CSE 571: Robotics

Motion Planning

Tapomayukh Bhattacharjee

28th January 2019

Many slides courtesy of Maxim Likhachev, Howie Choset, Siddhartha Srinivasa, and Seth Teller

Motion Planning in Robotics

Planning: Process of thinking about and organizing the activities required to achieve a desired goal

Motion Planning: Convert high-level task specification to low-level descriptions of how to move

Specification

Motion Planning Problem

Model of the Robot (states and actions)
Model of the world
Current state of the robot
Current state of the world
Cost function (Optional)
Desired state(s) of the robot

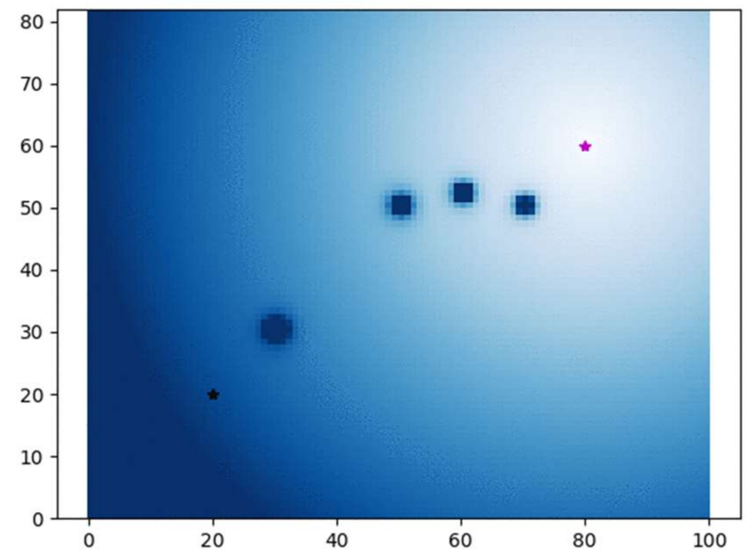
Solution

Plan that prescribes a sequence of actions
Plan terminates at the desired state
Optionally minimizes the cost of executing the actions

Omnidirectional Robot

Motion Planning Problem

- States and Actions?
- World specification?
- Current state of the robot
- Current state of the world
- Possible cost functions?
- Desired state(s) of the robot

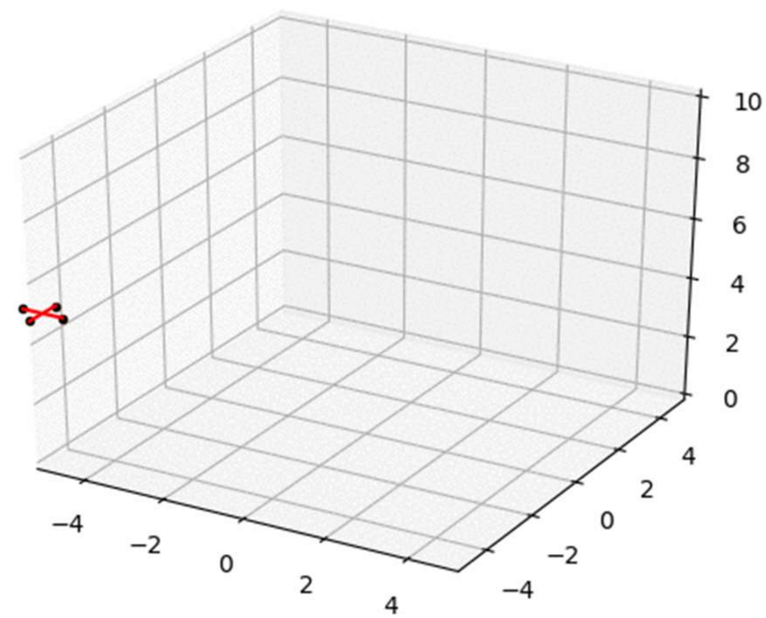


Examples?

Drones

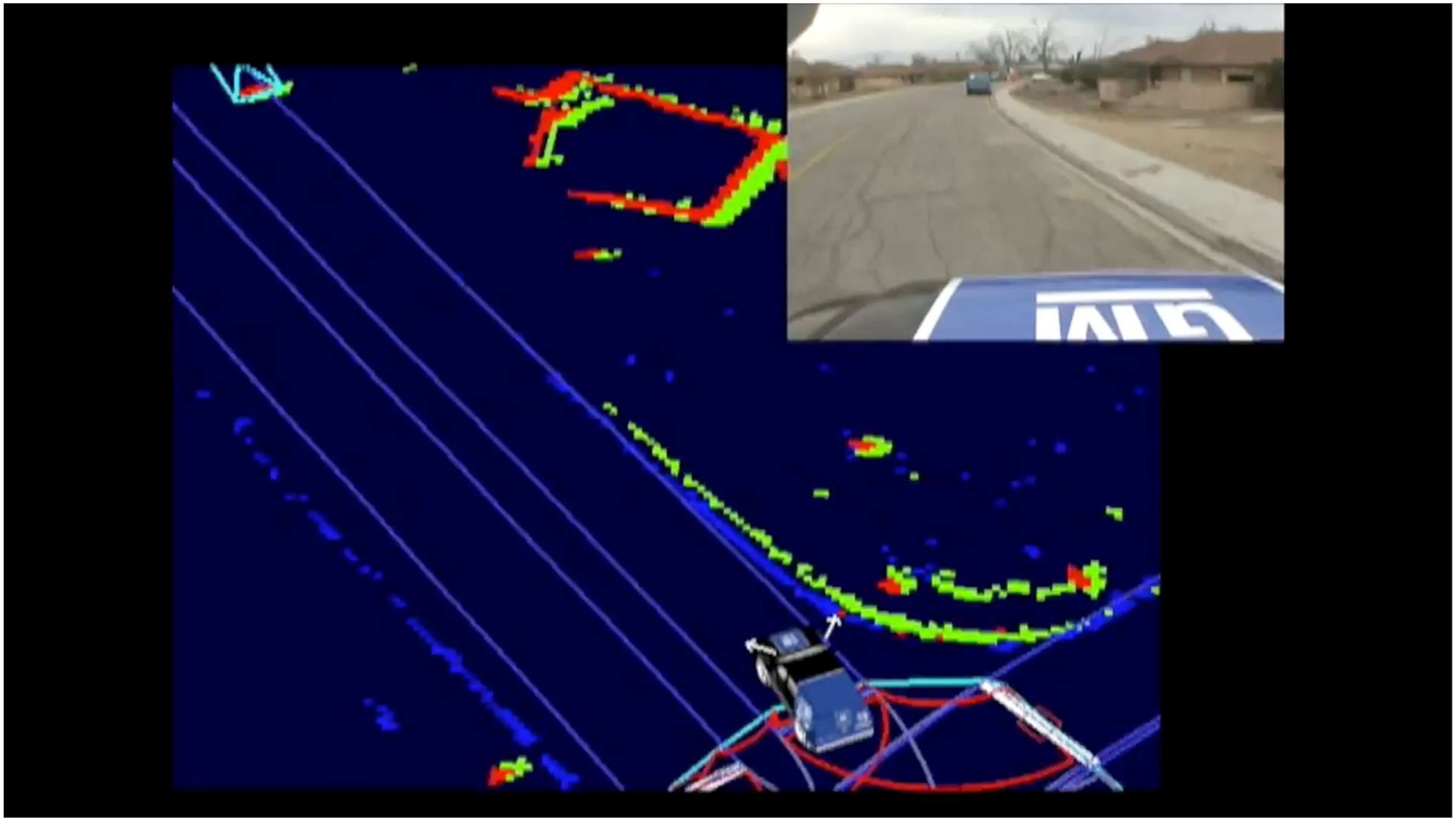
Motion Planning Problem

- States and Actions?
- World specification?
- Current state of the robot
- Current state of the world
- Possible cost functions?
- Desired state(s) of the robot



Drones

Autonomous Driving

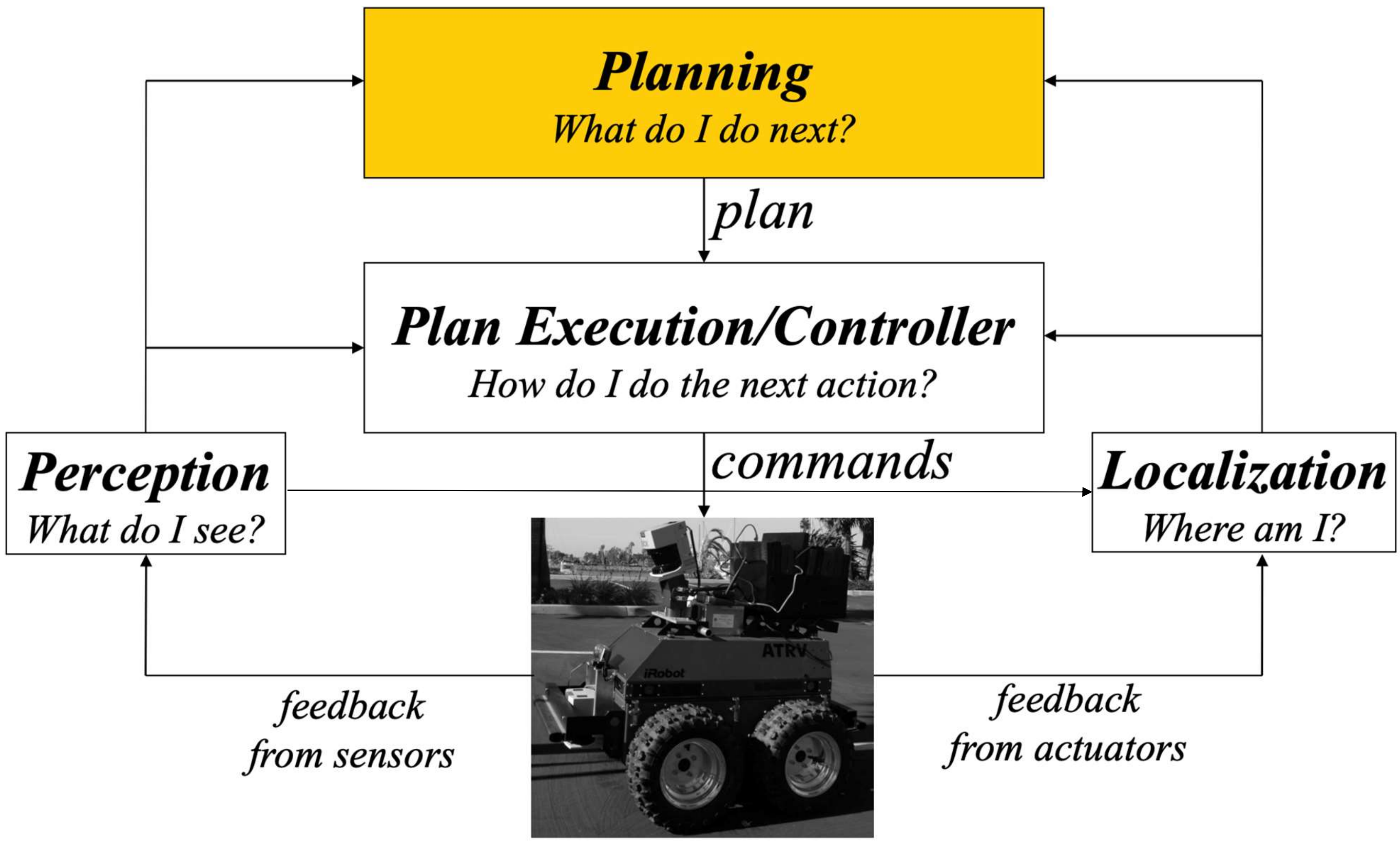


Urban Challenge Race, CMU Team, Planning with Anytime D* (A* with Replan)

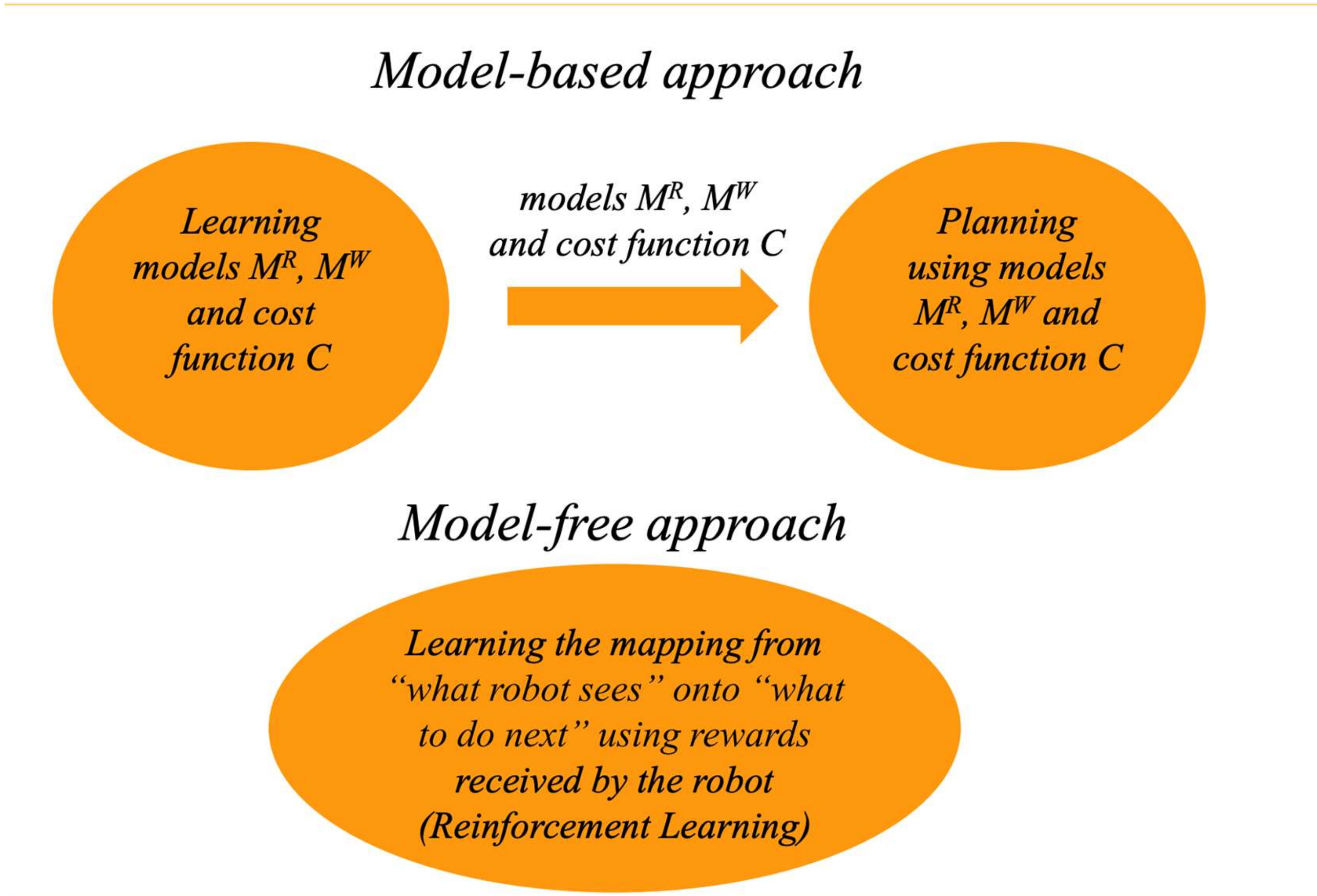
Manipulation

Food Manipulation: Pick up fork using planning with LRA* (Lazy variant of A*)

Where does Planning fit?

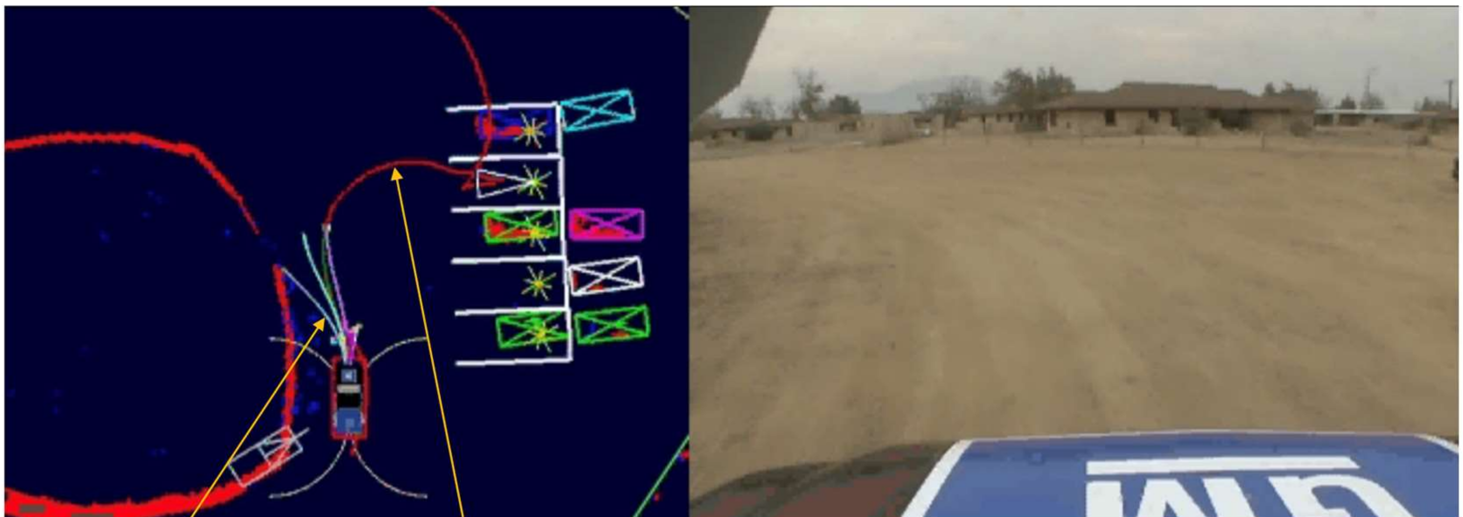


Planning and Learning



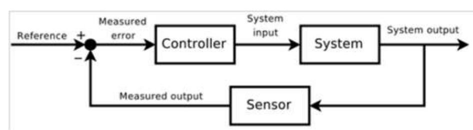
Deterministic vs. Under Uncertainty

Planning and Control

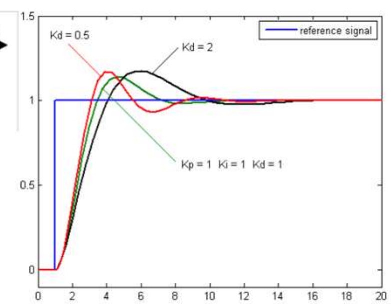


local planning
(trajectory following)

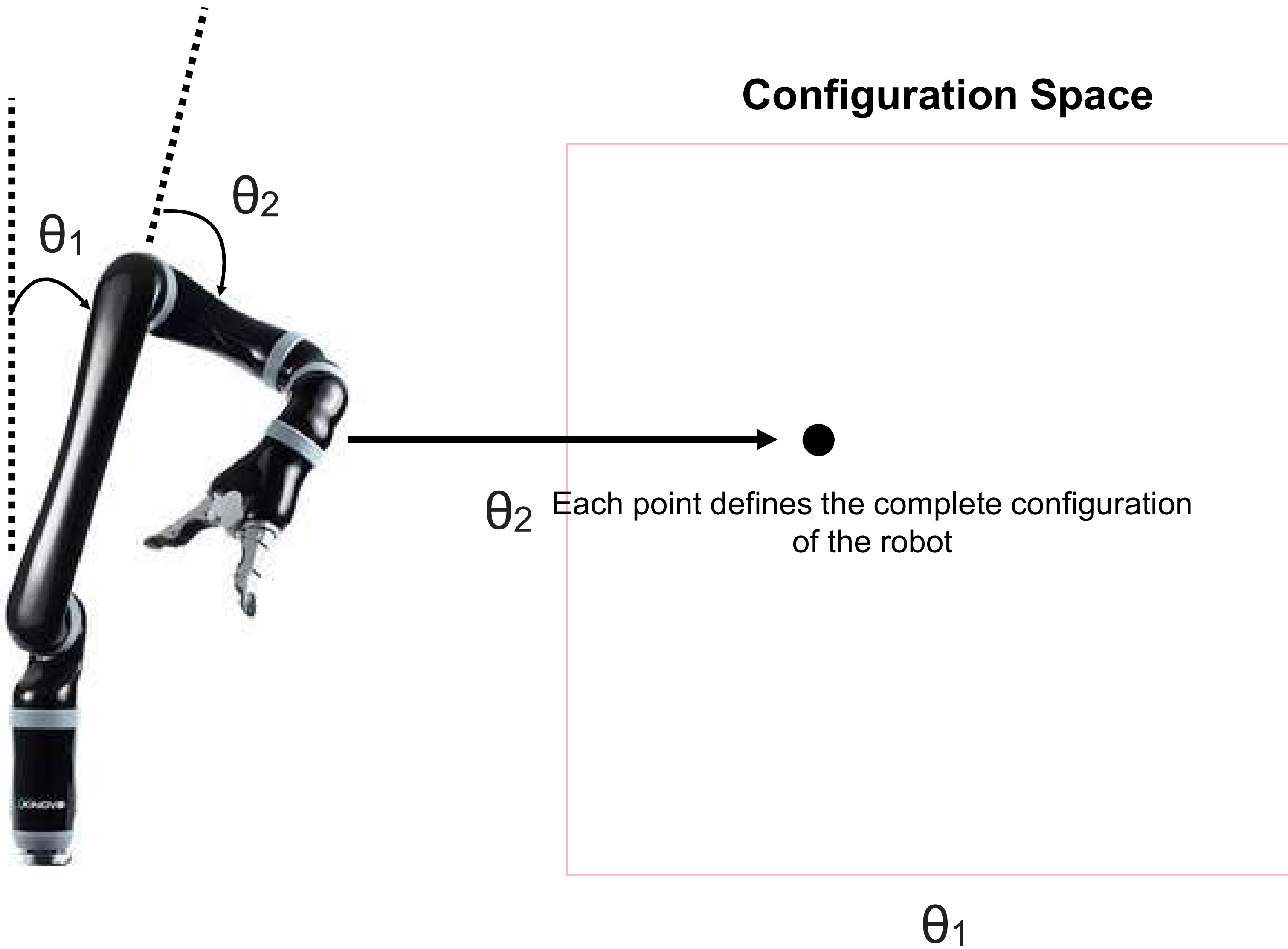
global planning



controller



What space to plan in?

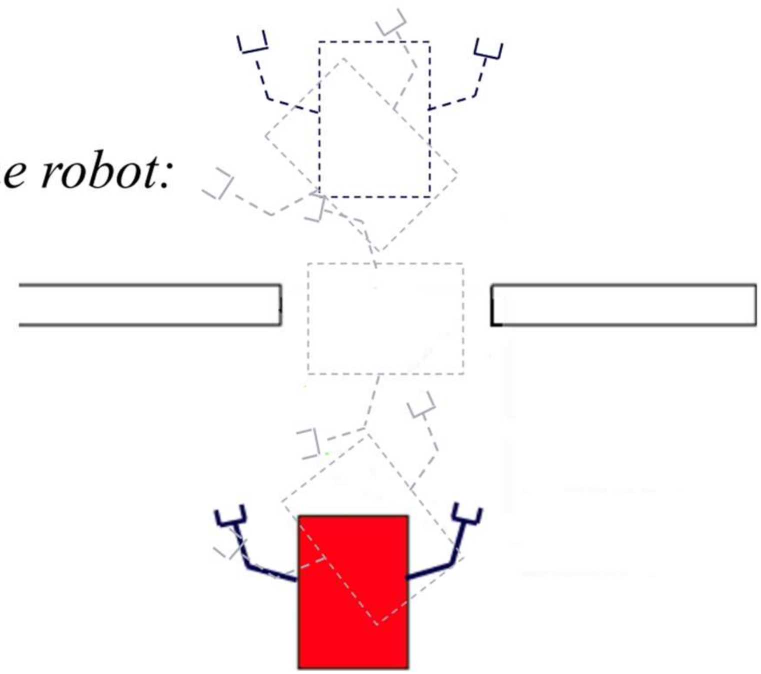


Configuration Space

A configuration is legal if
- it is not in collision
- is valid (within limits)

A configuration space is the set of legal configurations

Legal configurations for the base of the robot:



What is the dimensionality of the configuration space of the base?

Configuration Space: 2D base robot

Configuration space for a robot base in 2D world is:

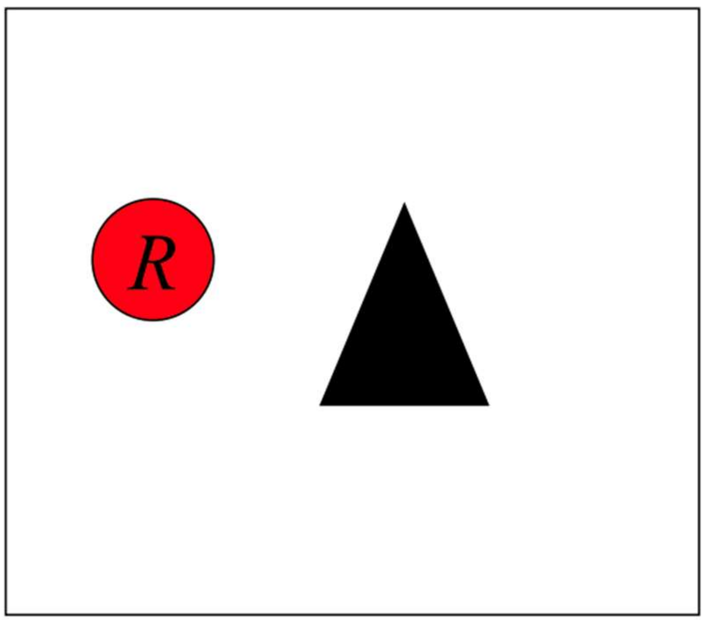
- 2D if the robot is circular (symmetric in all directions)
- 3D if the robot is non-circular (asymmetric)

Why?

Configuration Space: 2D base robot

Configuration space for a robot base in 2D world is:

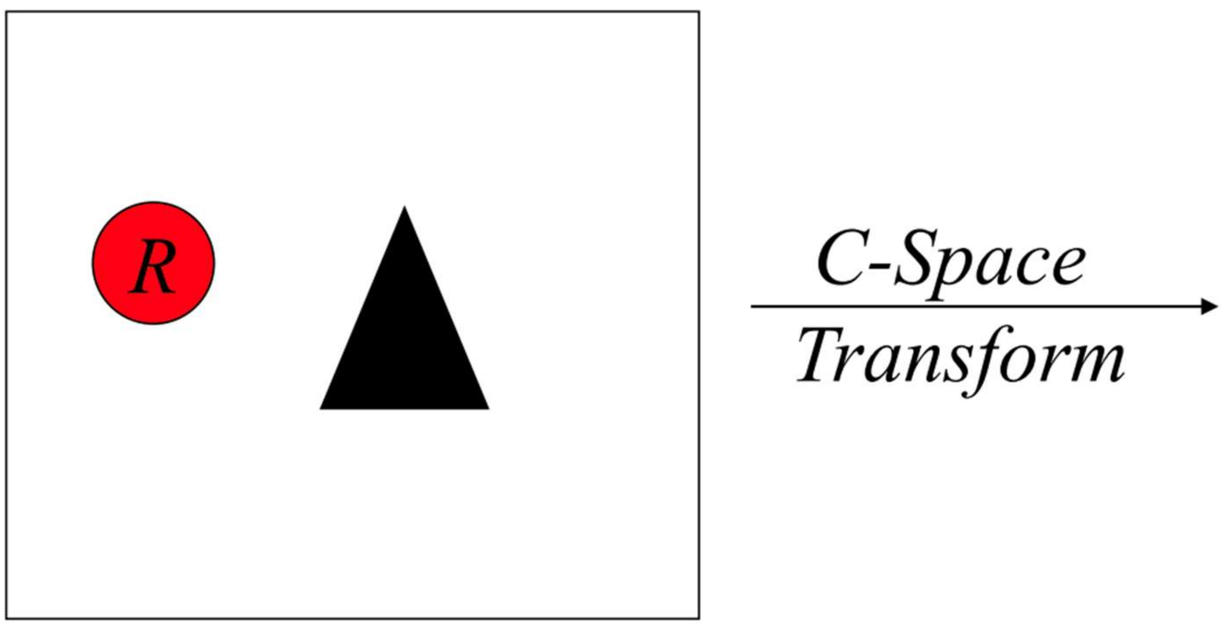
- 2D if the robot is circular (symmetric in all directions)



Configuration Space: 2D base robot

Configuration space for a robot base in 2D world is:

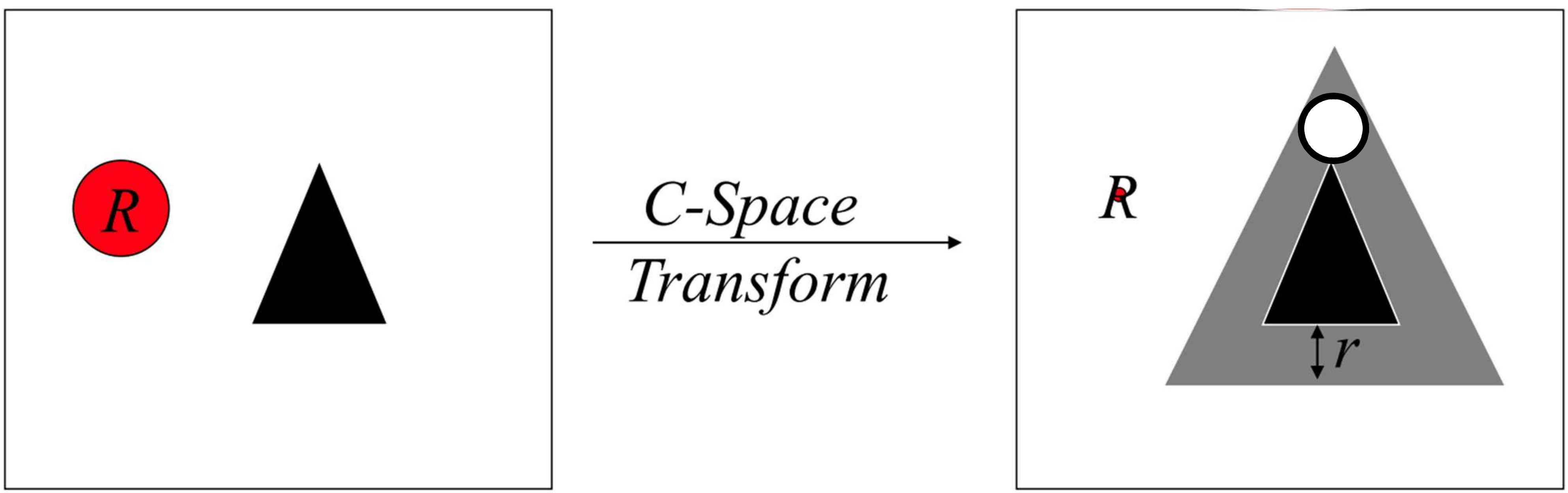
- 2D if the robot is circular (symmetric in all directions)



Configuration Space: 2D base robot

Configuration space for a robot base in 2D world is:

- 2D if the robot is circular (symmetric in all directions)

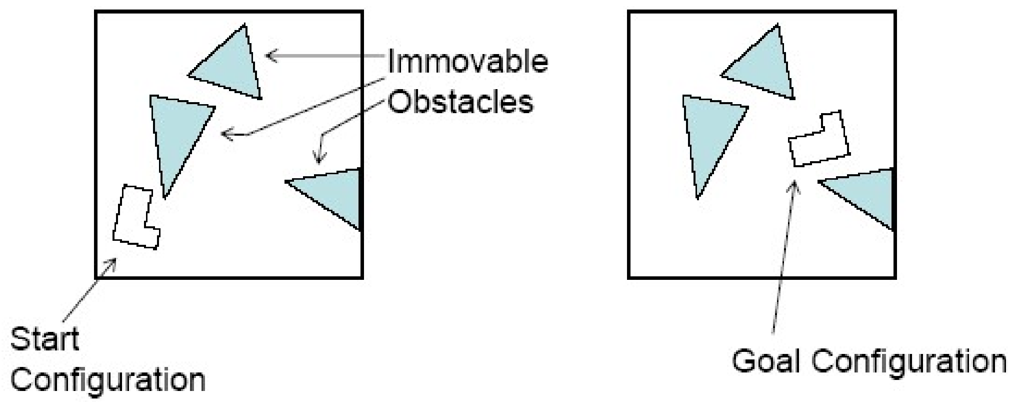


Is this a correct transformation?

Configuration Space

Mathematical Representation (See Notes)

Motion Planning: Piano Movers' Problem (See Notes)

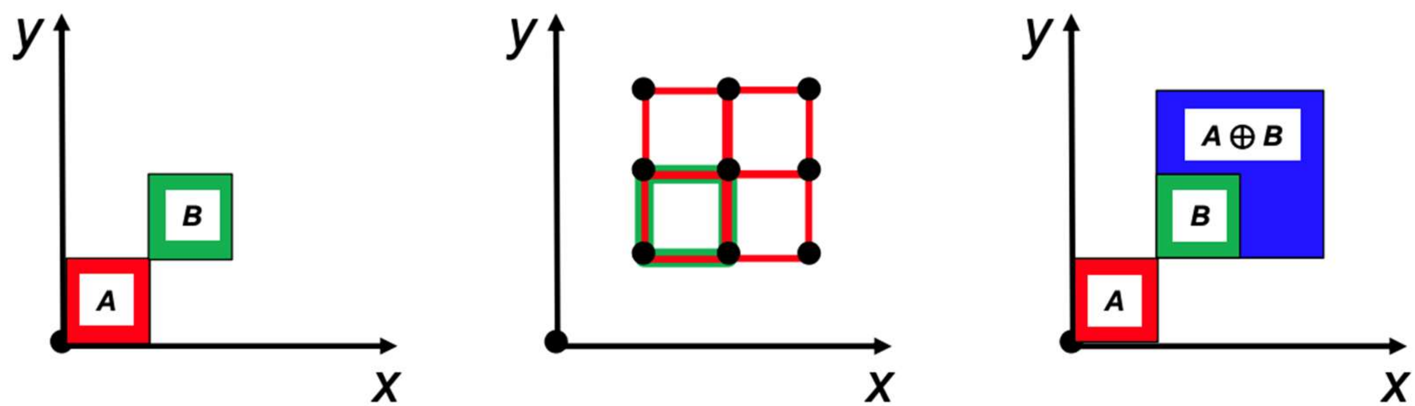


Piano Movers' problem

Notes courtesy of Siddhartha Srinivasa

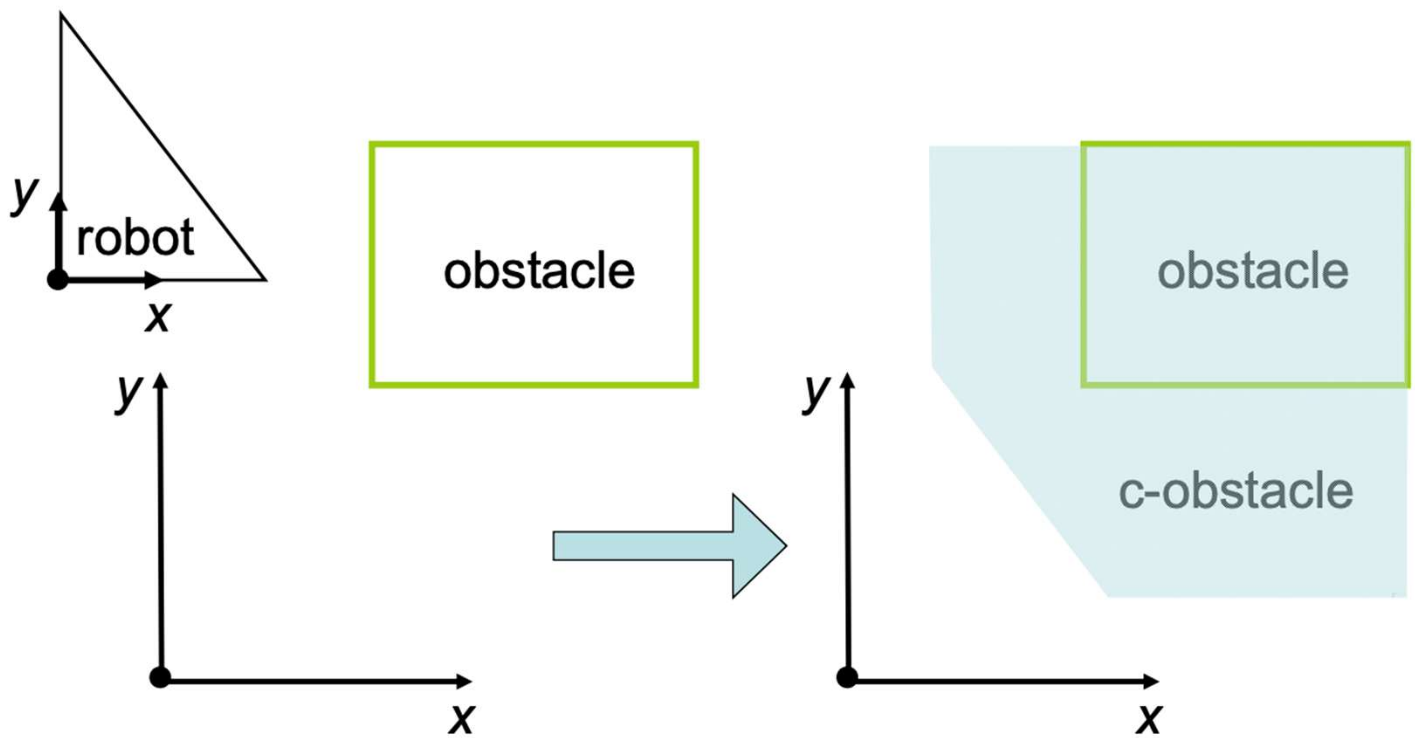
Minkowsky Sum

- Given two sets $A,B \in R^d$, their *Minkowski sum*, denoted $A \oplus B$, is the set $\{ a + b \mid a \in A, b \in B \}$
  - Result of *adding* each element of A to each element of B
- If A & B convex, just add vertices & find convex hull:



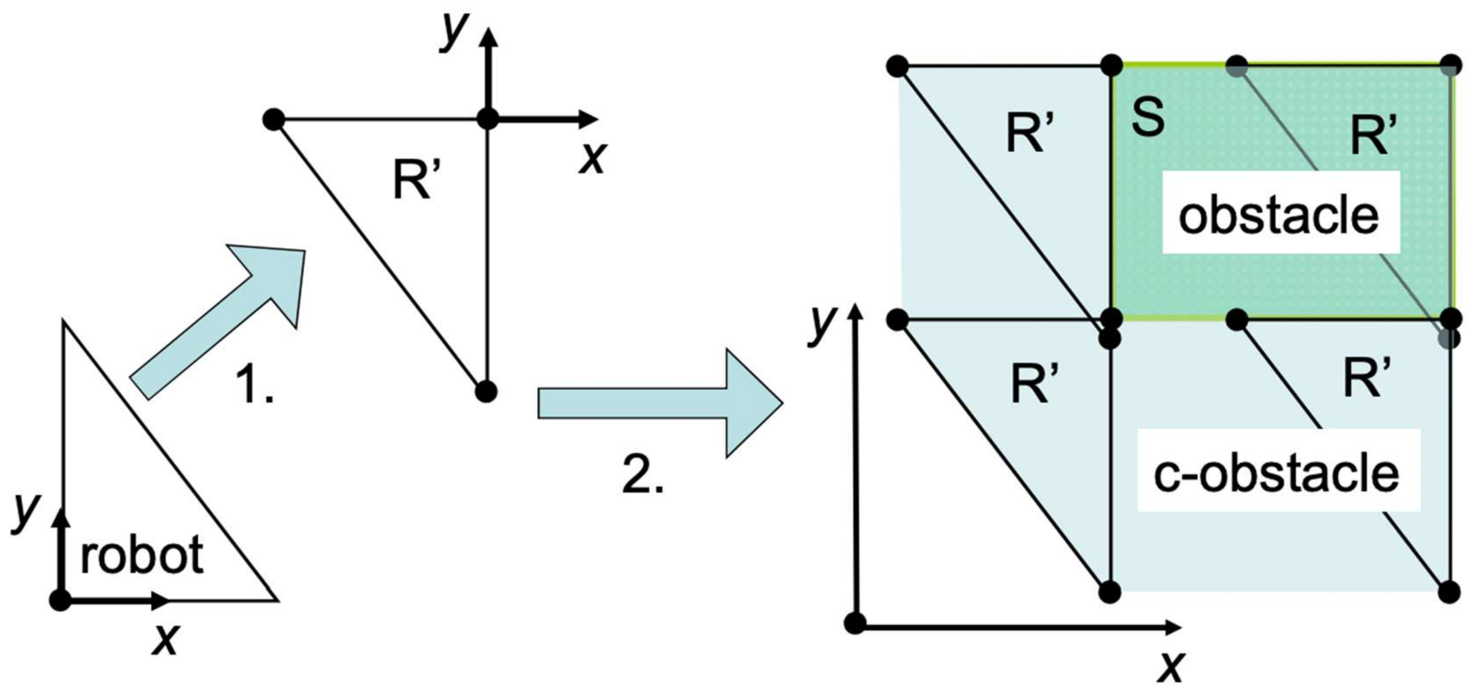
Computation of C-Obstacle: Minkowsky Difference

- Inputs: robot polygon R and obstacle polygon S
- Output: c-space obstacle c-obstacle(S, R)



Computation of C-Obstacle: Minkowsky Difference

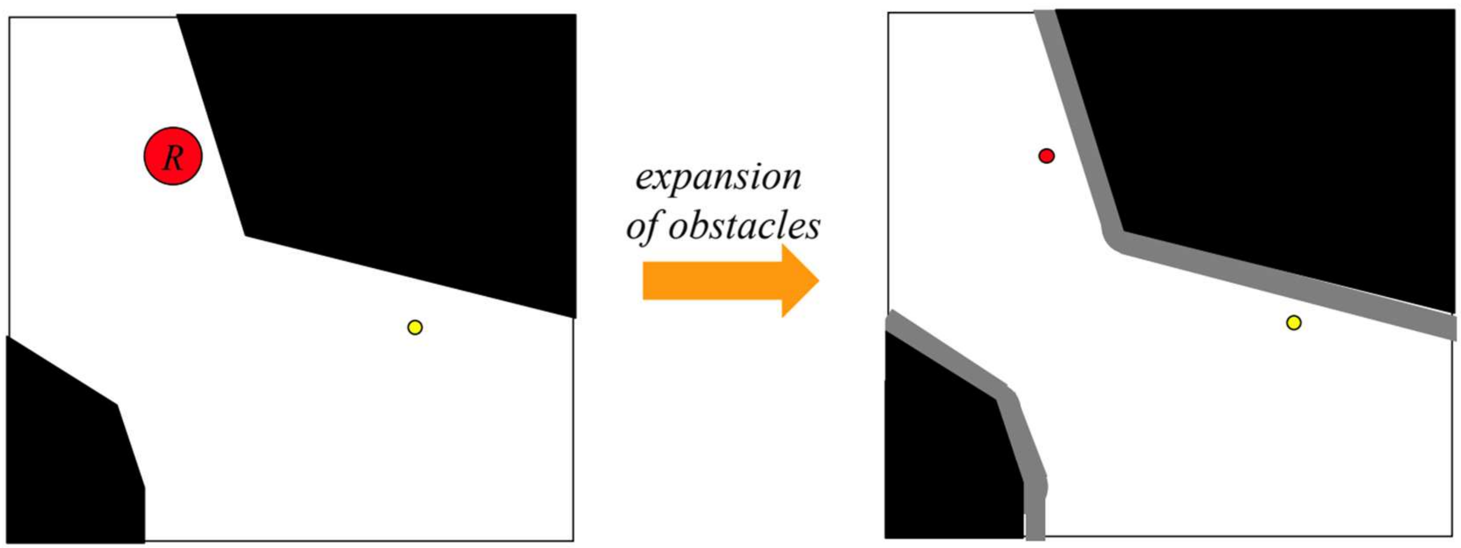
1. Reflect robot R about its origin to produce R'
2. Compute Minkowski sum of R' and obstacle S



Configuration Space: 2D base robot

Configuration space for a robot base in 2D world is:

- 2D if the robot is circular (symmetric in all directions)

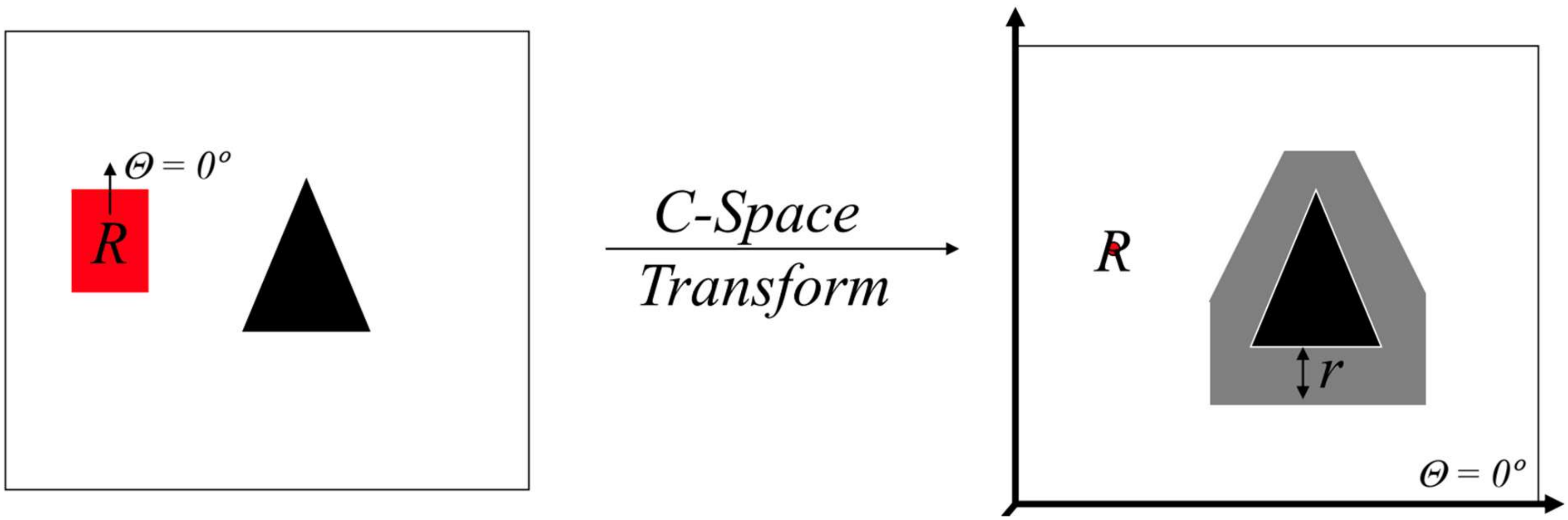


Is it necessary to build c-space obstacles?

Configuration Space: 2D base robot

Configuration space for a robot base in 2D world is:

- 3D if the robot is asymmetric

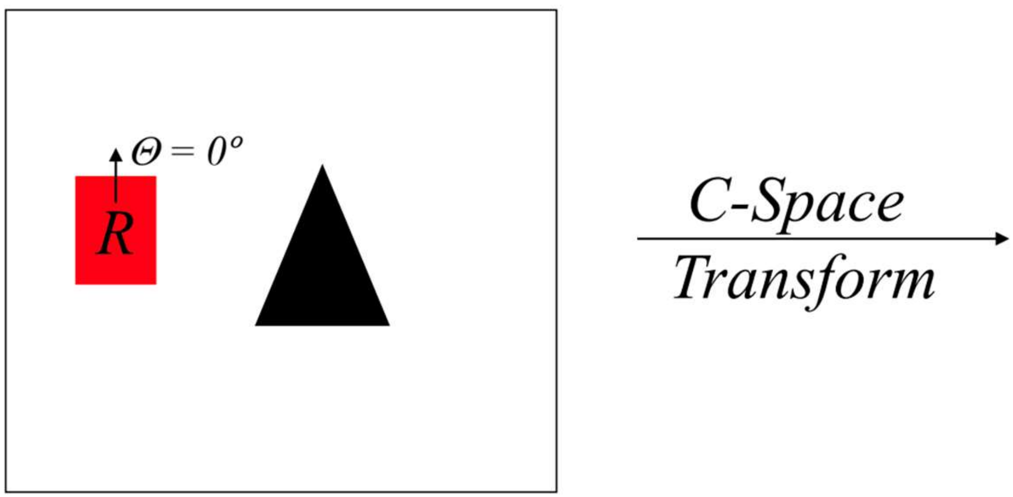


Difficult to build in real-time! What do we do?

Configuration Space: 2D base robot

Configuration space for a robot base in 2D world is:

- 3D if the robot is asymmetric



Earlier methods like potential fields

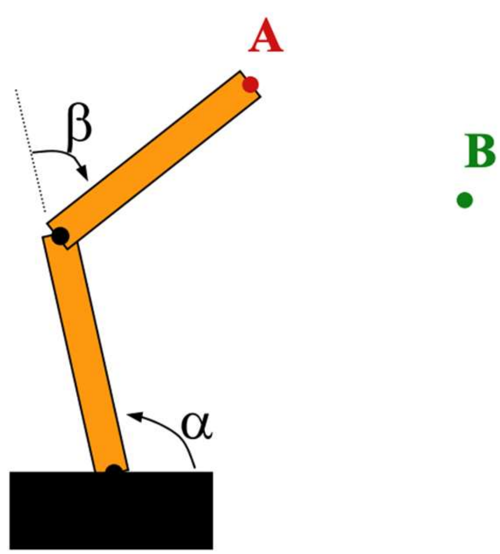
- Is it necessary to reason about c-space obstacles?

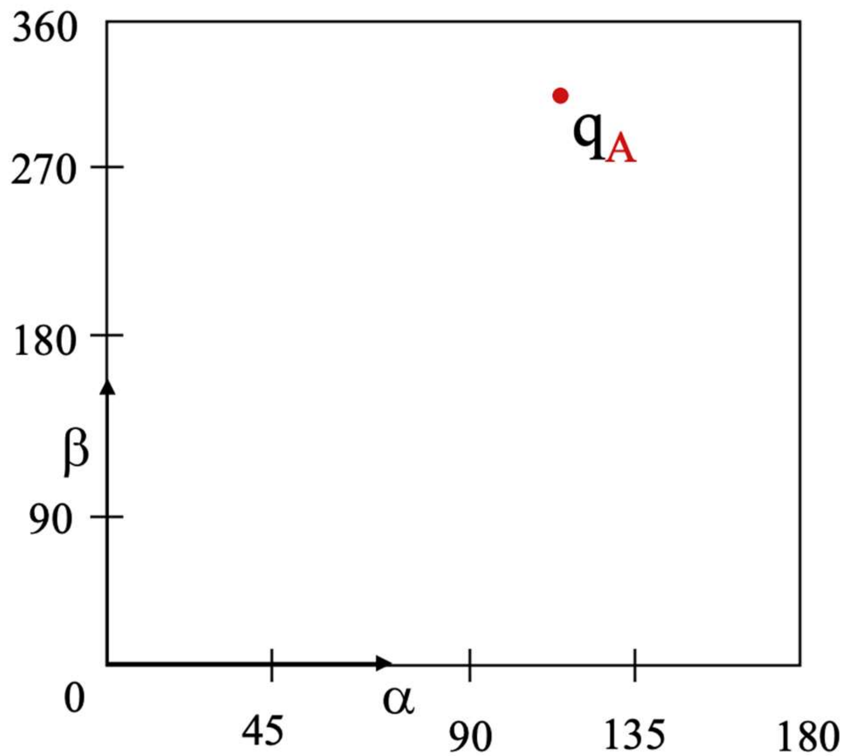
- Collision checking independent of setup of planner

Difficult to build in real-time! What do we do?

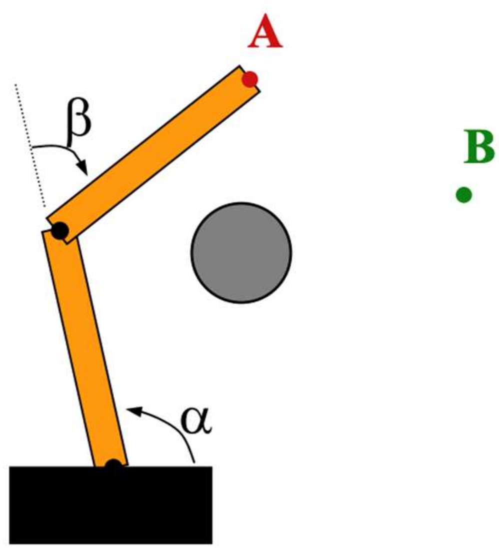
Configuration Space: Manipulator

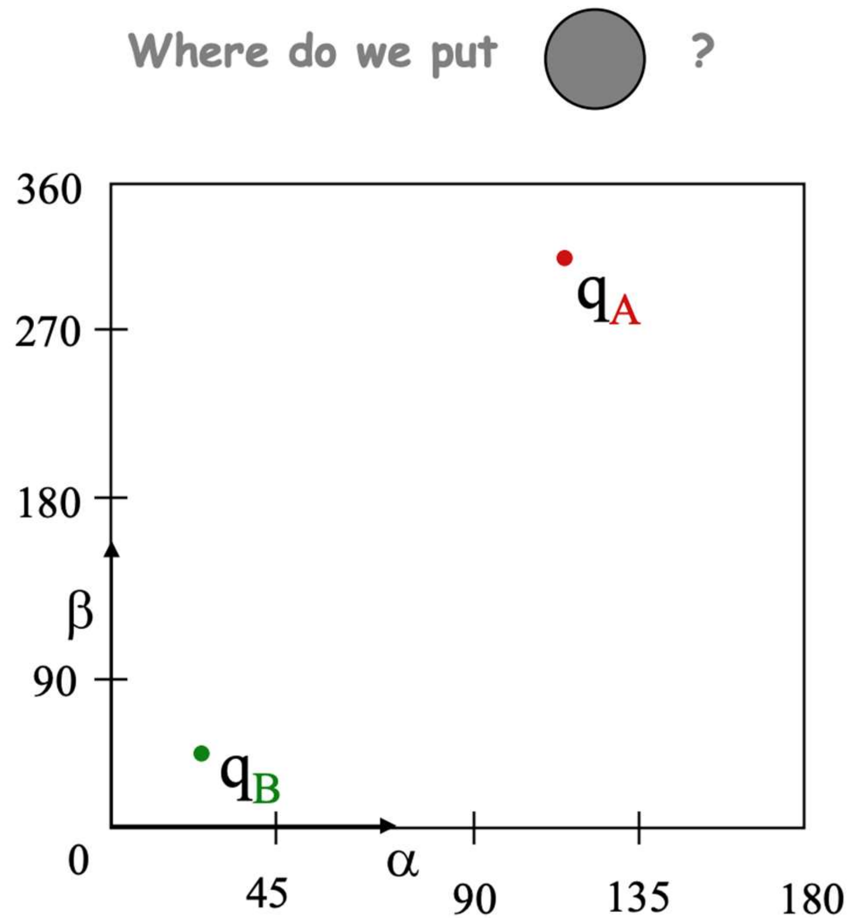
What is the C-Space?





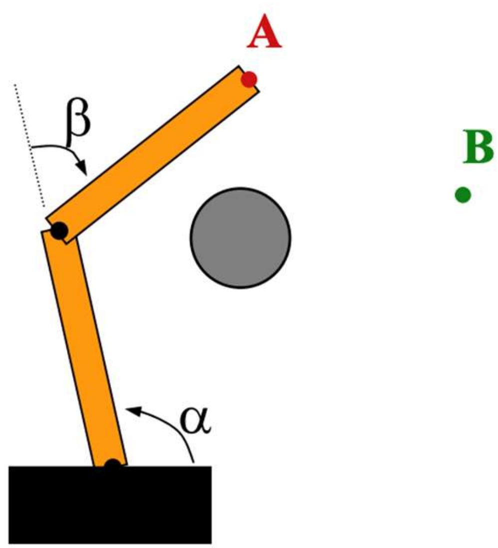
Configuration Space: Manipulator



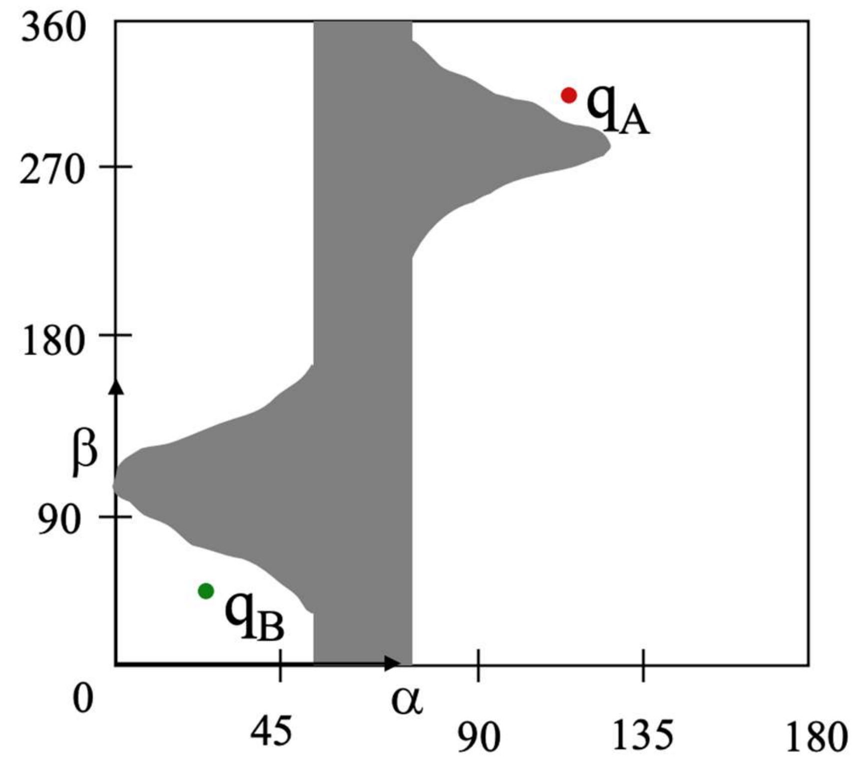


Configuration Space: Manipulator

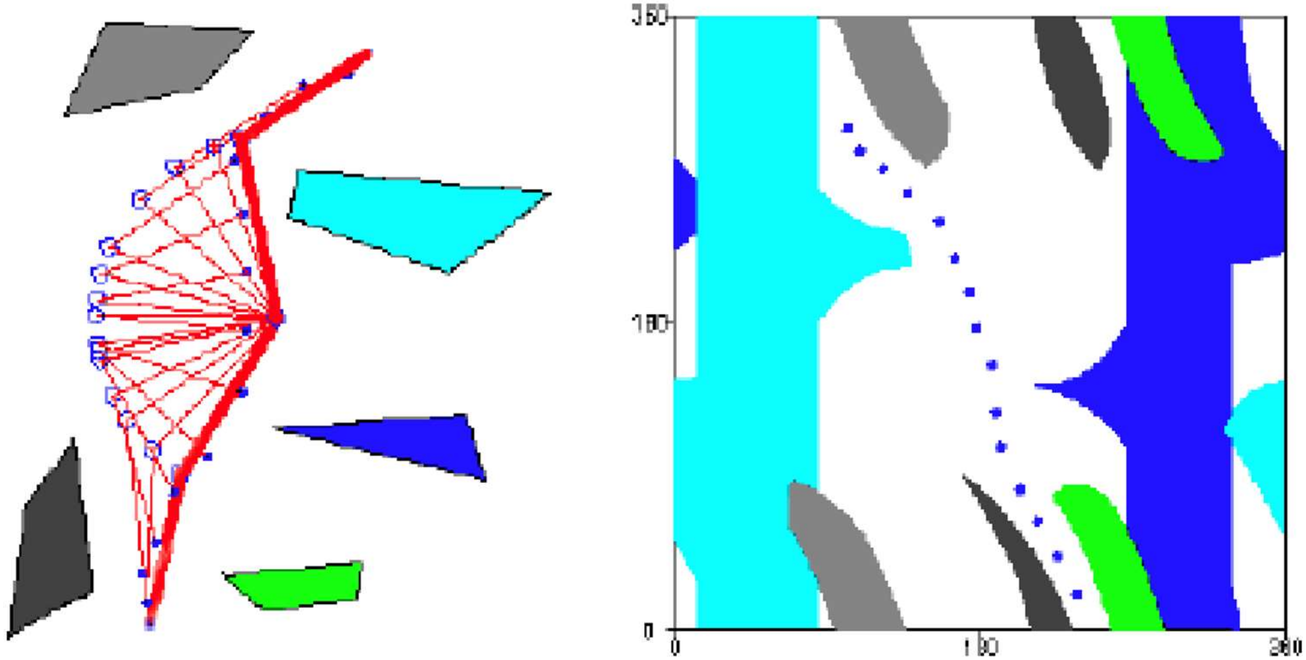
Reference *configuration*



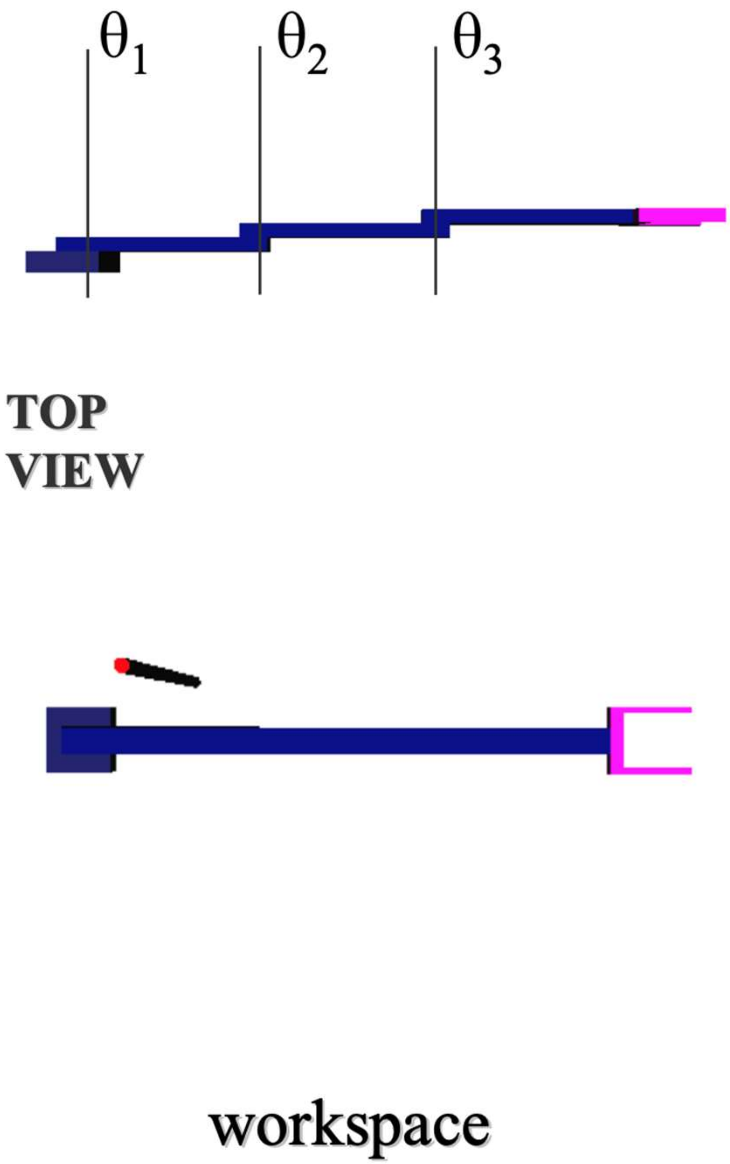
How do we get from A to B ?

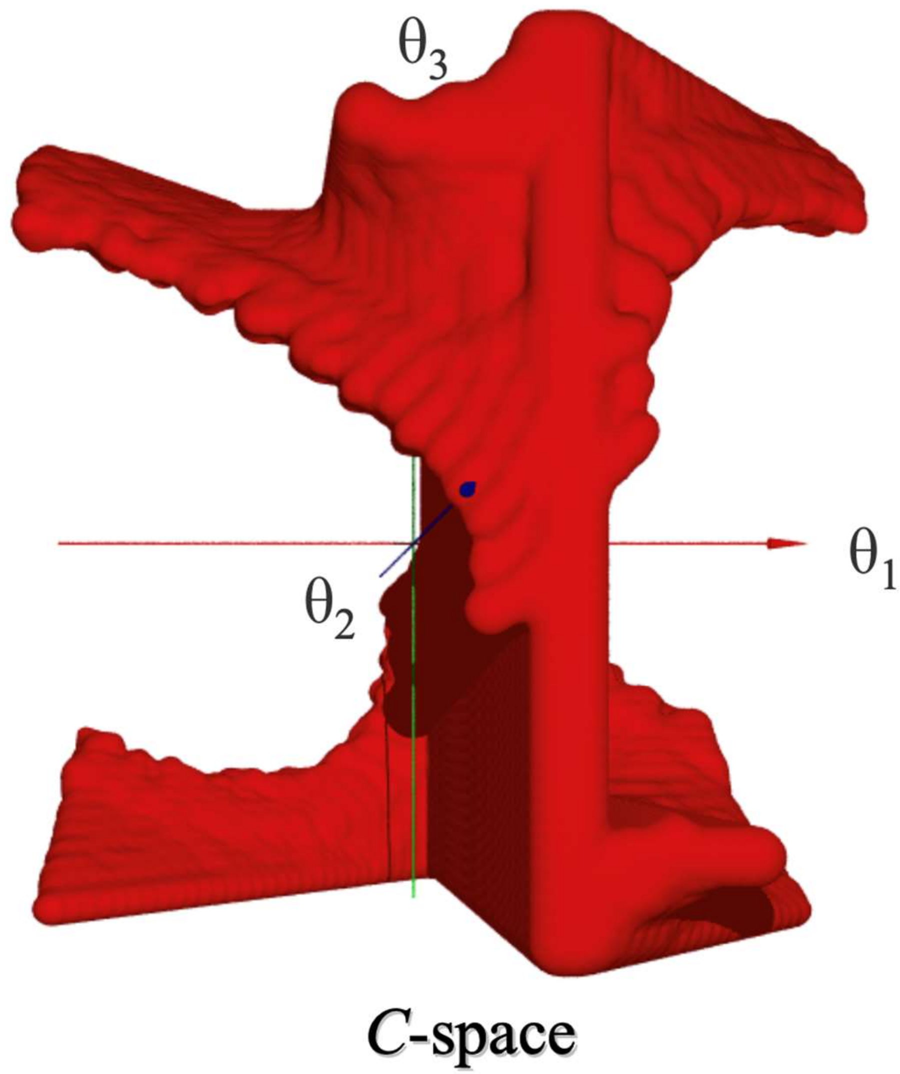


Configuration Space: Manipulator



Configuration Space: Manipulator





Configuration Space

A configuration is legal if

- it is not in collision
- is valid (within limits)

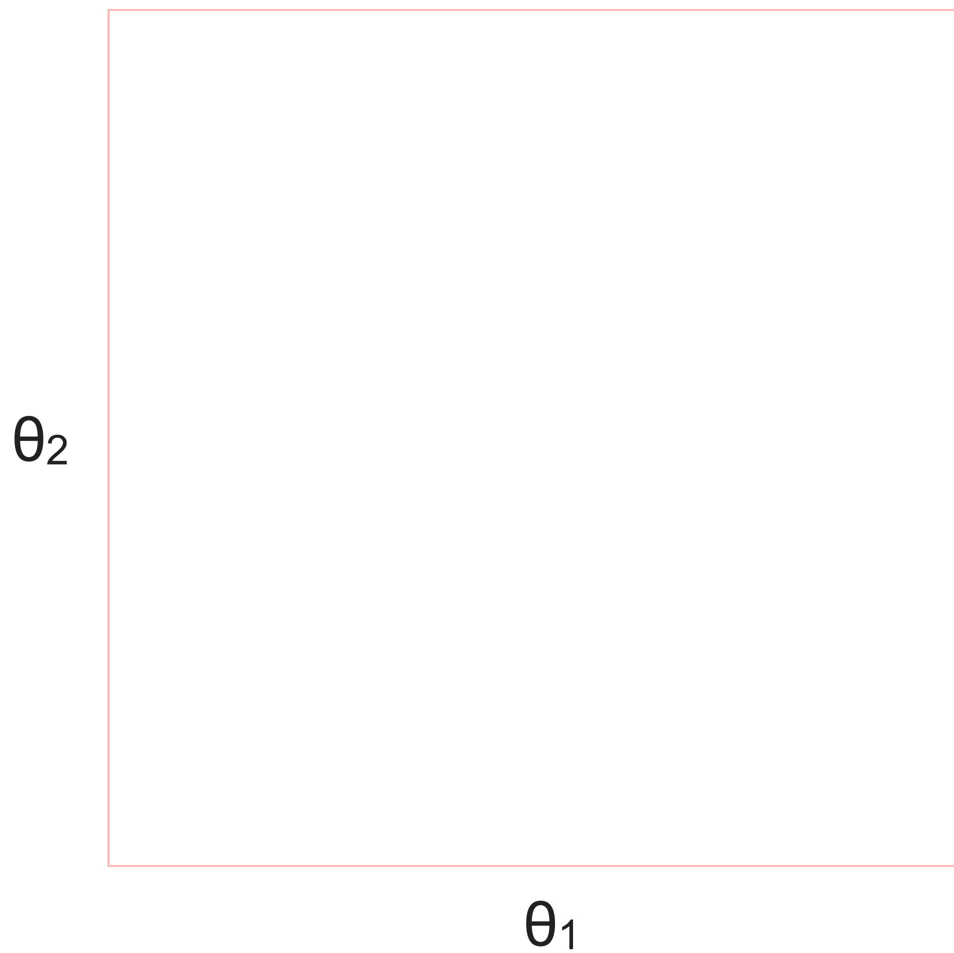
A configuration space is the set of legal configurations

Each point in the space -> configuration

Obstacle representation is non-trivial - legality of configuration is determined when necessary

How do we plan in this continuous space?

Configuration Space



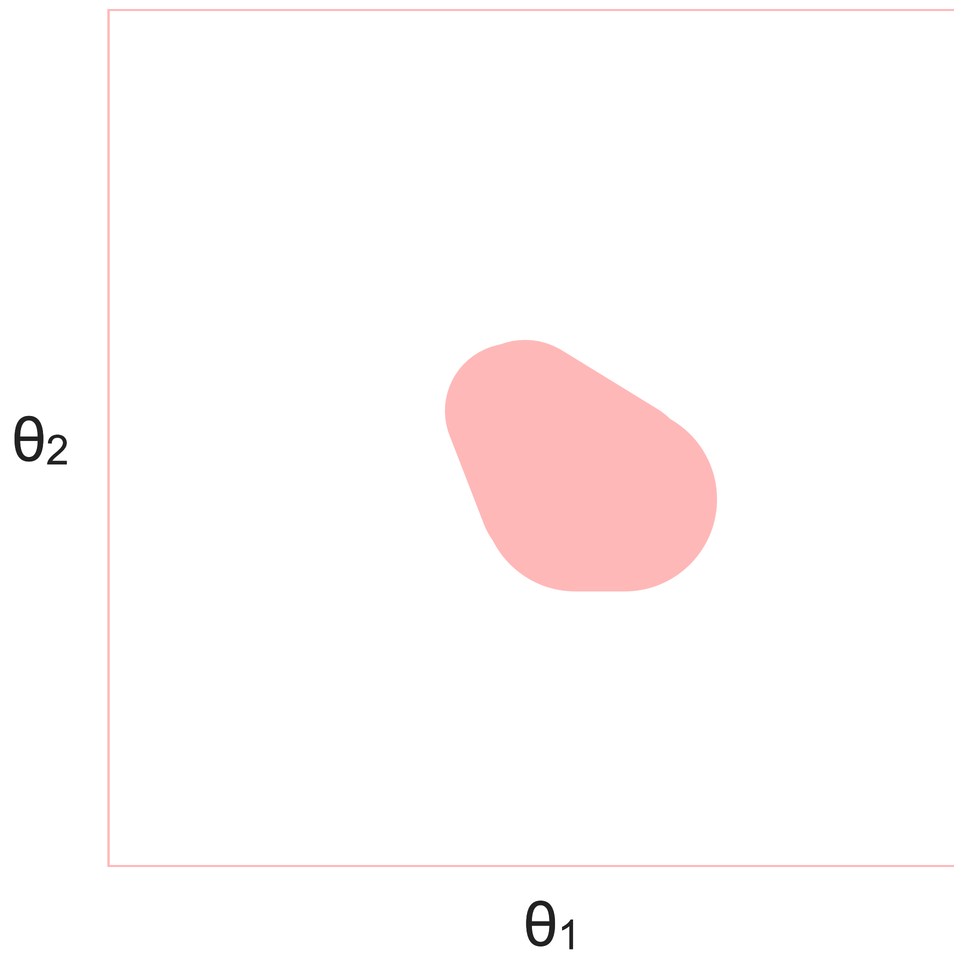
Motion Planning Algorithms

Graph-Based Methods

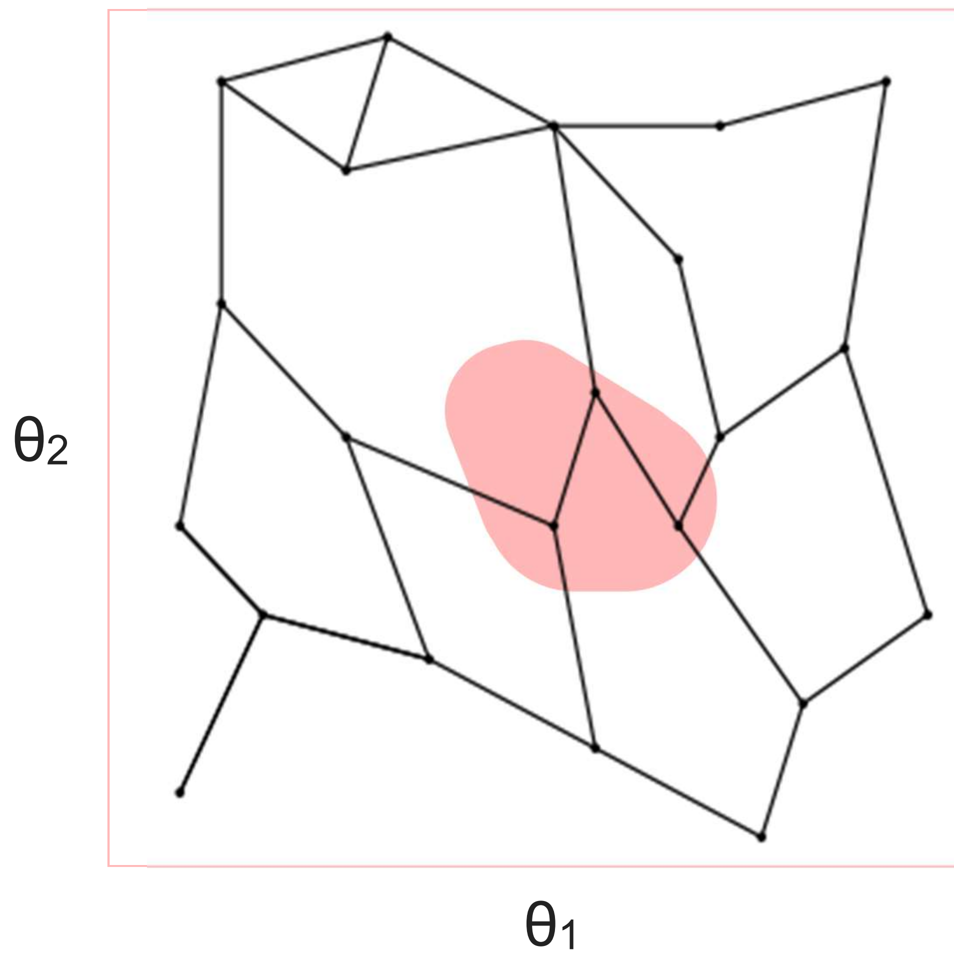
Sampling-Based Methods

Trajectory Optimization

Planning in continuous space



Planning as Graph Search



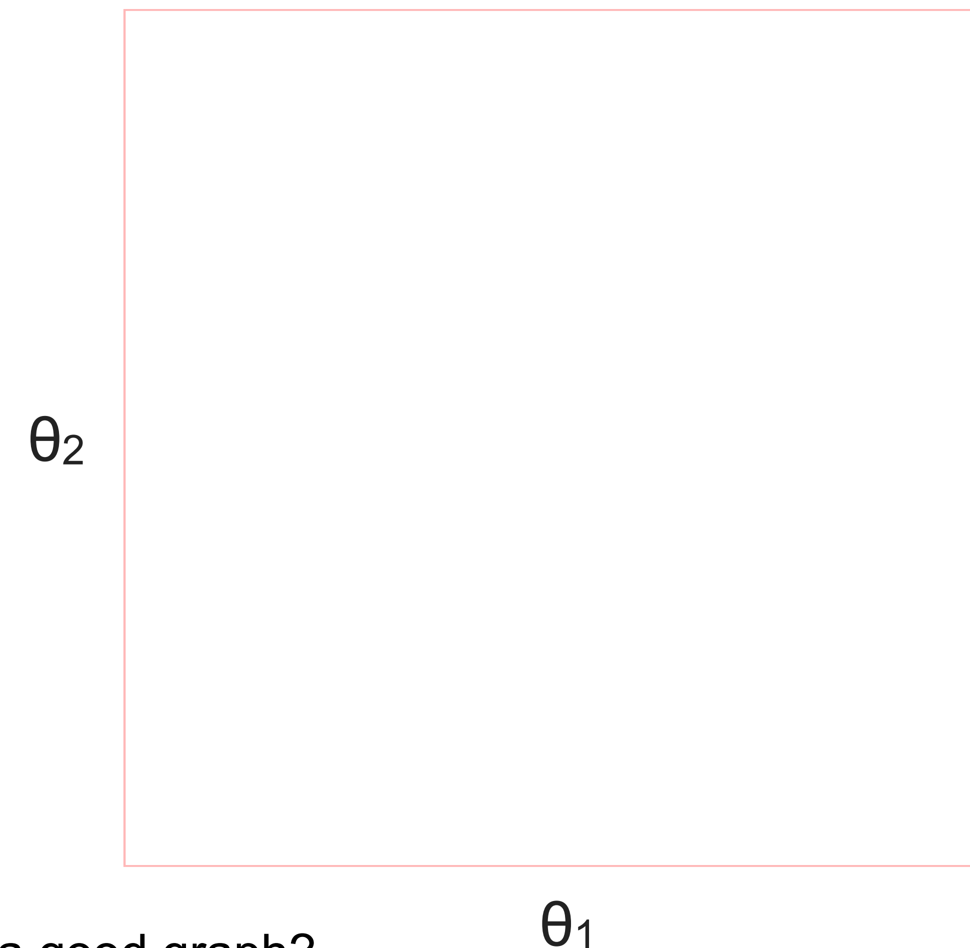
Graph Representations

Skeletonization

1. Visibility Graphs
2. Voronoi Diagrams
3. Probabilistic Roadmaps

Cell Decomposition

1. X-Connected Grid
2. Lattice-Based Graphs



Properties of a good graph?

Connectivity

Coverage

What other characteristics define a good graph?

Graph Representations

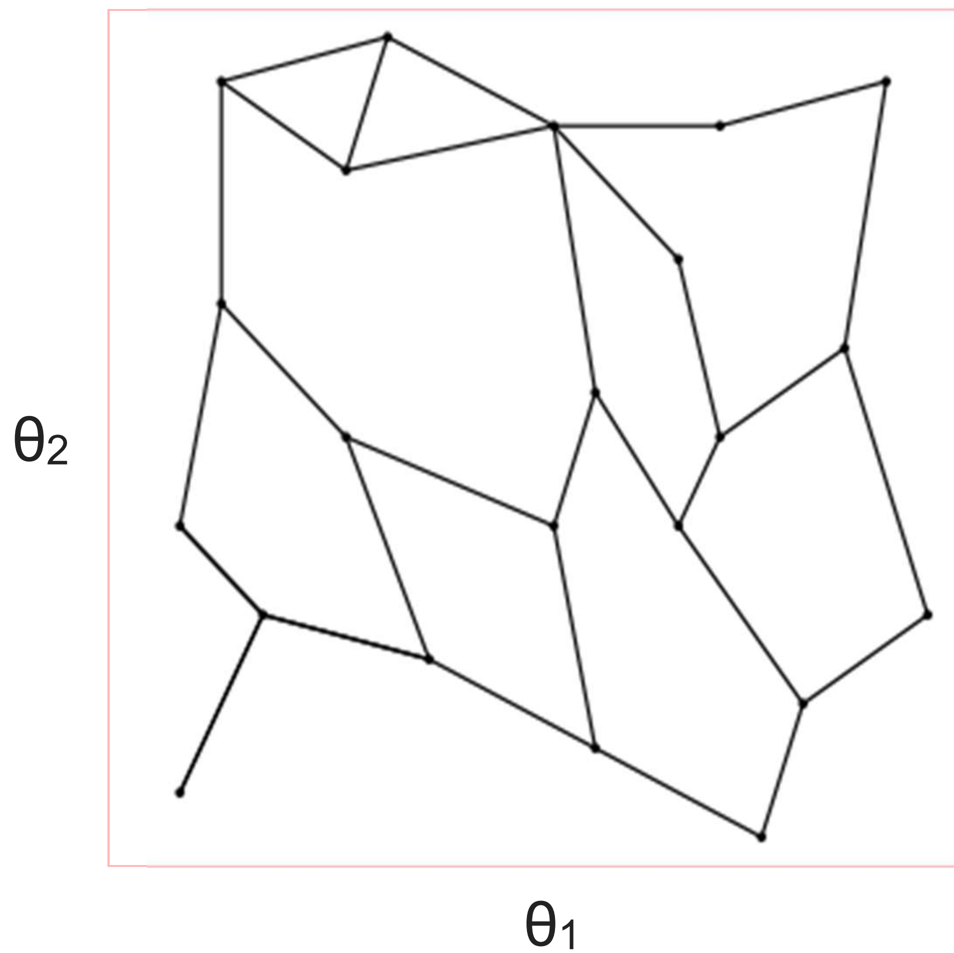
Skeletonization

1. Visibility Graphs
2. Voronoi Diagrams
3. Probabilistic Roadmaps

Cell Decomposition

1. X-Connected Grid
2. Lattice-Based Graphs

1. What is the sampling strategy?
2. How do we connect vertices?
3. Pros and Cons?
4. Explicit or Implicit? – See Notes



Graph Representations

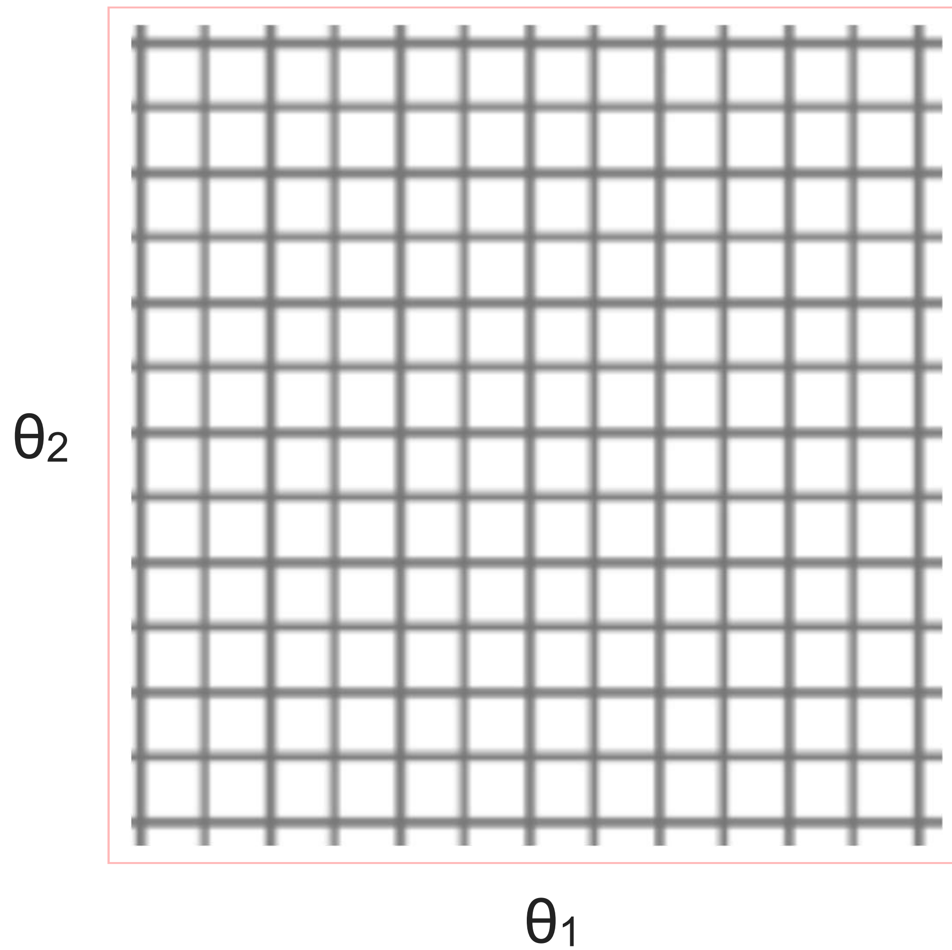
Skeletonization

1. Visibility Graphs
2. Voronoi Diagrams
3. Probabilistic Roadmaps

Cell Decomposition

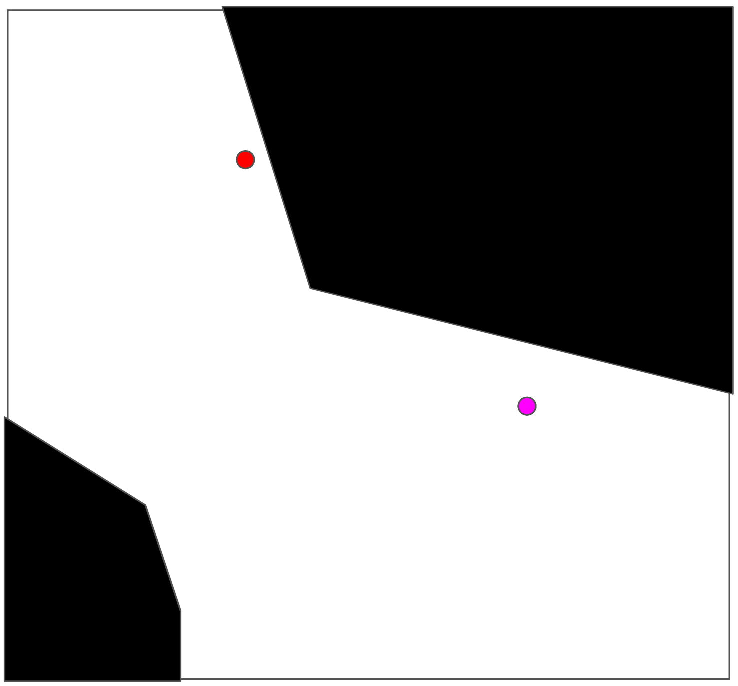
1. X-Connected Grid
2. Lattice-Based Graphs

1. What should X be?
2. What are the pros and cons?
3. Explicit or Implicit?

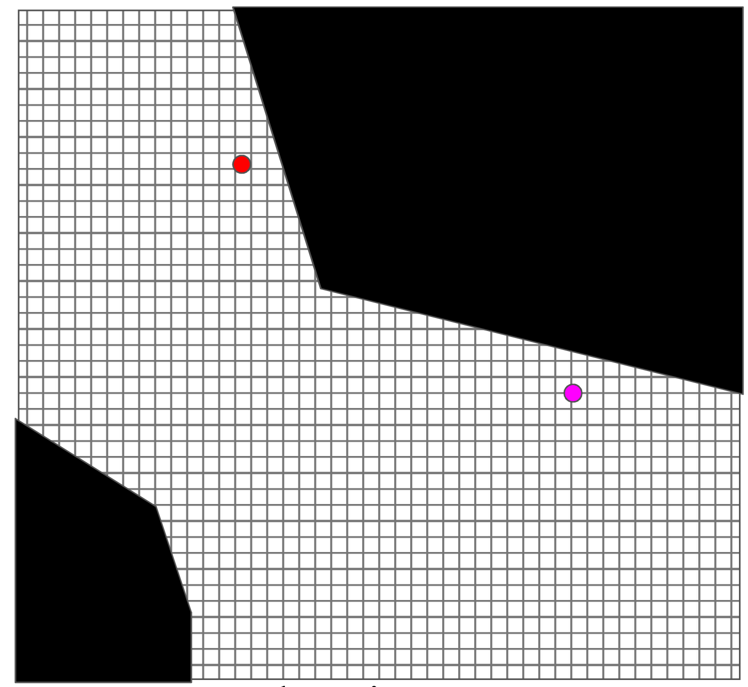


Planning via Cell Decomposition

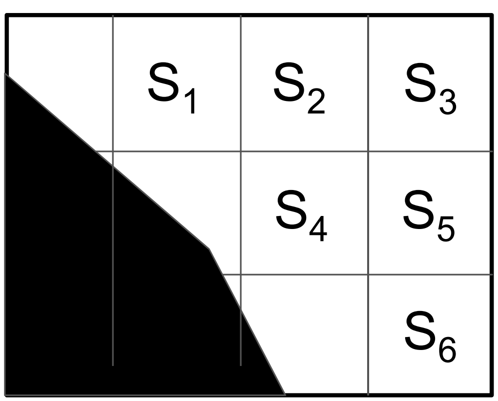
- Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path



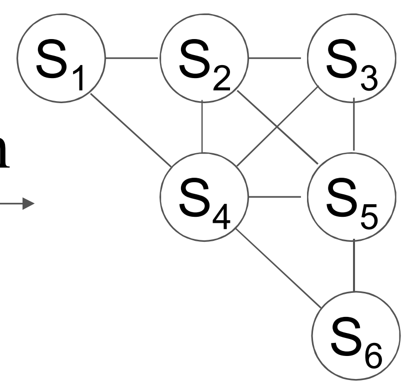
discretize



planning map



convert into a graph



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

Graph Representations

Skeletonization

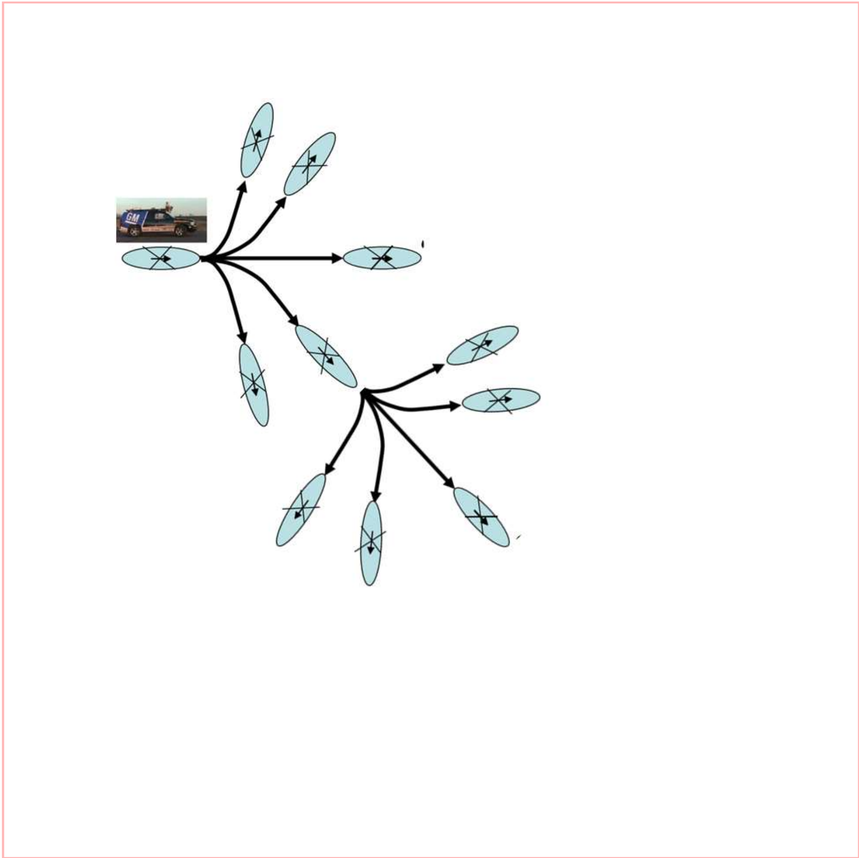
1. Visibility Graphs
2. Voronoi Diagrams
3. Probabilistic Roadmaps

Cell Decomposition

1. X-Connected Grid
2. Lattice-Based Graphs

1. How are motion primitives defined?
2. How are cost of edges determined?
3. Explicit or Implicit?

θ_2

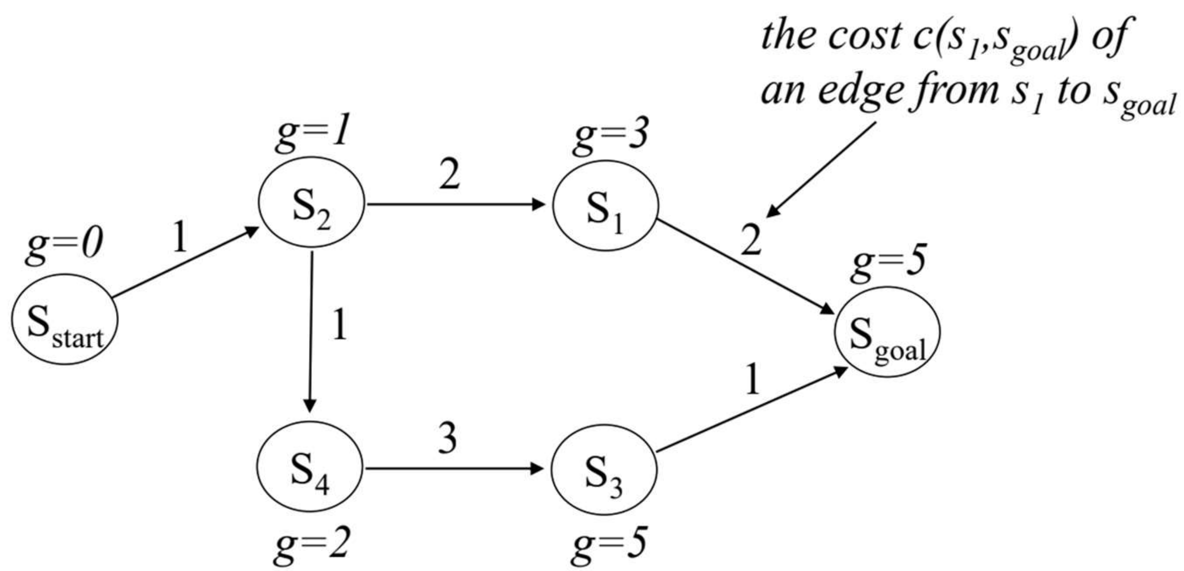


θ_1

Search for Least-Cost Path

Many searches work by computing optimal g-values for relevant states

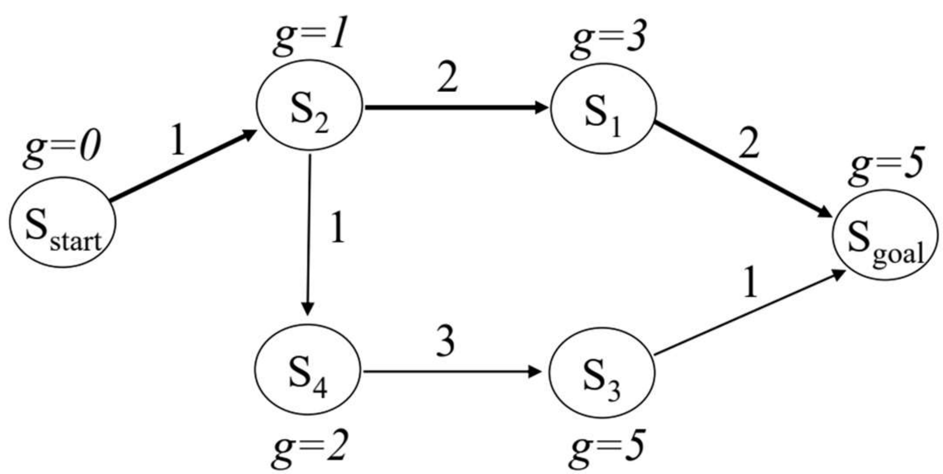
- $g(s)$ – an estimate of the cost of a least-cost path from s_{start} to s
- optimal values satisfy: $g(s) = \min_{s'' \in pred(s)} g(s'') + c(s'',s)$ Why?



Search for Least-Cost Path

Least-cost path is a greedy path computed by backtracking:

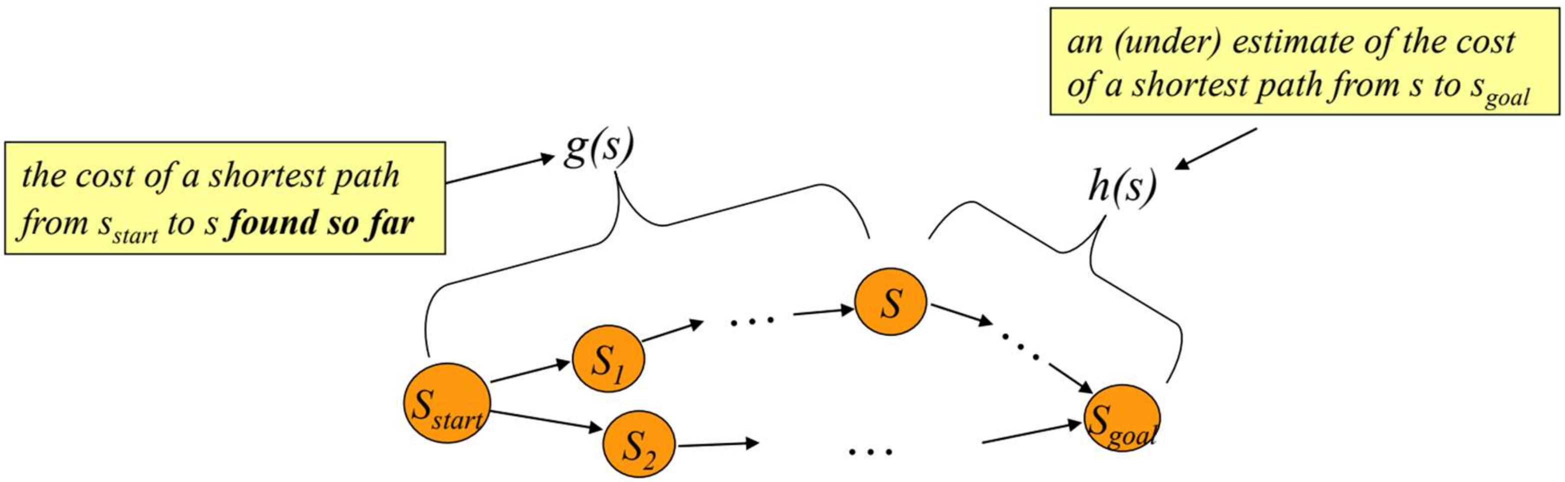
– start with s_{goal} and from any state s move to the predecessor state s' such that $$s' = \arg\min_{s'' \in pred(s)} (g(s'') + c(s'', s))$$



A* Search

- Computes optimal g-values for relevant states

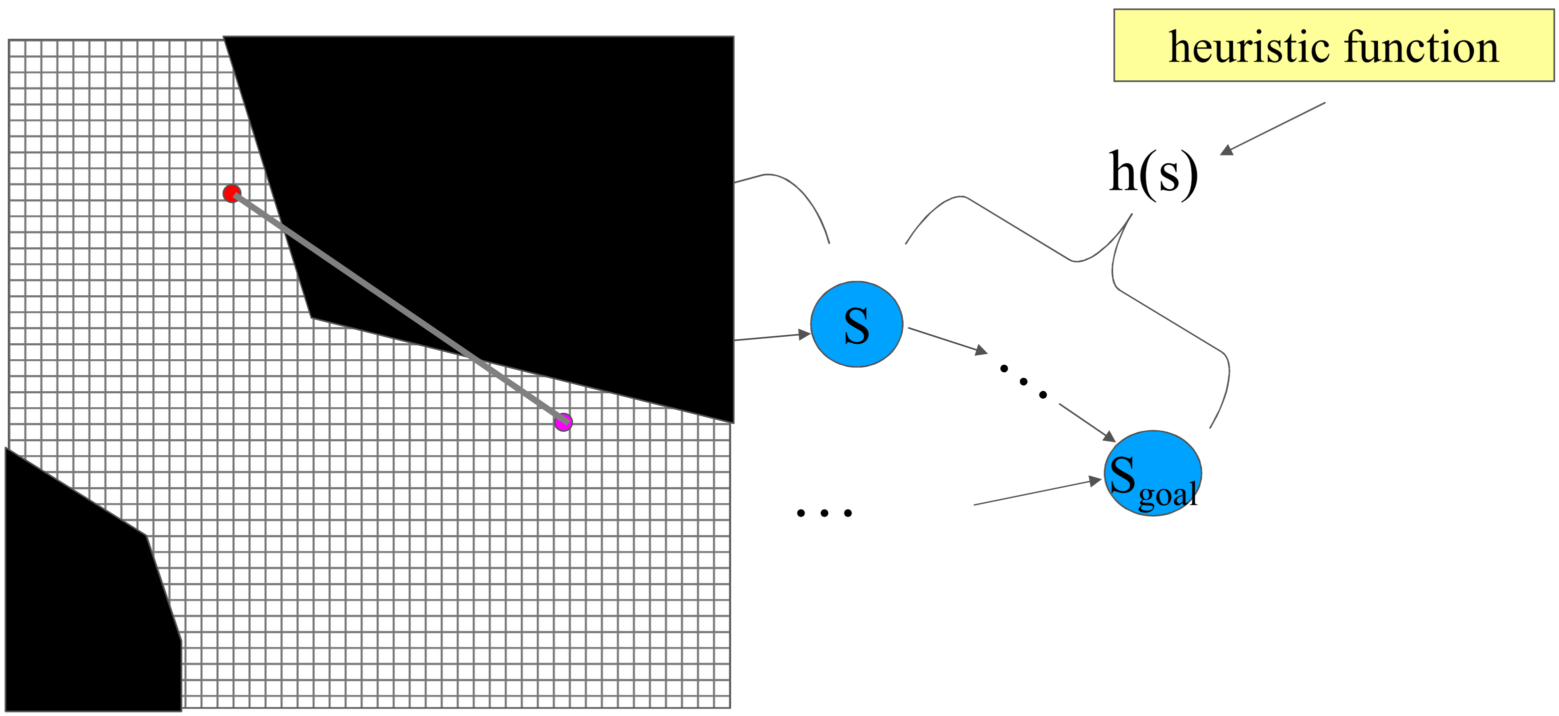
at any point of time:



A* Search

Computes optimal g-values for relevant states

at any point of time:



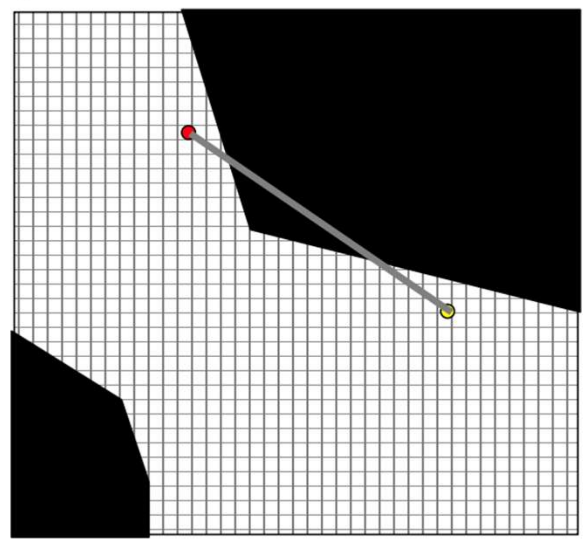
one popular heuristic function – Euclidean distance

A* Search

minimal cost from s to s_{goal}

- Heuristic function must be:
  - admissible: for every state s, $h(s) \leq c^*(s, s_{goal})$
  - consistent (satisfy triangle inequality):

    $h(s_{goal}, s_{goal}) = 0$ *and for every* $s \neq s_{goal}$, $h(s) \leq c(s, succ(s)) + h(succ(s))$
  - admissibility <u>provably</u> follows from consistency and often (<u>not always</u>) consistency follows from admissibility



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

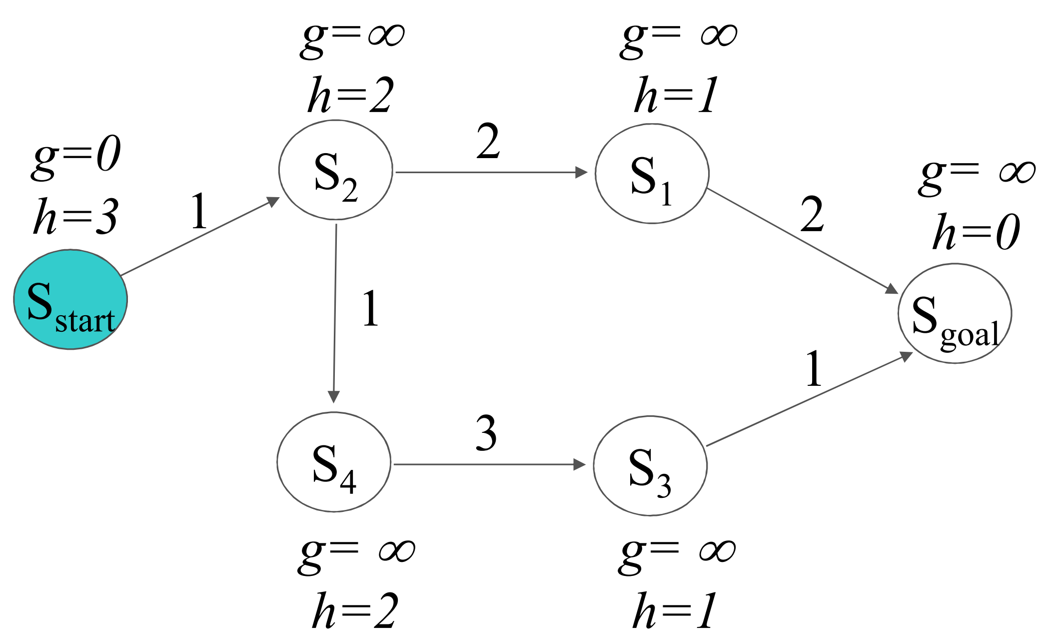
      $g(s') = g(s) + c(s,s')$;

      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*;

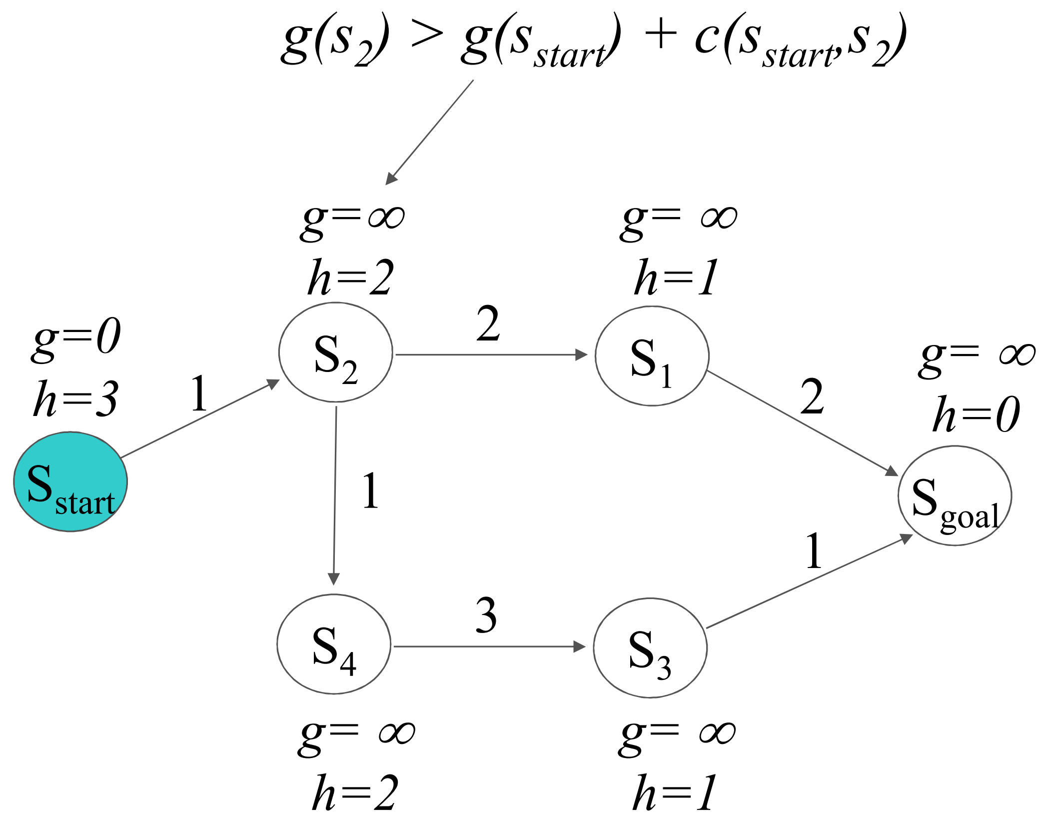
  for every successor s' of s such that s' not in *CLOSED*

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

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

      insert s' into *OPEN*;

$$CLOSED = \{\}$$
$$OPEN = \{s_{start}\}$$
$$\textit{next state to expand: } s_{start}$$



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

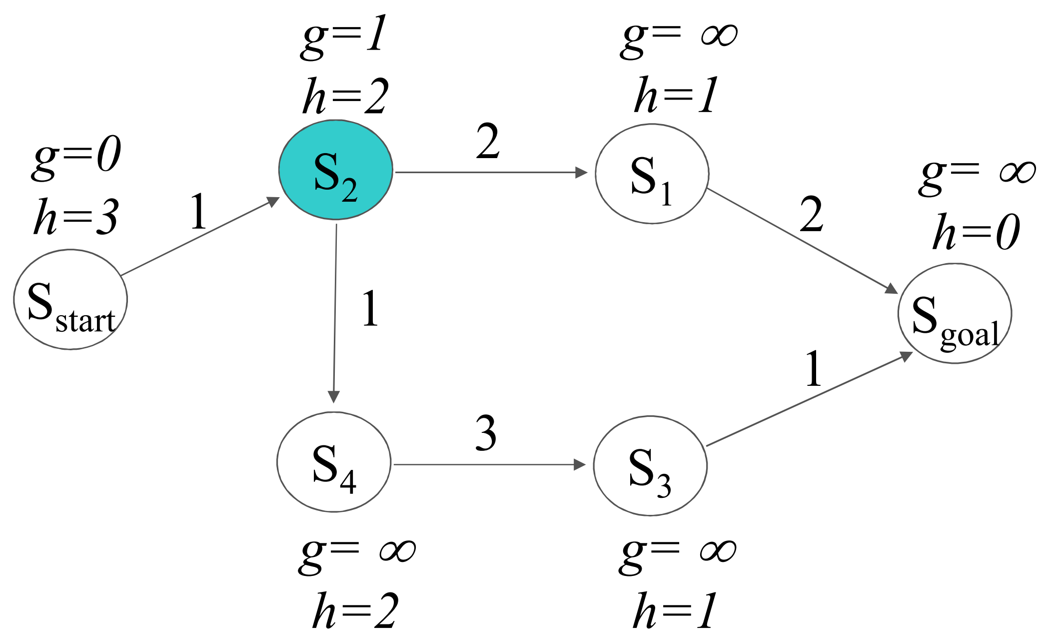
$g(s') = g(s) + c(s,s')$;

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 $CLOSED$;

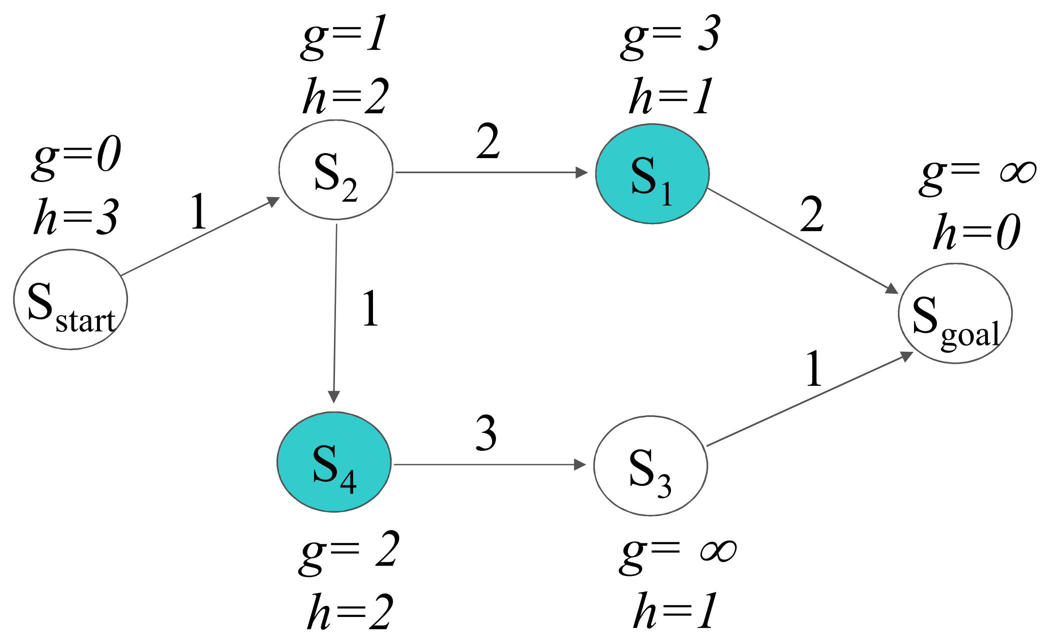
for every successor s' of s such that s' not in $CLOSED$

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

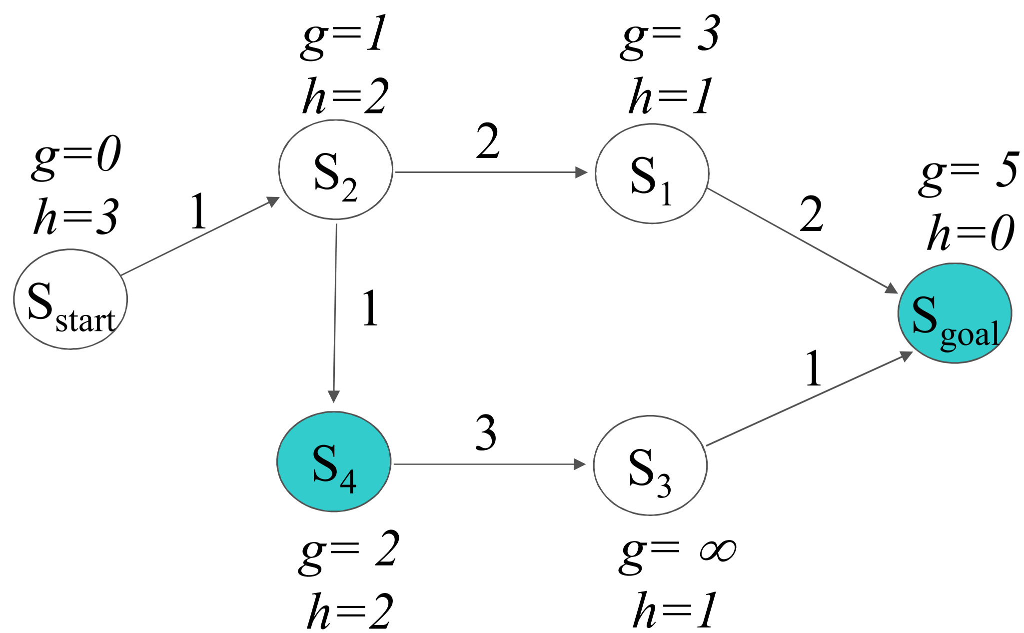
$g(s') = g(s) + c(s,s')$;

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

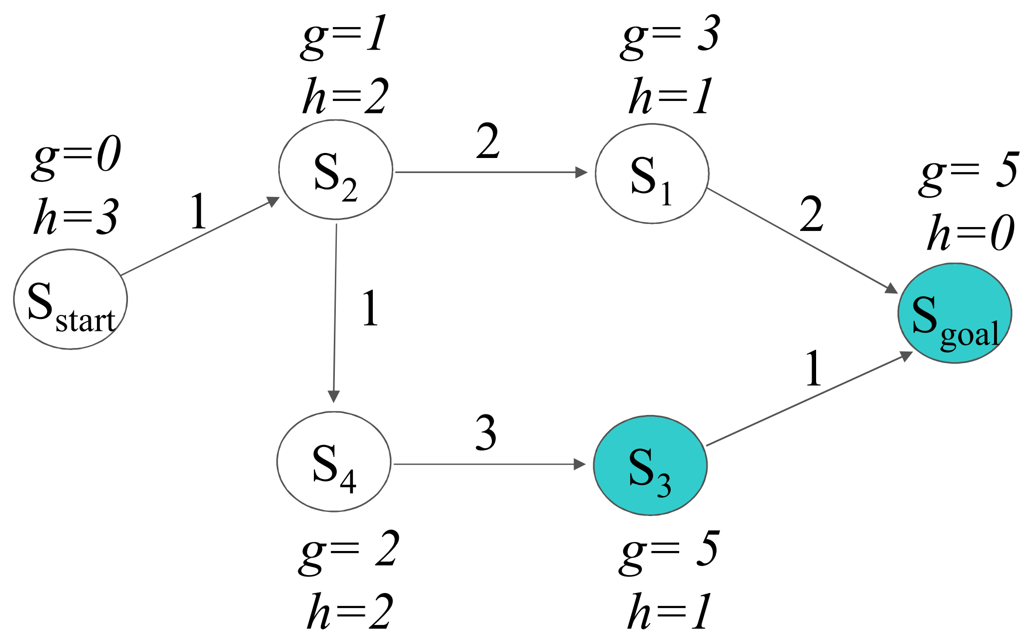
$g(s') = g(s) + c(s,s')$;

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 $CLOSED$

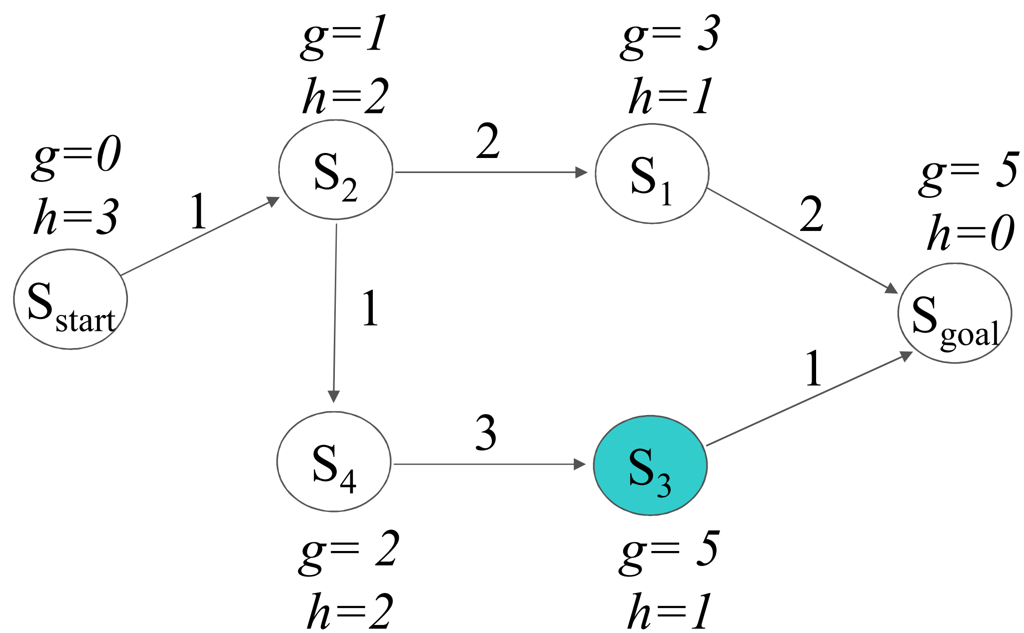
if $g(s') > g(s) + c(s,s')$

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

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*;

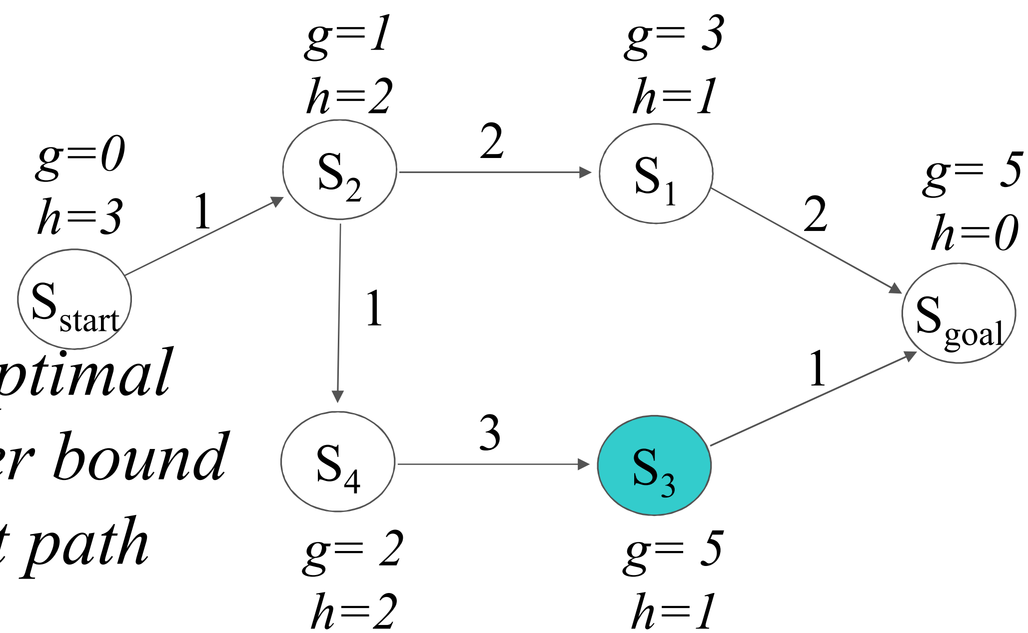
  insert s into *CLOSED*;

  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 $OPEN$;

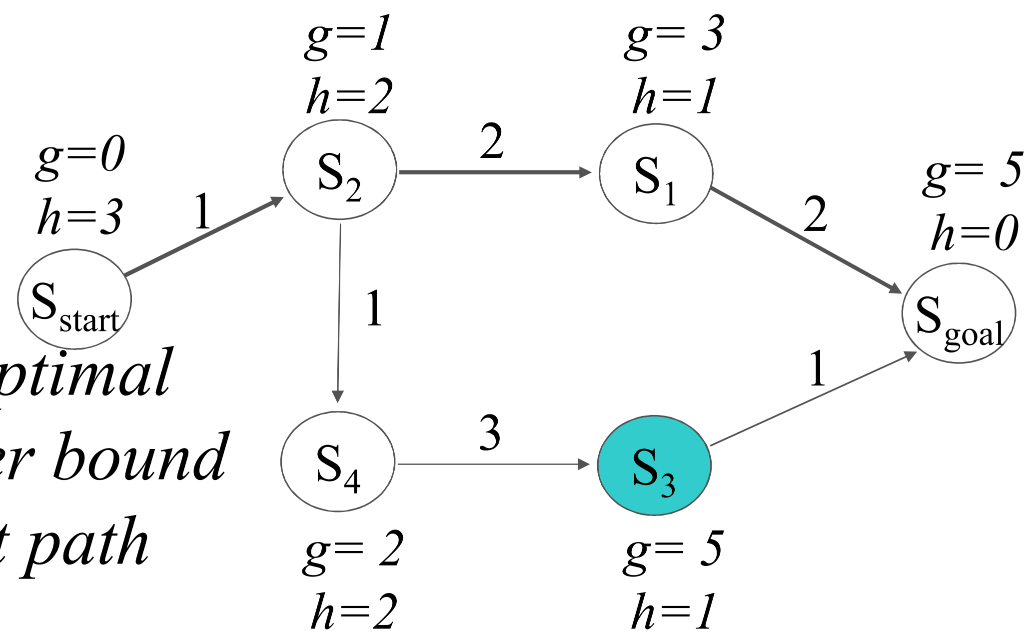
  insert s into $CLOSED$;

  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 $OPEN$;

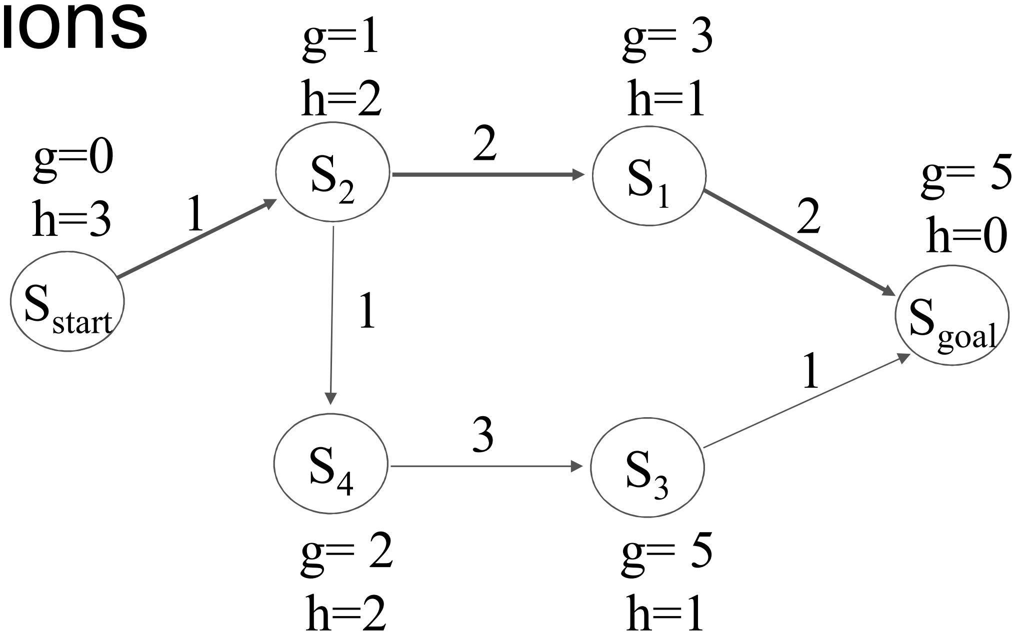


for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

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

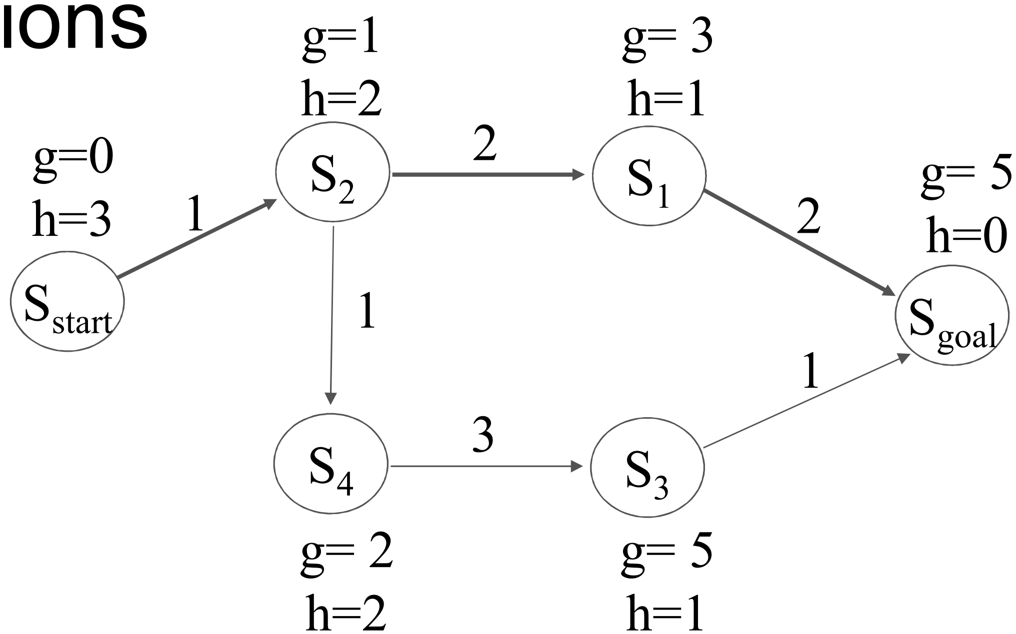


A* Search

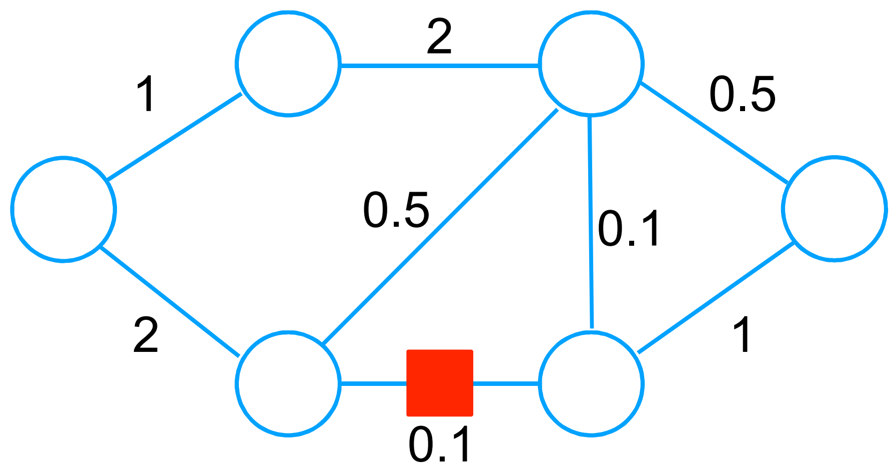
Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution

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



A* Search: More interesting example (Try in Class)



A* Search

1. Correctness
2. Completeness
3. Optimality

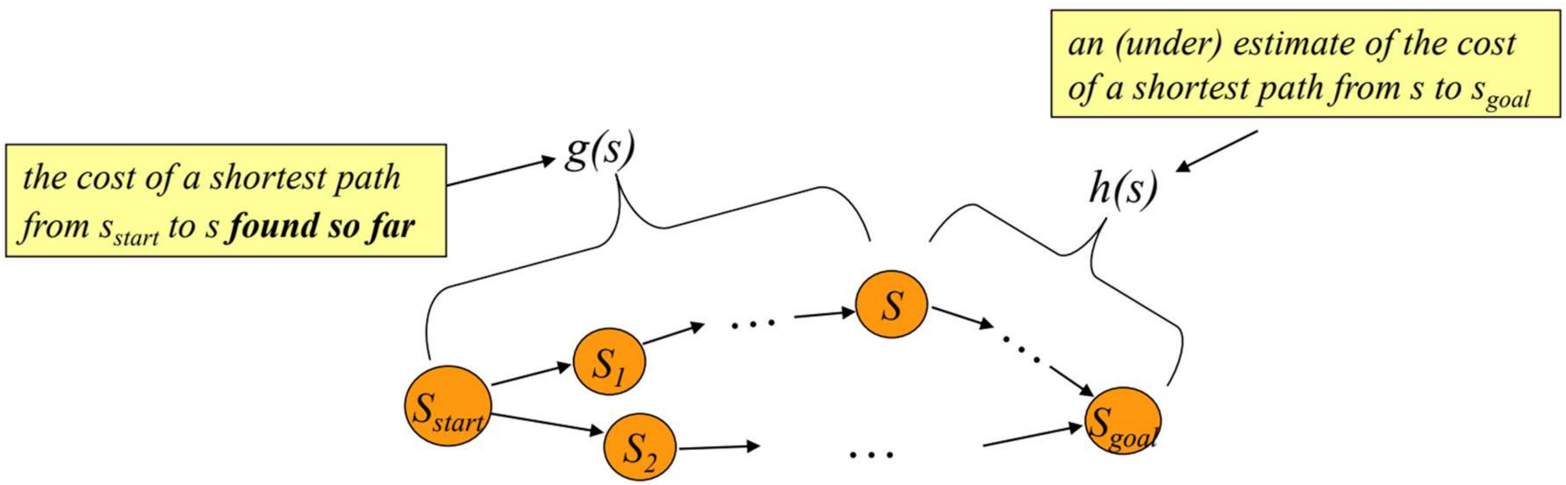
- Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution

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

Role of Heuristic

- A* Search: expands states in the order of $f = g+h$ values
- Dijkstra's: expands states in the order of $f = g$ values

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



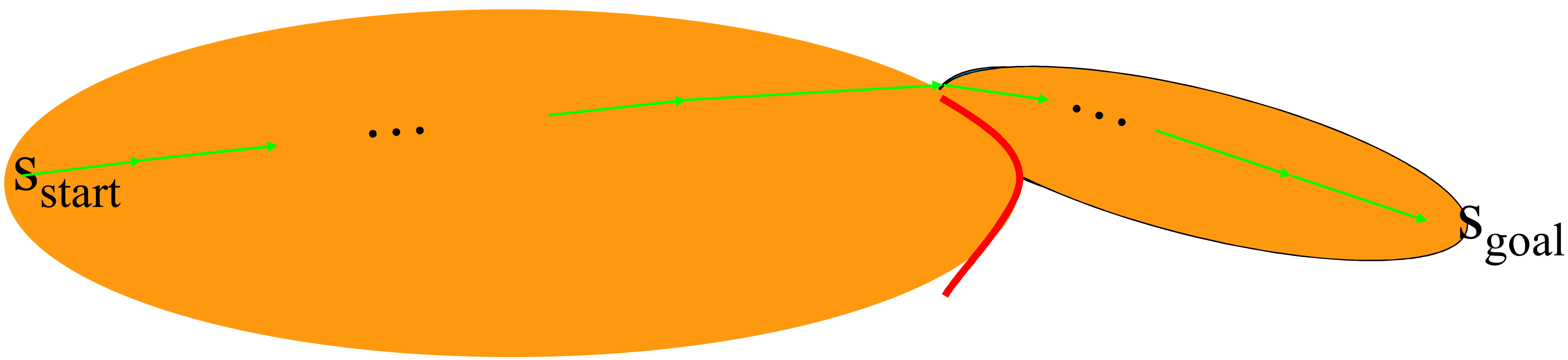
Role of Heuristic

- Dijkstra's: expands states in the order of $f = g$ values

What are the states expanded?

Effect of the Heuristic Function

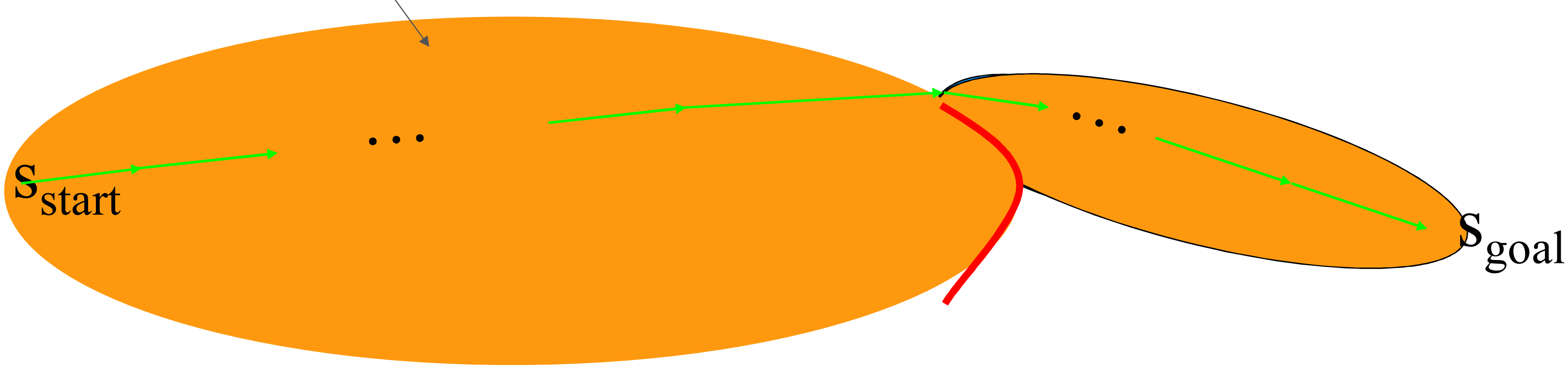
A* Search: expands states in the order of $f = g+h$ values



Effect of the Heuristic Function

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

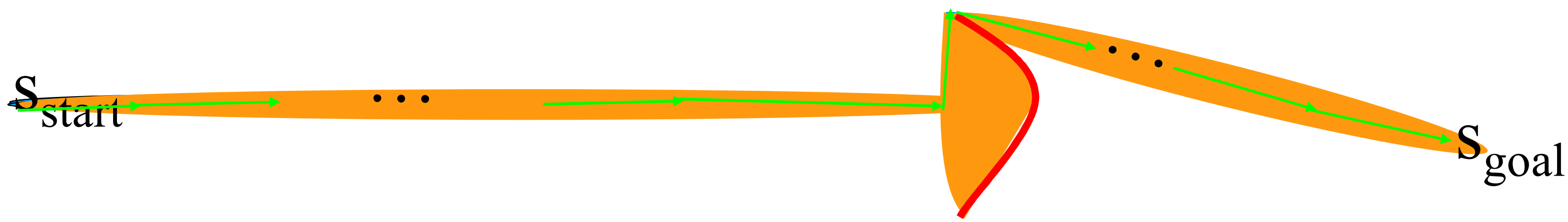
for large problems this results in A* quickly running out of memory (memory: O(n))



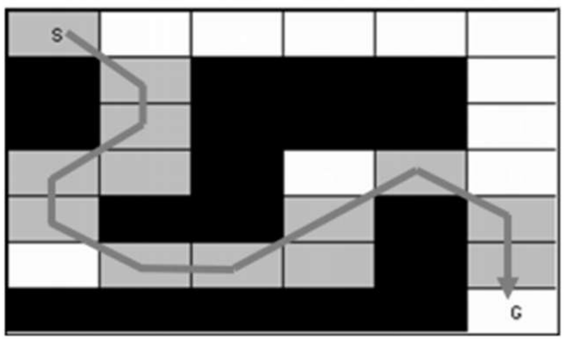
Effect of the Heuristic Function

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

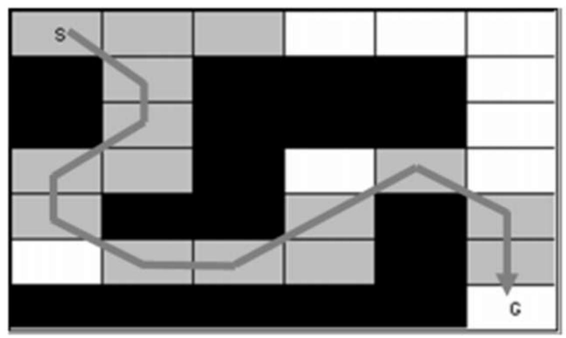
solution is always ε-suboptimal:
cost(solution) ≤ ε·cost(optimal solution)



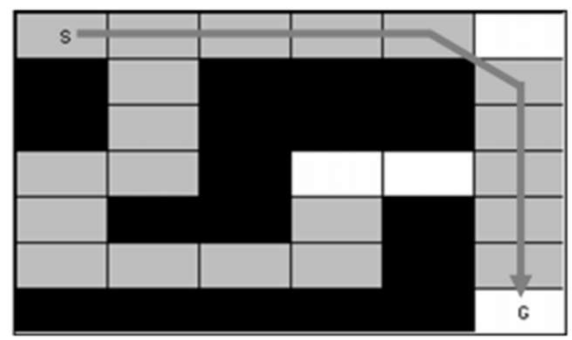
Effect of the Heuristic Function



$\epsilon = 2.5$



$\epsilon = 1.5$

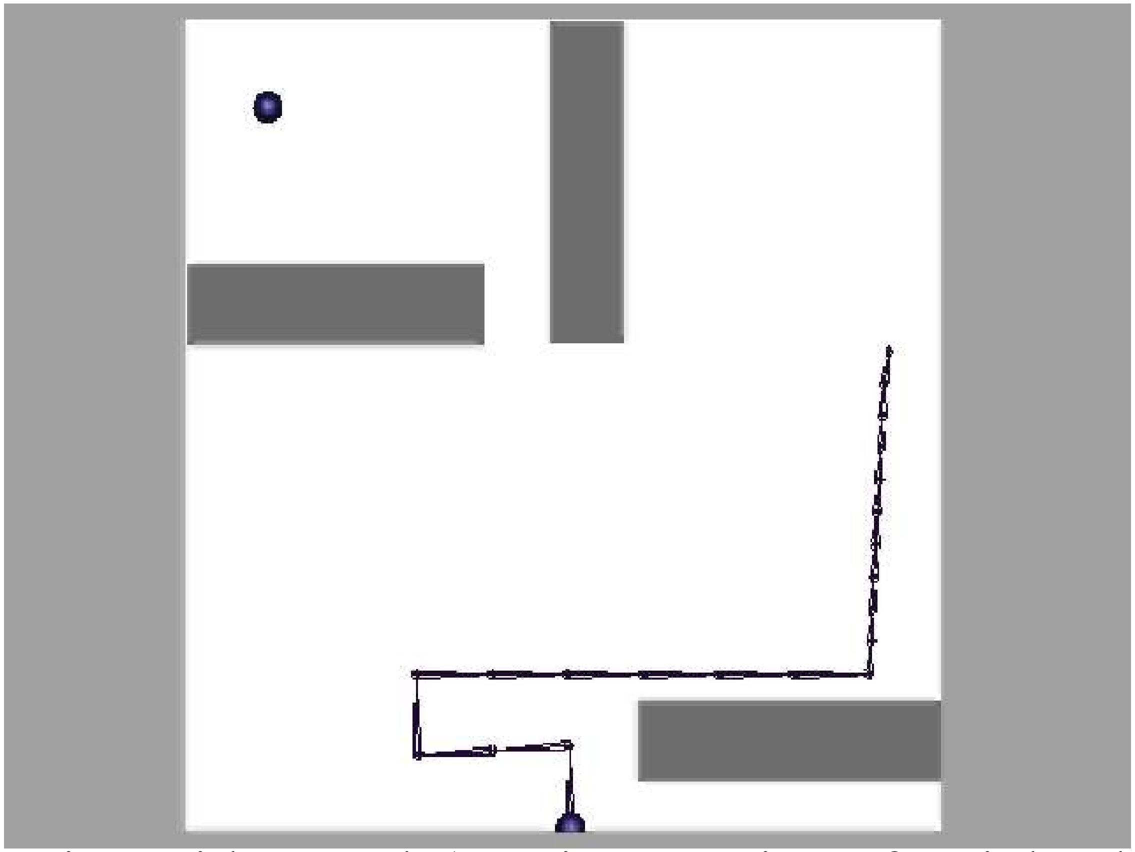


$\epsilon = 1.0$ (optimal search)

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

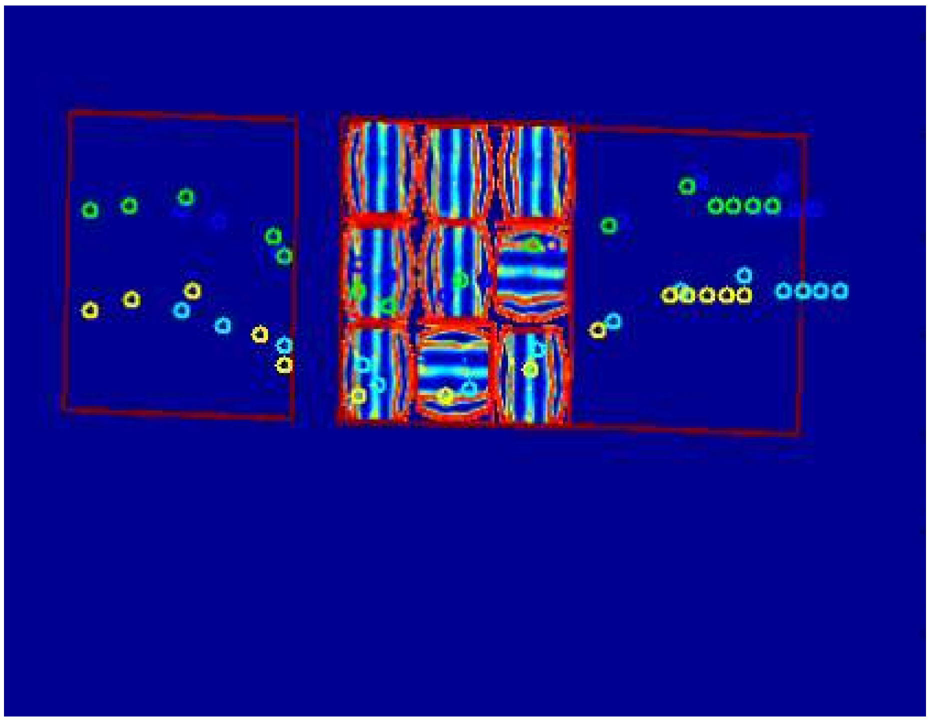
20DOF simulated robotic arm
state-space size: over 10^{26} states



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

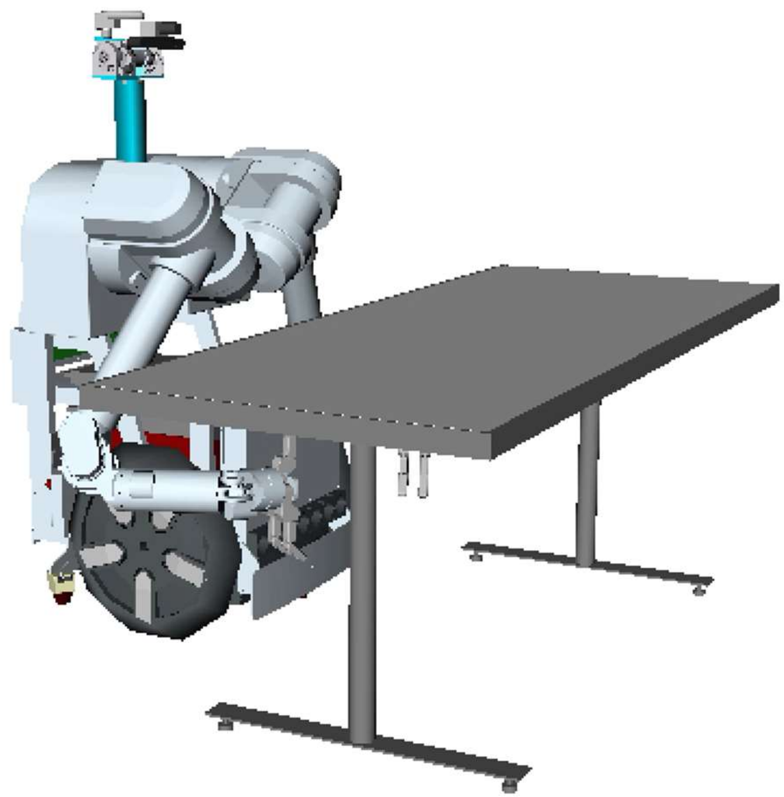
Effect of the Heuristic Function

- 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



Uses R* - A randomized version of weighted A*
Joint work between Max Likhachev, Subhrajit Bhattacharya, Joh Bohren, Sachin Chitta, Daniel D. Lee, Aleksandr Kushleyev, and Paul Vernaza

Another example of A* in Action



Did we solve motion planning?