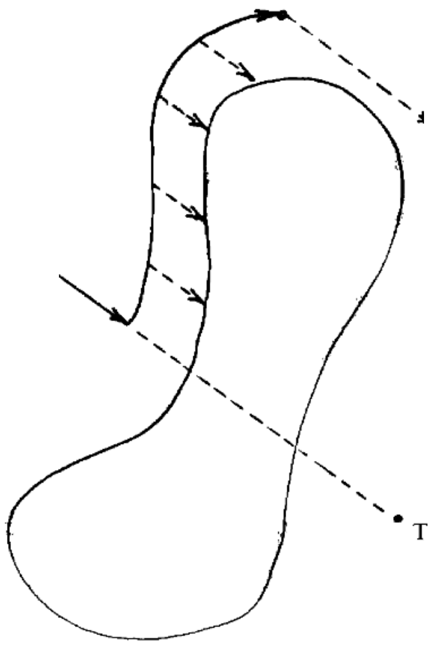


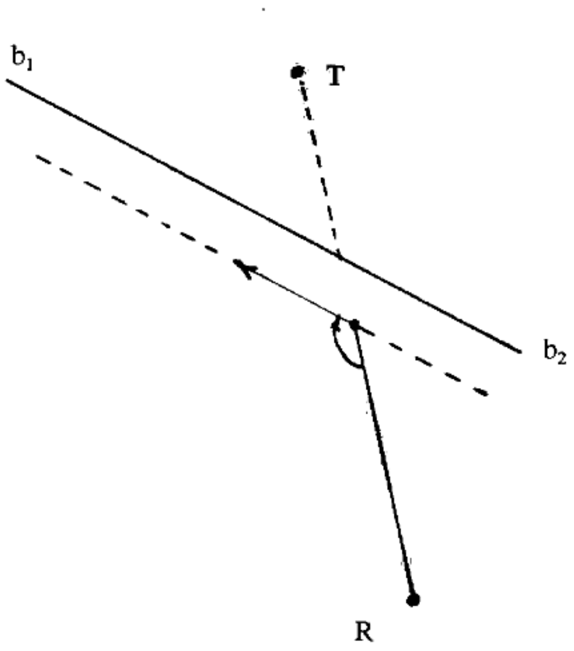
A* search returns optimal paths on graphs only when the heuristic is admissible and consistent

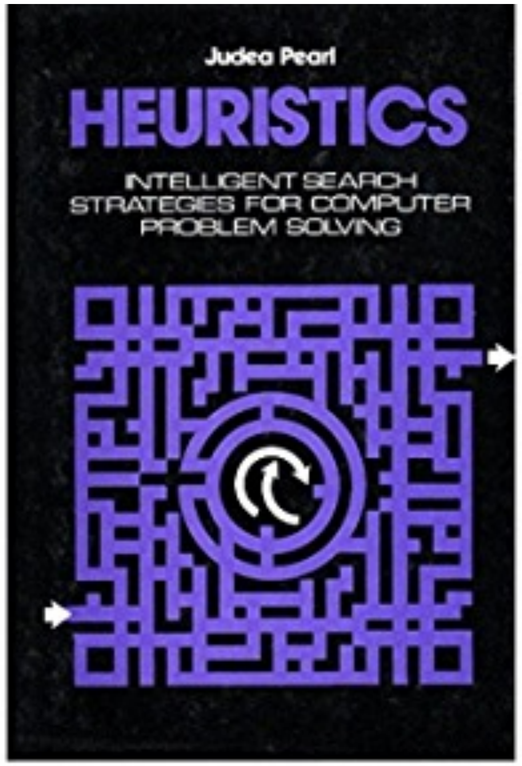
Common Heuristics in Robotics

Art more than a science – commonly used heuristics are Euclidean/Manhattan distance or distance through coarse/convexified obstacles



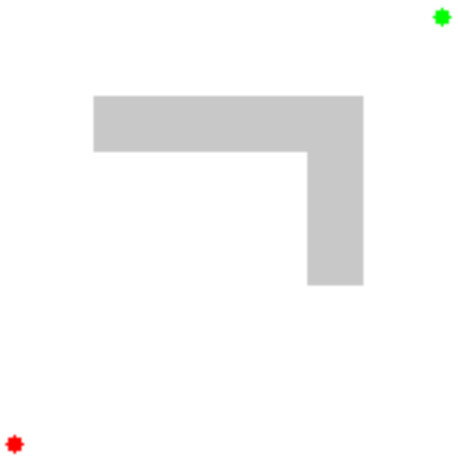




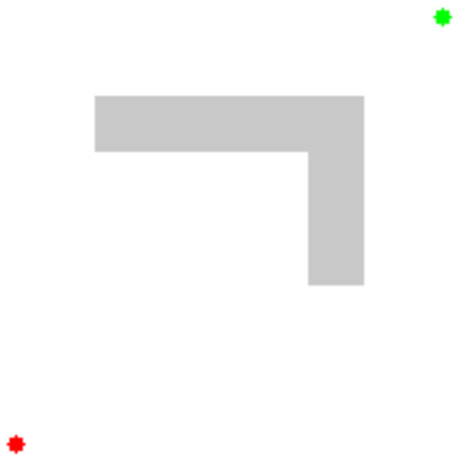


Visualization of Search

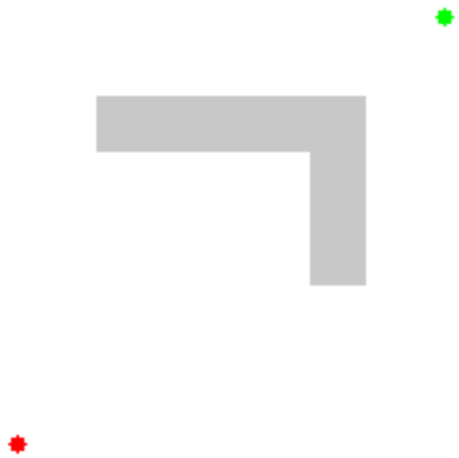
Uniform cost search



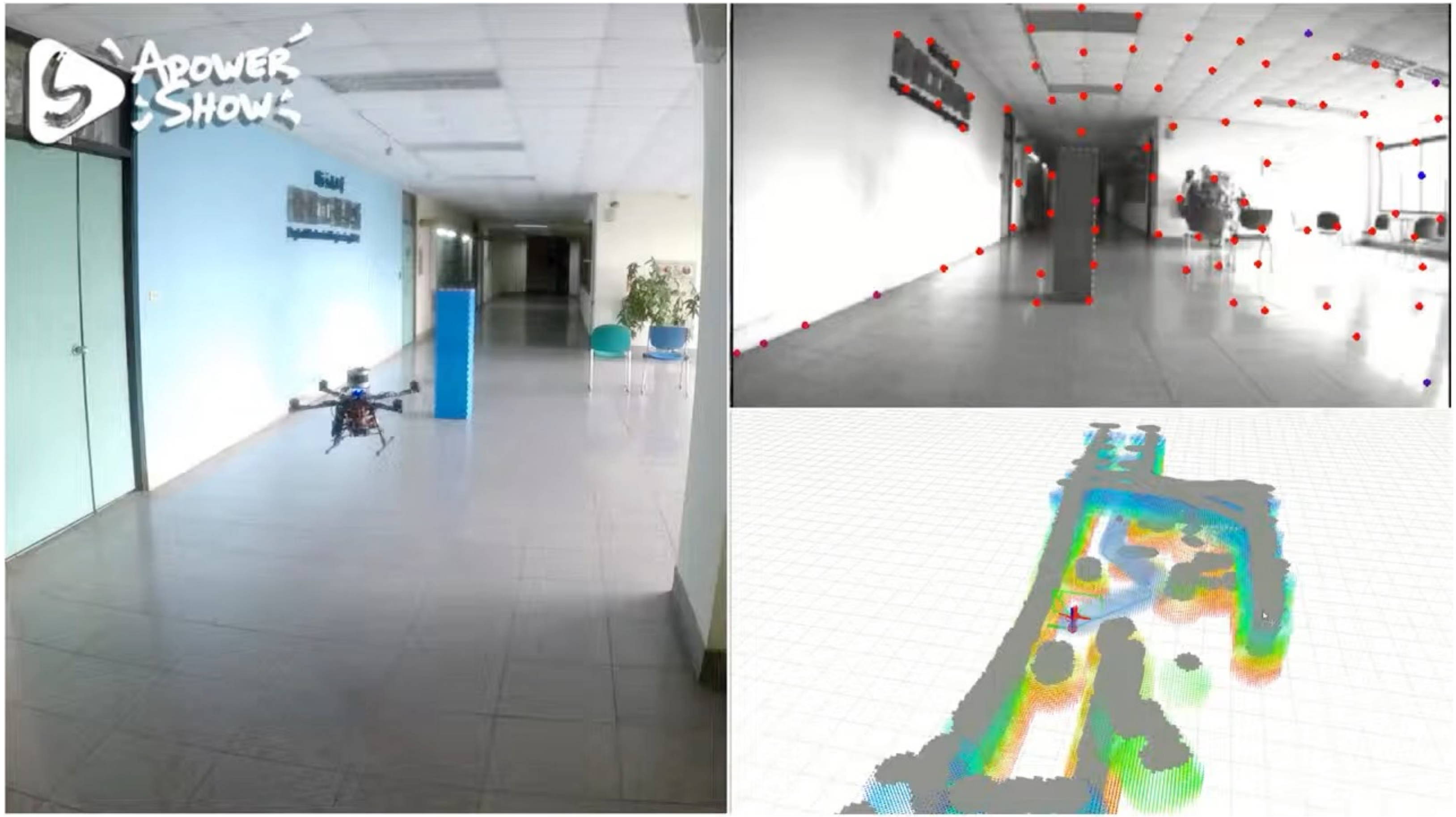
A* search



Weighted A* search



Motion Planning via Search



Lecture Outline

Casting motion planning as a search problem

↓

Motion Planning via A* search

↓

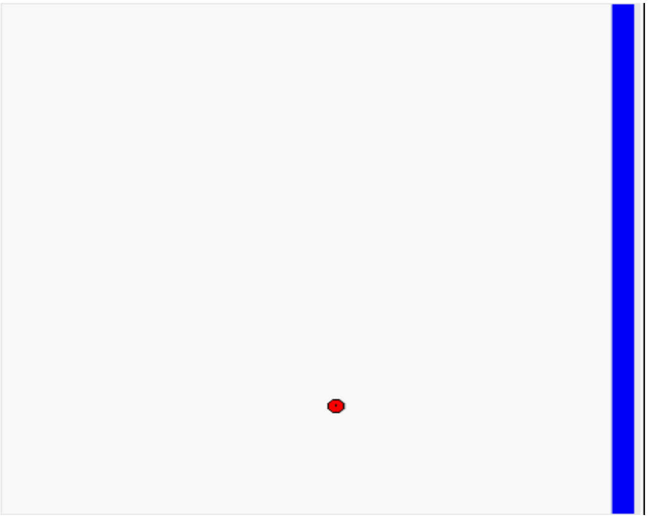
Incremental Search for Replanning

Incremental version of A* (LPA*)

- Robot needs to re-plan whenever
  - new information arrives (partially-known environments or/and dynamic environments)
  - robot deviates off its path

ATRV navigating
initially-unknown environment

planning map and path

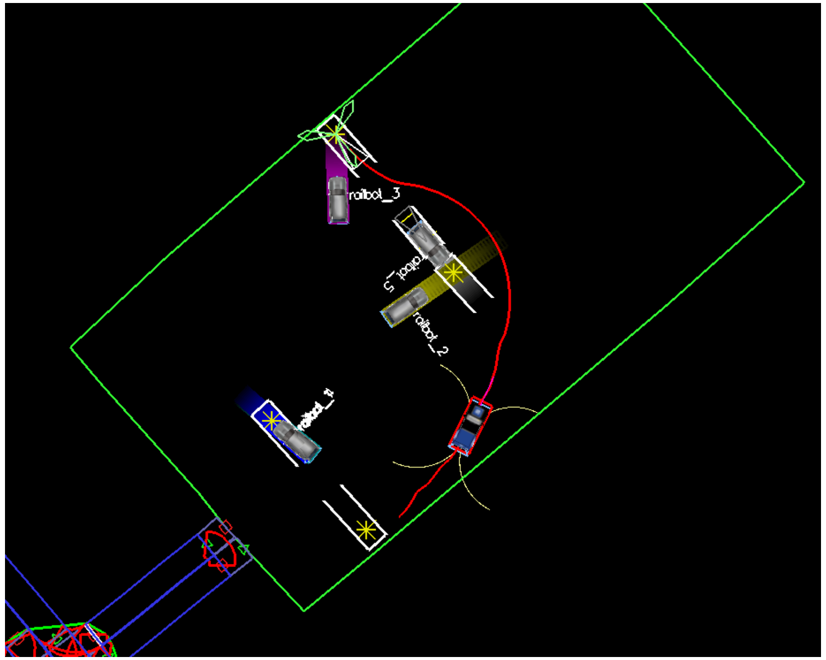


Incremental version of A* (LPA*/D*/D* Lite)

- Robot needs to re-plan whenever
  - new information arrives (partially-known environments or/and dynamic environments)
  - robot deviates off its path

incremental planning (re-planning): reuse of previous planning efforts

planning in dynamic environments

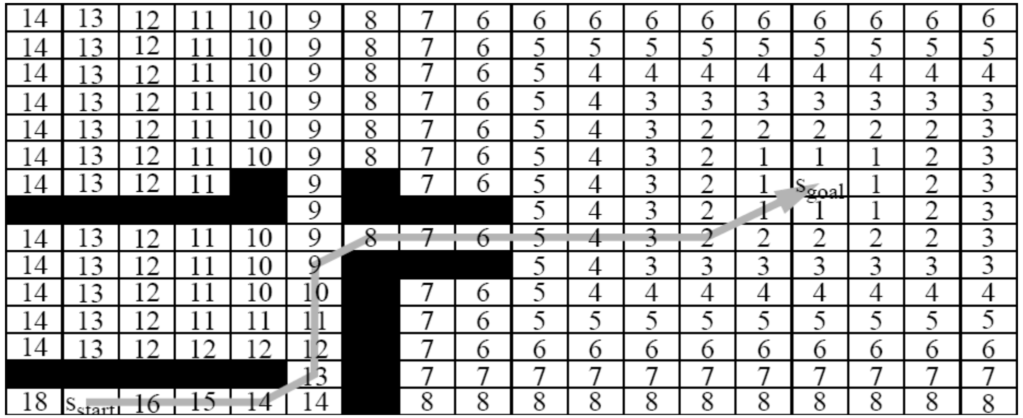


Tartanracing, CMU

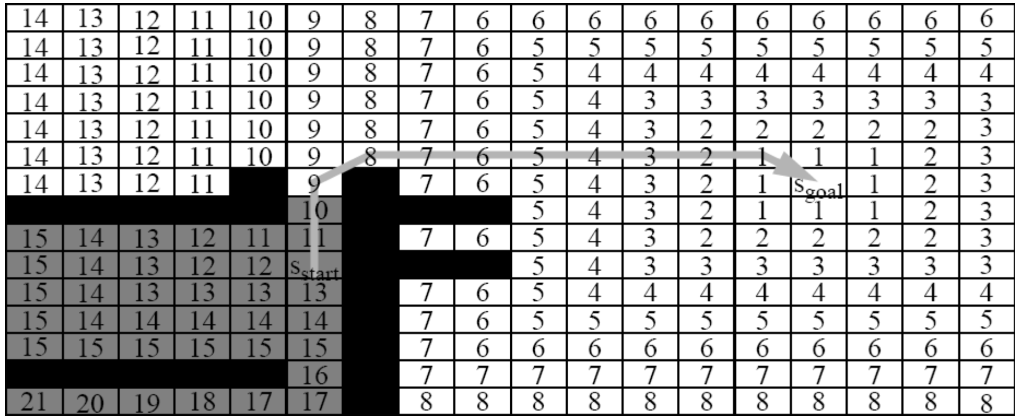
Motivation for Incremental Version of A*

- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially



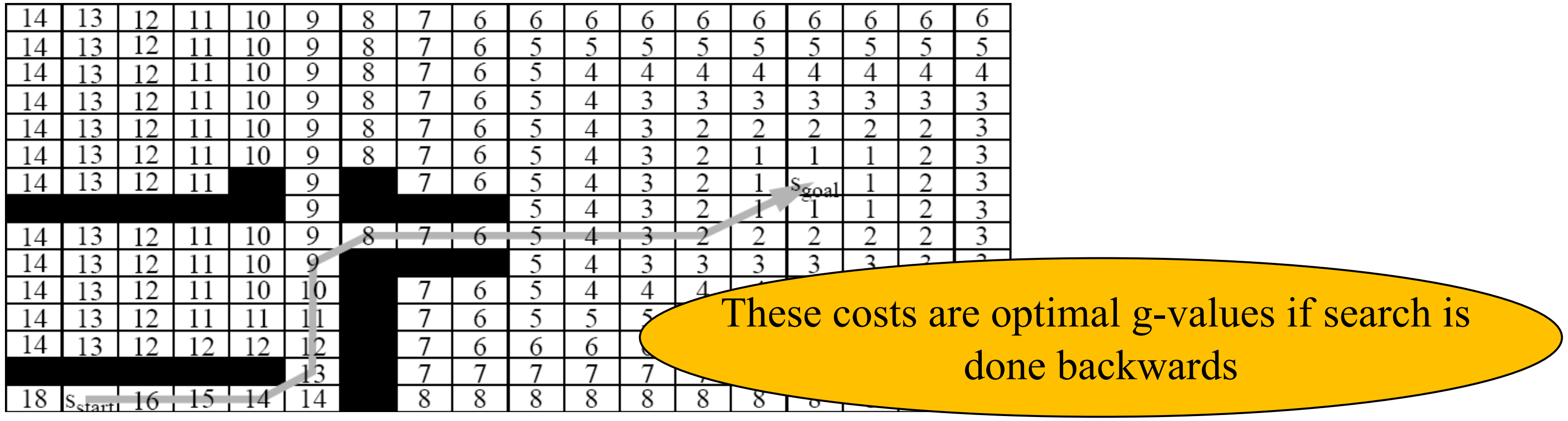
cost of least-cost paths to s_{goal} after the door turns out to be closed



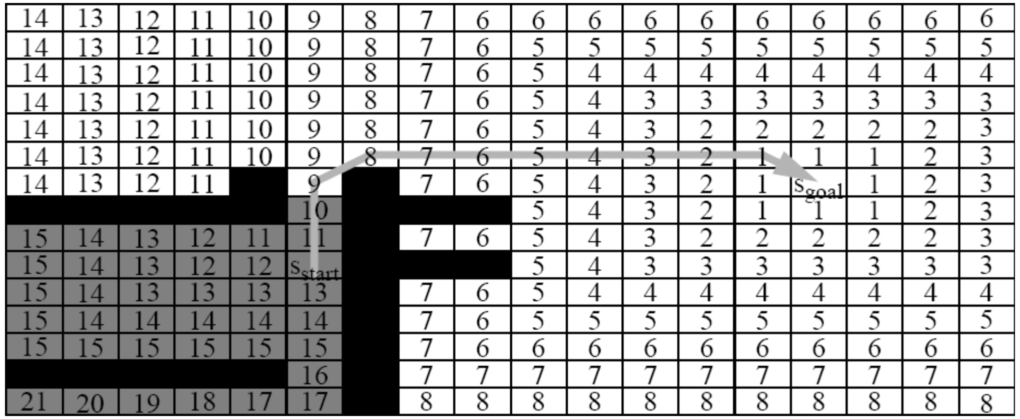
Motivation for Incremental Version of A*

- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially



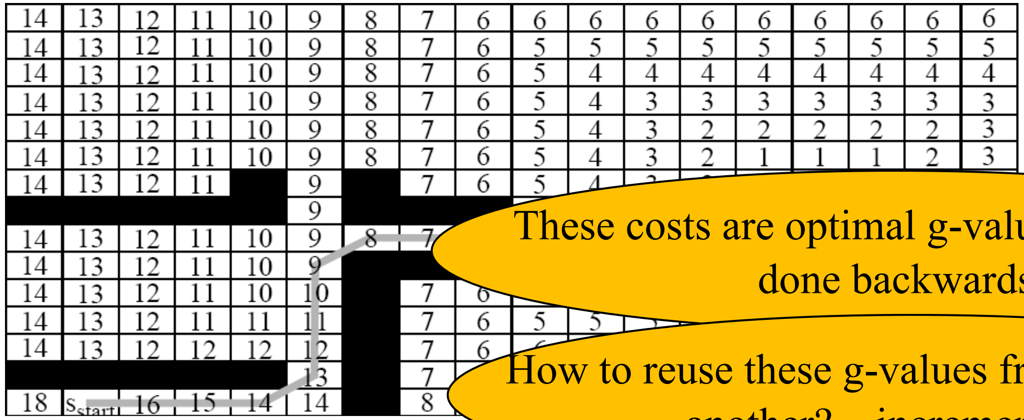
cost of least-cost paths to s_{goal} after the door turns out to be closed



Motivation for Incremental Version of A*

- Reuse state values from previous searches

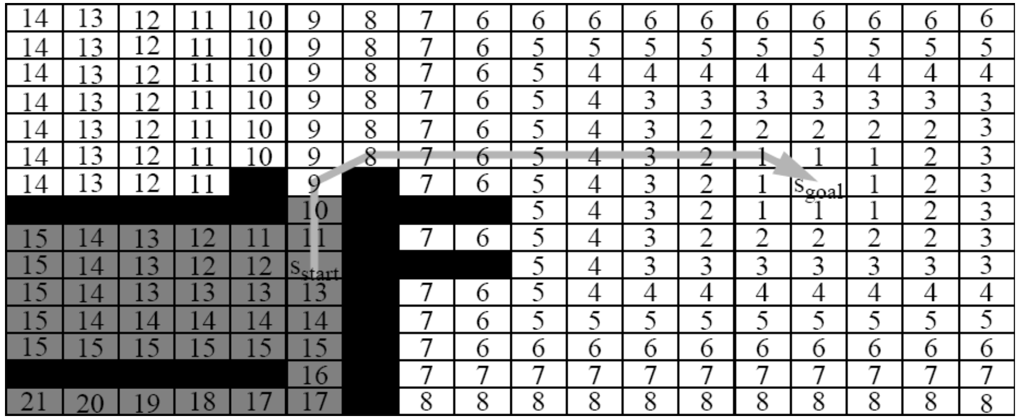
cost of least-cost paths to s_{goal} initially



These costs are optimal g-values if search is done backwards

How to reuse these g-values from one search to another? – incremental A*

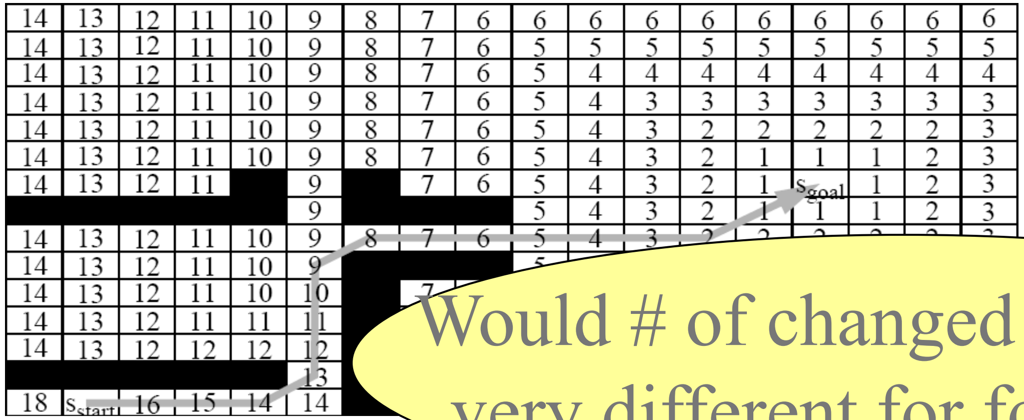
cost of least-cost paths to s_{goal} after the door turns out to be closed



Motivation for Incremental Version of A*

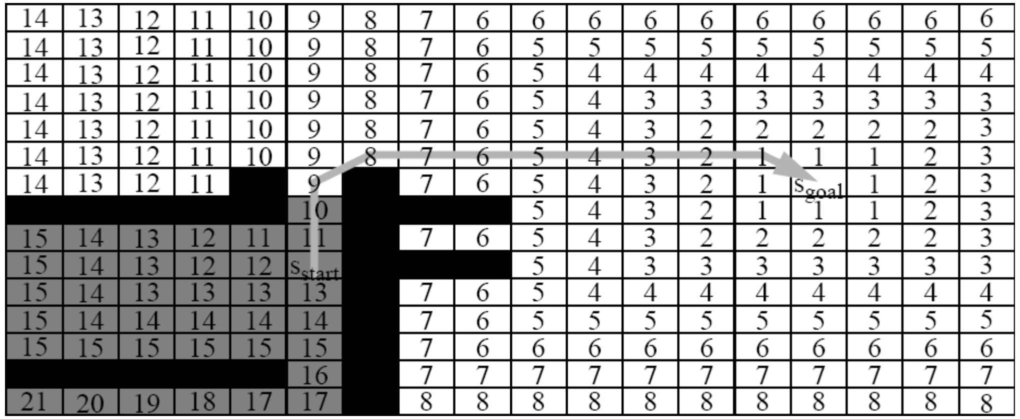
- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially



Would # of changed g-values be very different for forward A*?

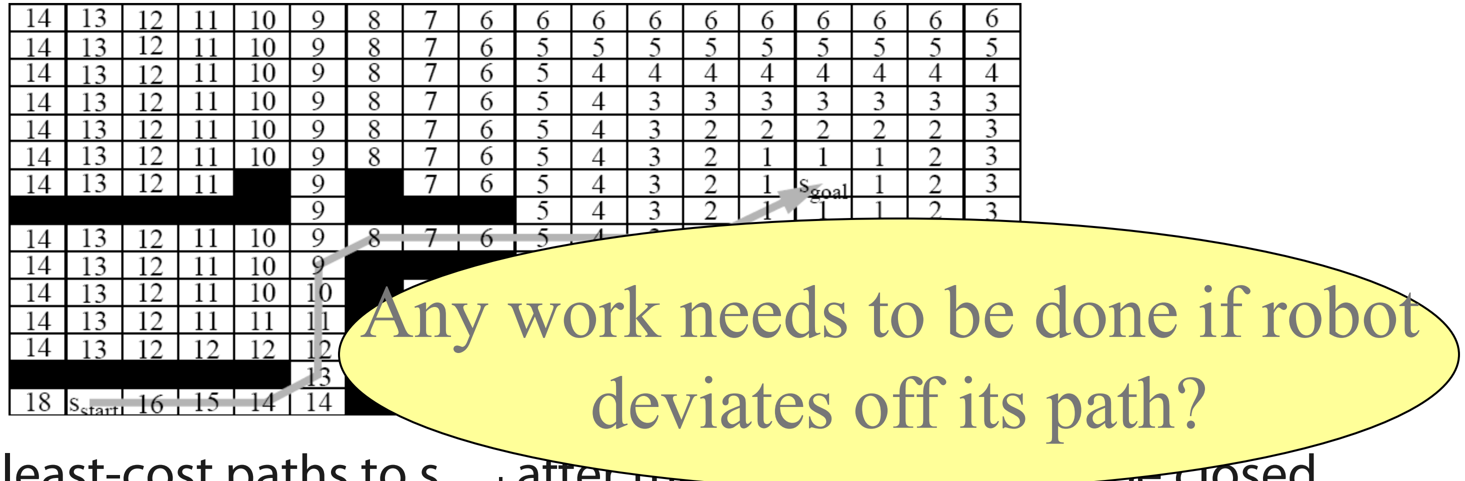
cost of least-cost paths to s_{goal} after th… to be closed



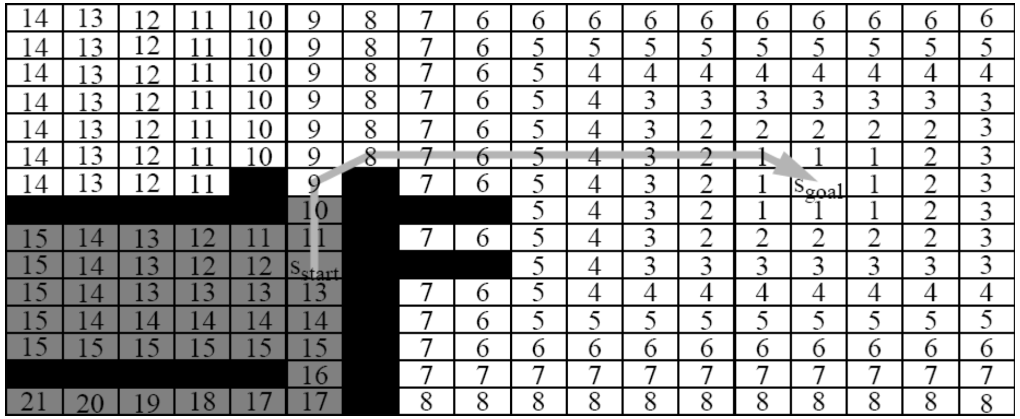
Motivation for Incremental Version of A*

- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially



cost of least-cost paths to s_{goal} after the door turns out to be closed



Incremental Version of A*

- Reuse state values from previous searches