ETH zürich

Planning I
- getting from A to B

Roland Siegwart, Margarita Chli, Juan Nieto, Nick Lawrance

Additional thanks to Jen Jen Chung for many of the slides

Today

- Motion planning
- Representing planning problems, configuration space
- Graph search methods
- Collision avoidance

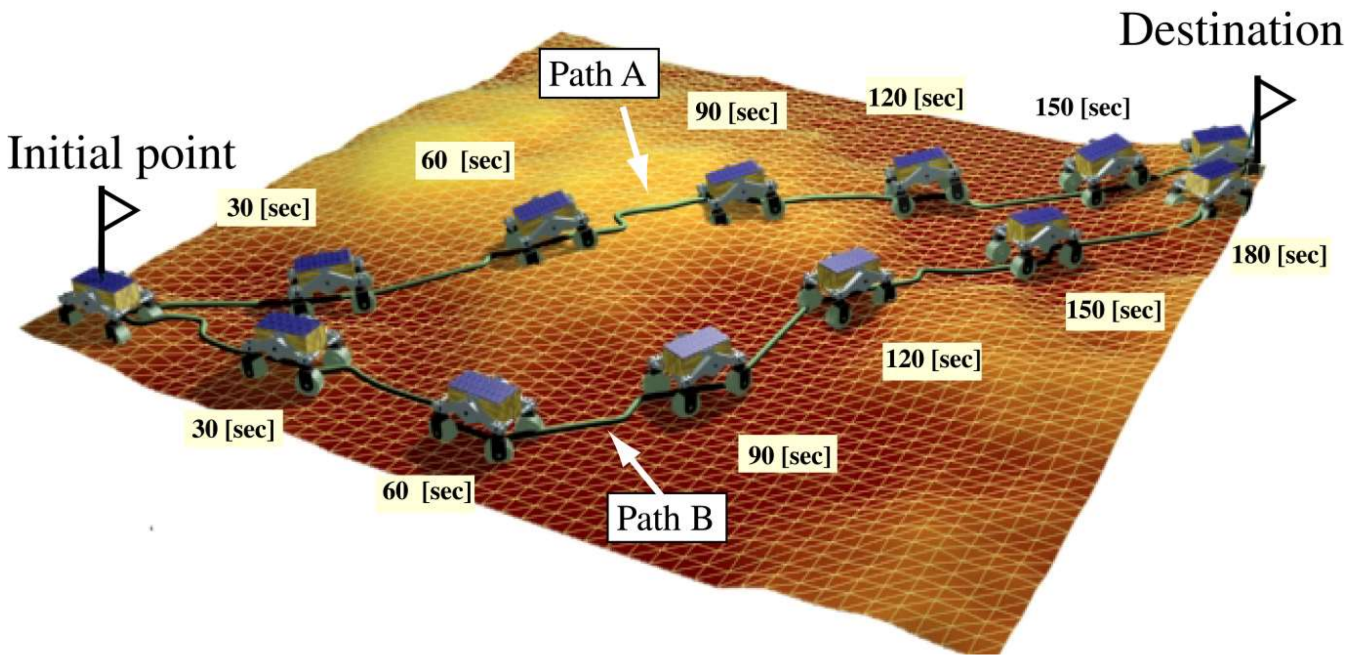
Next week – Juan Nieto

- Sampling-based methods
- Planning with uncertainty
- Recent planning research

What does planning mean?

- Different things to different people
- In general, we will focus on ***motion planning*** for robotics, namely determining a set of actions to take a robot from one known state to another known state
- We are also interested in considering some of the ***constraints*** of the platform, ensuring that our generated plans are ***feasible***

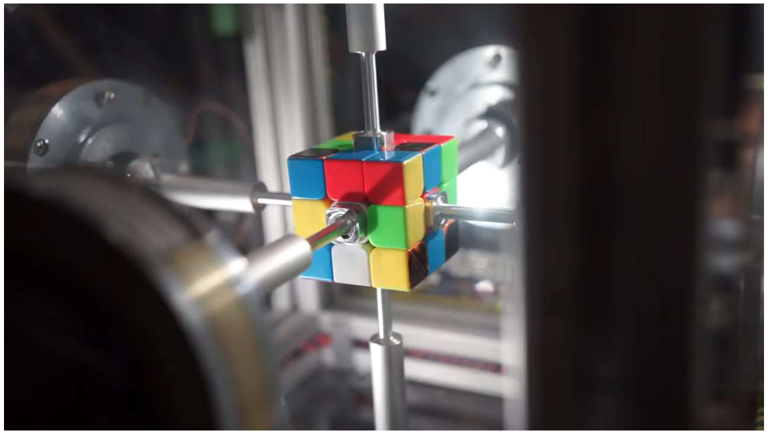
What is (robot) Planning?



Related topics

- Control:
  - Generally concerned with reaching and maintaining a desired state in some kind of robust way
  - Often **feedback-based**
  - Success measured in terms of stability, robustness, ability to reject disturbance

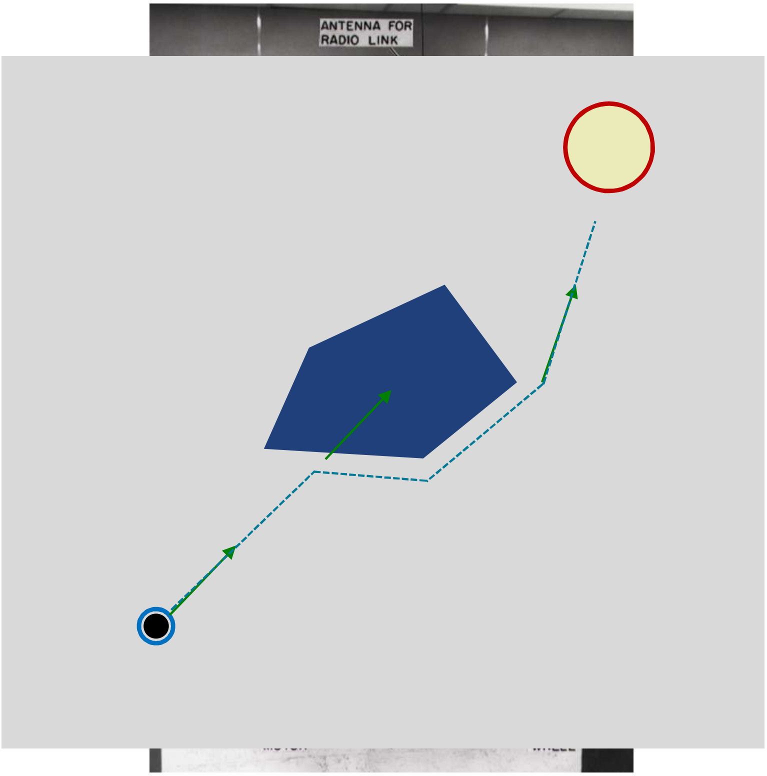
- Planning in Artificial Intelligence:
  - Generally more focused on **discrete** problems
  - Classic AI ‘planning' problems (spanning graphs, travelling salesman, orienteering) often appear in robotic planning approaches



Used with permission of Ben Katz

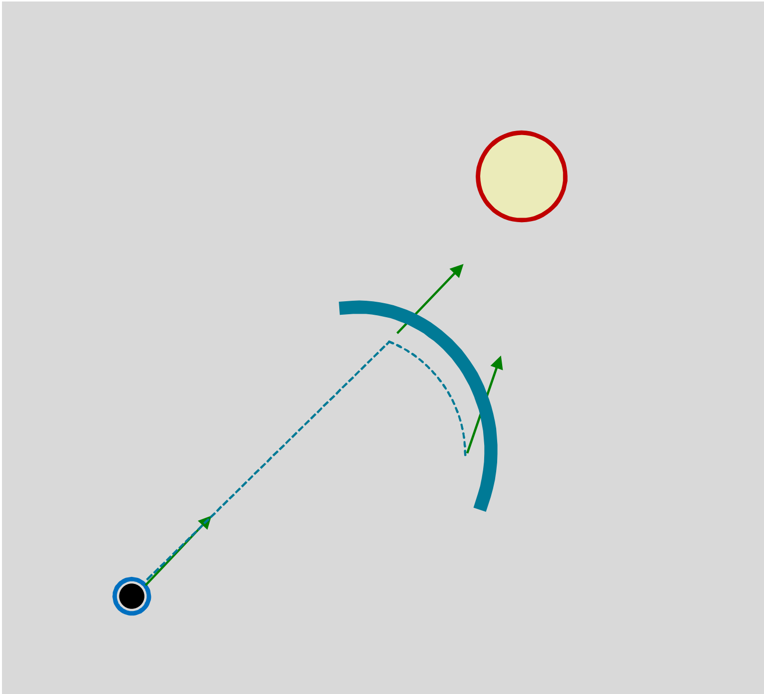
Historical robot motion planning – Adapted from Howie Choset

- Classical robotics (mid-70s)
  - Exact models, no sensing
- Reactive motion planning (mid-80s)
  - No models, rely on sensors only



Historical robot motion planning – Adapted from Howie Choset

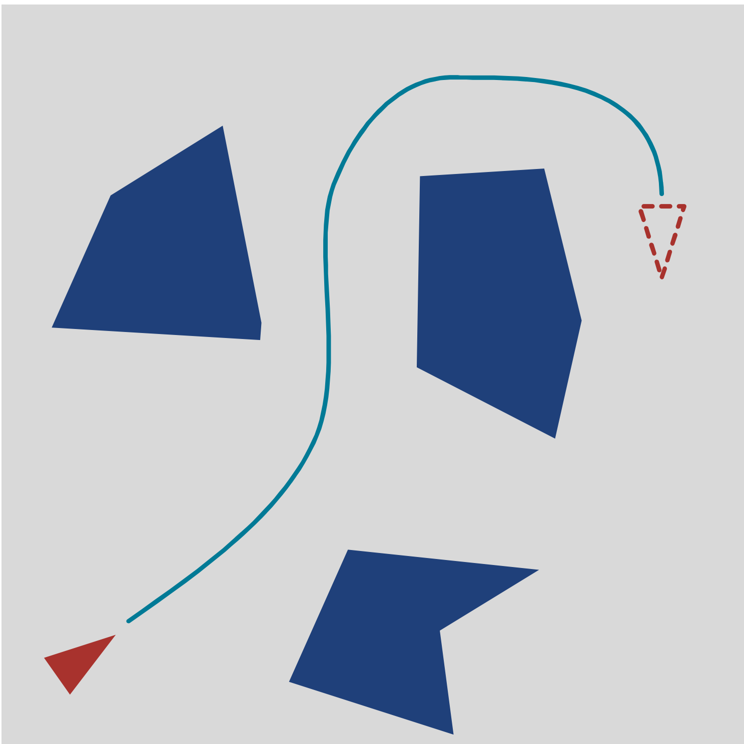
- Classical robotics (mid-70s)
  - Exact models, no sensing
- Reactive motion planning (mid-80s)
  - No models, rely on sensors only



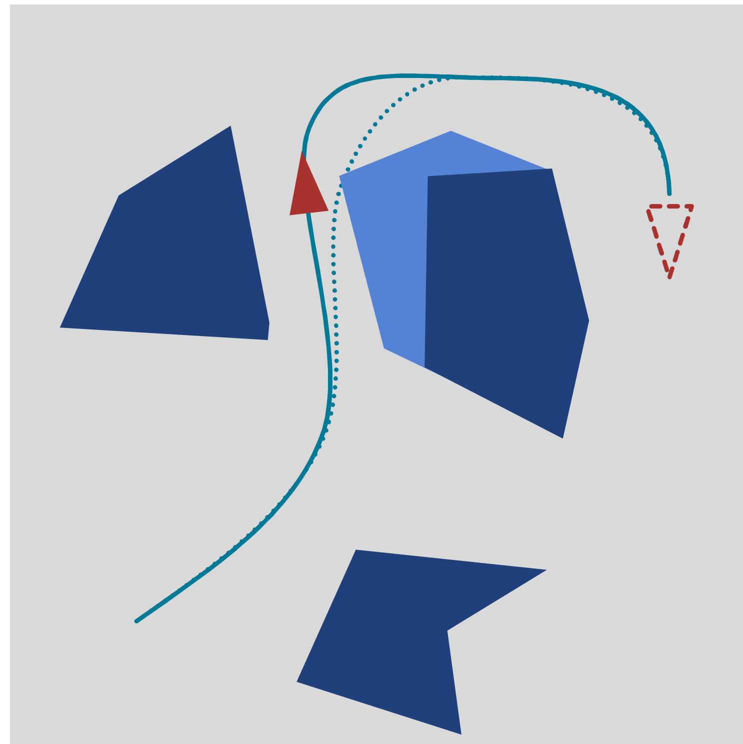
Historical robot motion planning – Adapted from Howie Choset

- Classical robotics (mid-70s)
  - Exact models, no sensing
- Reactive motion planning (mid-80s)
  - No models, rely on sensors only
- Hybrid / hierarchical (since early 90s)
  - Use models/planning at high level
  - Reactive (obstacle avoidance) at lower level
- Probabilistic (since mid 90s)
  - Incorporate uncertainty in models and sensors in all stages of planning

Navigation



- Plan



- Execute

MOTION PLANNING

Navigation Competence

- In a nutshell, work out how the robot could *feasibly* move from one position to another
- Simplifying assumptions (for now…)
  - Our **representation** of the robot and the world is sufficiently expressive
  - We know **where we are** and **where we want to go**
  - We have a **motion model** for our robot
- Typically cast as an optimisation problem – minimise cost (time, distance, energy), within constraints

Motion Planning

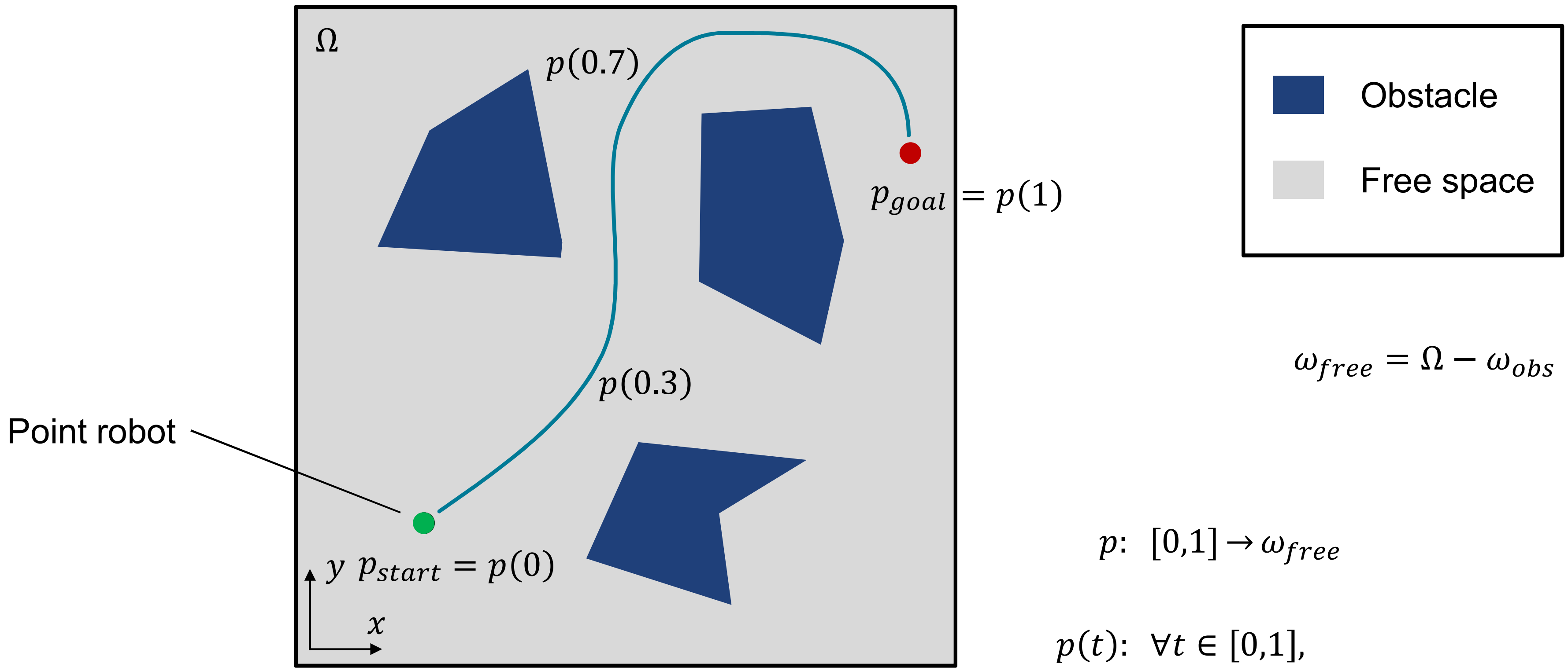
- Given start point S, goal point G:
  - Find path to take robot from S to G.
- What are challenges:
  - Obstacles?
  - Dynamics?
  - Feasibility?
- Typically cast as an optimization problem

REPRESENTATION

Representation

- How the world is represented and understood by the planner (robot) is important
- Usually some degree of simplification in choosing a representation
- By choosing a suitable representation of the world, we may be able to apply existing algorithms to solve our planning problem

Representation – workspace and paths



$$\omega_{free} = \Omega - \omega_{obs}$$

$$p\colon\ [0,1] \rightarrow \omega_{free}$$

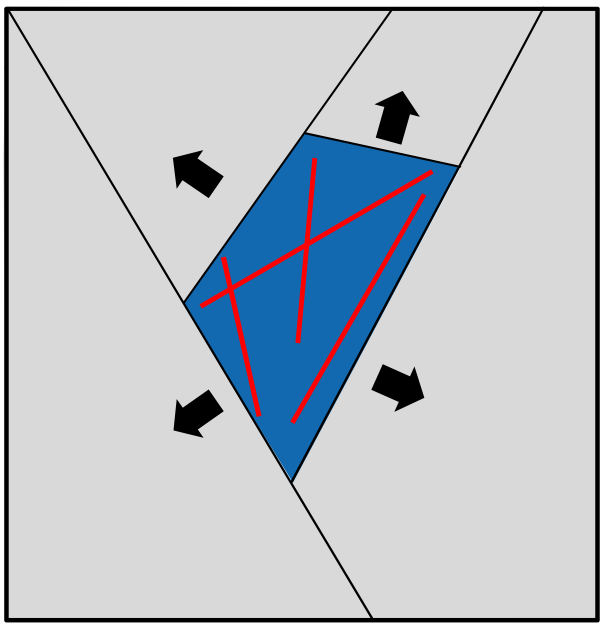
$$p(0) = p_{start}$$

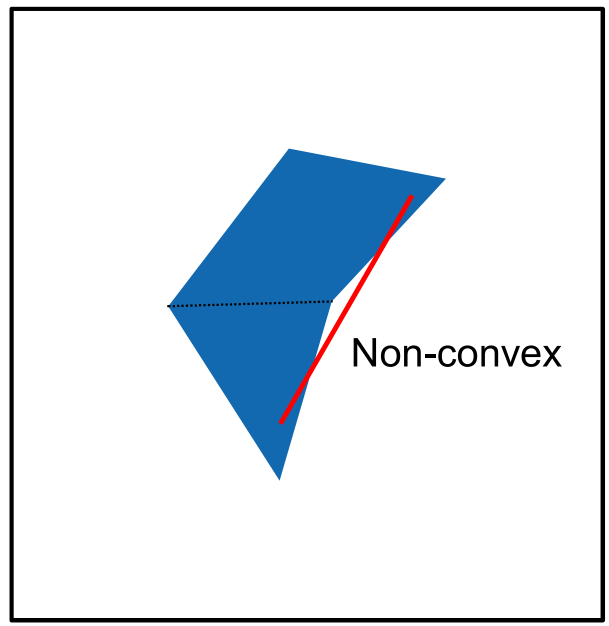
$$p(1) = p_{goal}$$

$$p(t)\colon\ \forall t \in [0,1],\ p(t) \in \omega_{free}$$

A brief aside – (polygonal) convexity

- A convex shape can be defined by a set of half-planes





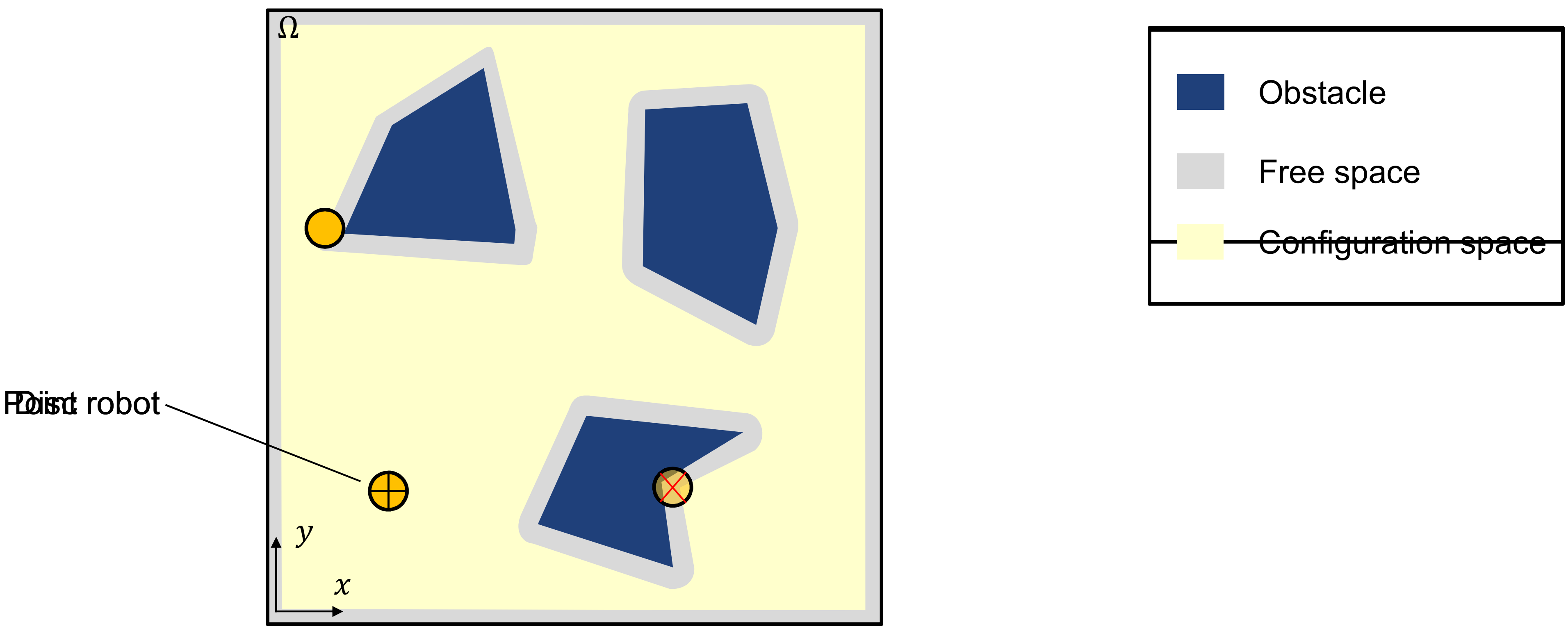
Representation – Workspace and configuration space

- The workspace is often the representation of the world, possibly independent of the robot itself. Often describes some notion of reachability, what space is free or occupied?

- Configuration space describes the full state of the robot in the world (actuator positions, orientation, etc.)

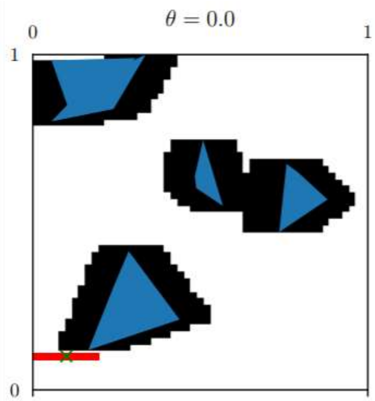
- Let’s consider that our robot is no longer a point, but occupies an area…

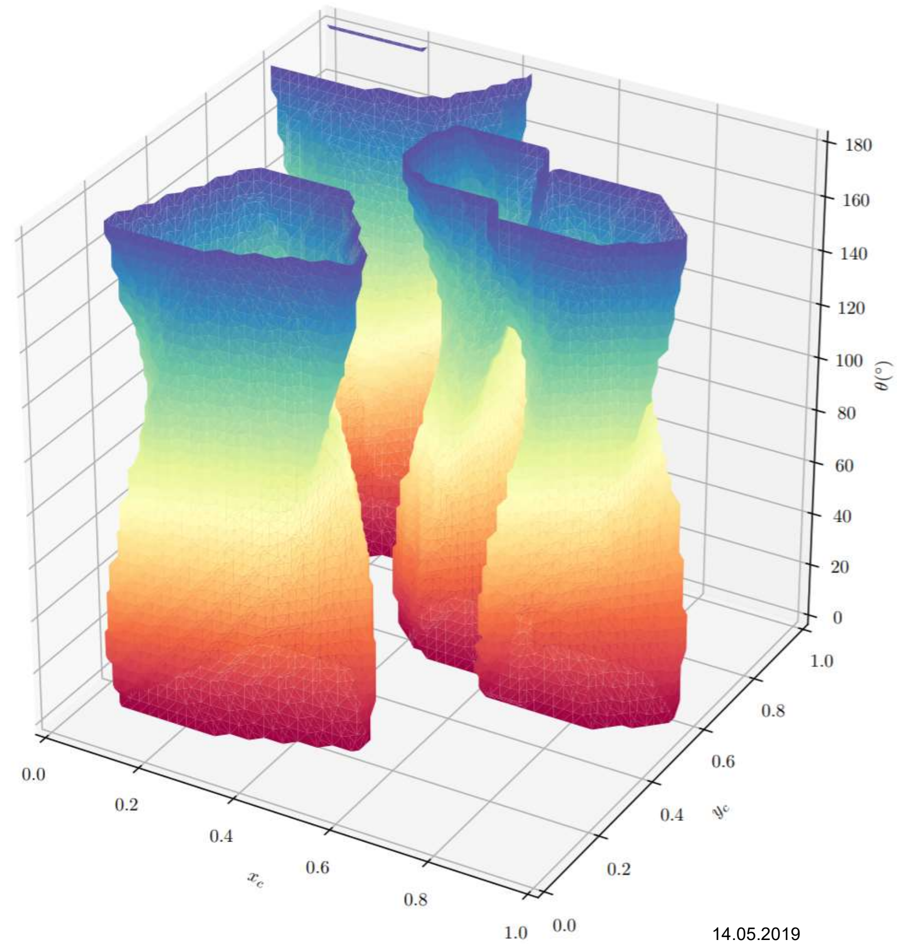
Representation – configuration space



Representation – configuration space

- A robot without rotational symmetry (x, y, θ)





Configuration space for alternative morphologies

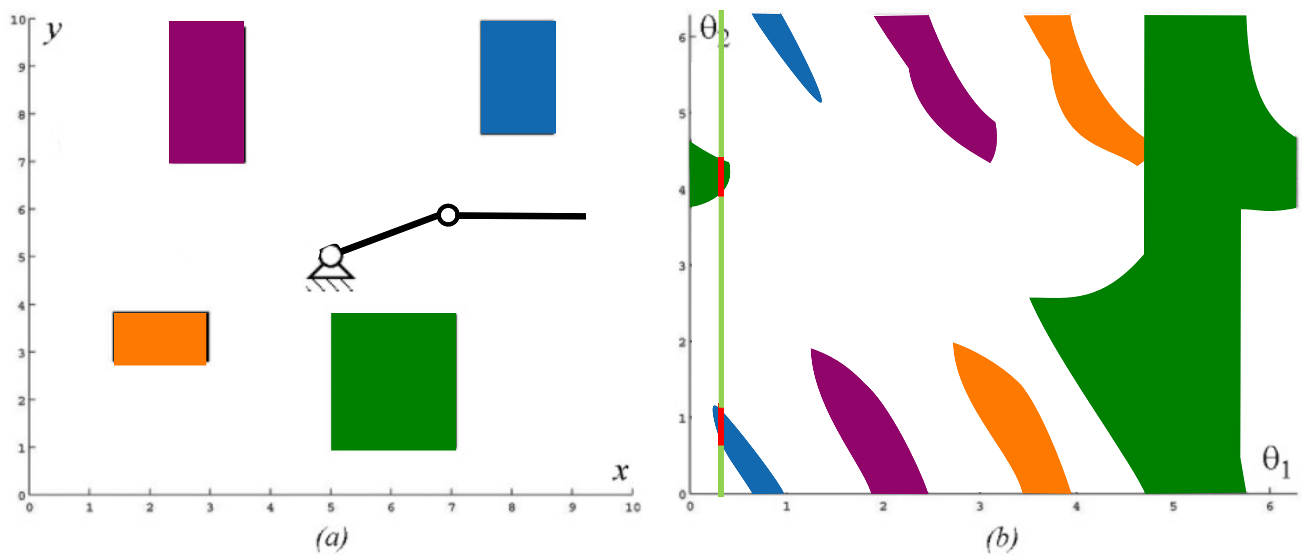


Figure 6.1
Physical space (a) and configuration space (b): (a) A two-link planar robot arm has to move from the configuration *start* to *end*. The motion is thereby constraint by the obstacles 1 to 4. (b) The corresponding configuration space shows the free space in joint coordinates (angle θ_1 and θ_2) and a path that achieves the goal.

Configuration space for alternative morphologies

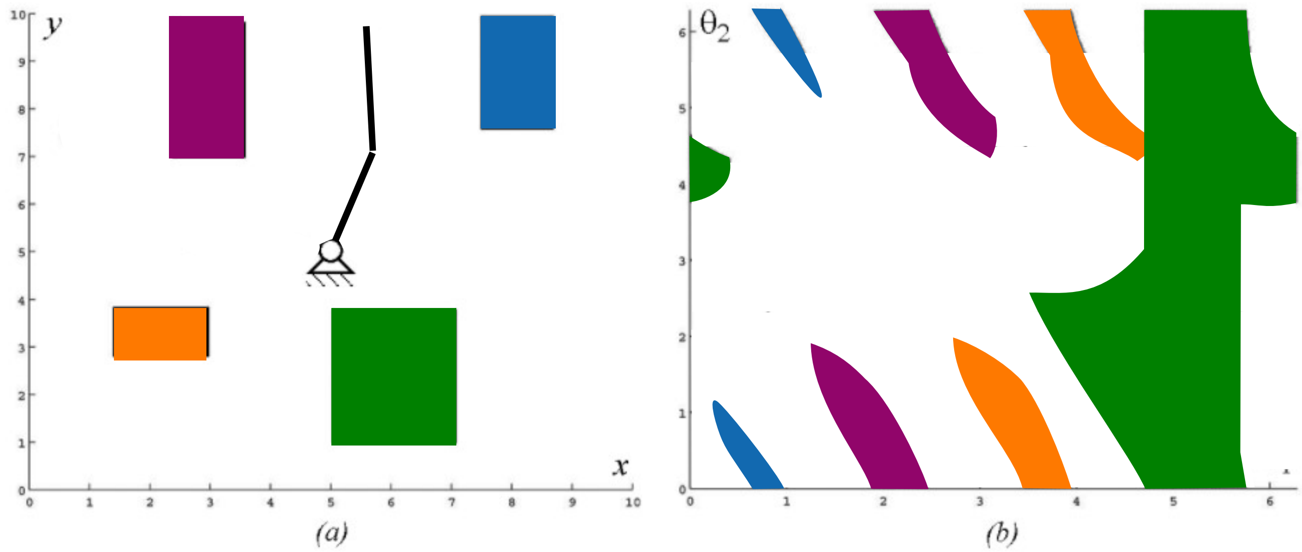


Figure 6.1
Physical space (a) and configuration space (b): (a) A two-link planar robot arm has to move from the configuration *start* to *end*. The motion is thereby constraint by the obstacles 1 to 4. (b) The corresponding configuration space shows the free space in joint coordinates (angle θ_1 and θ_2) and a path that achieves the goal.

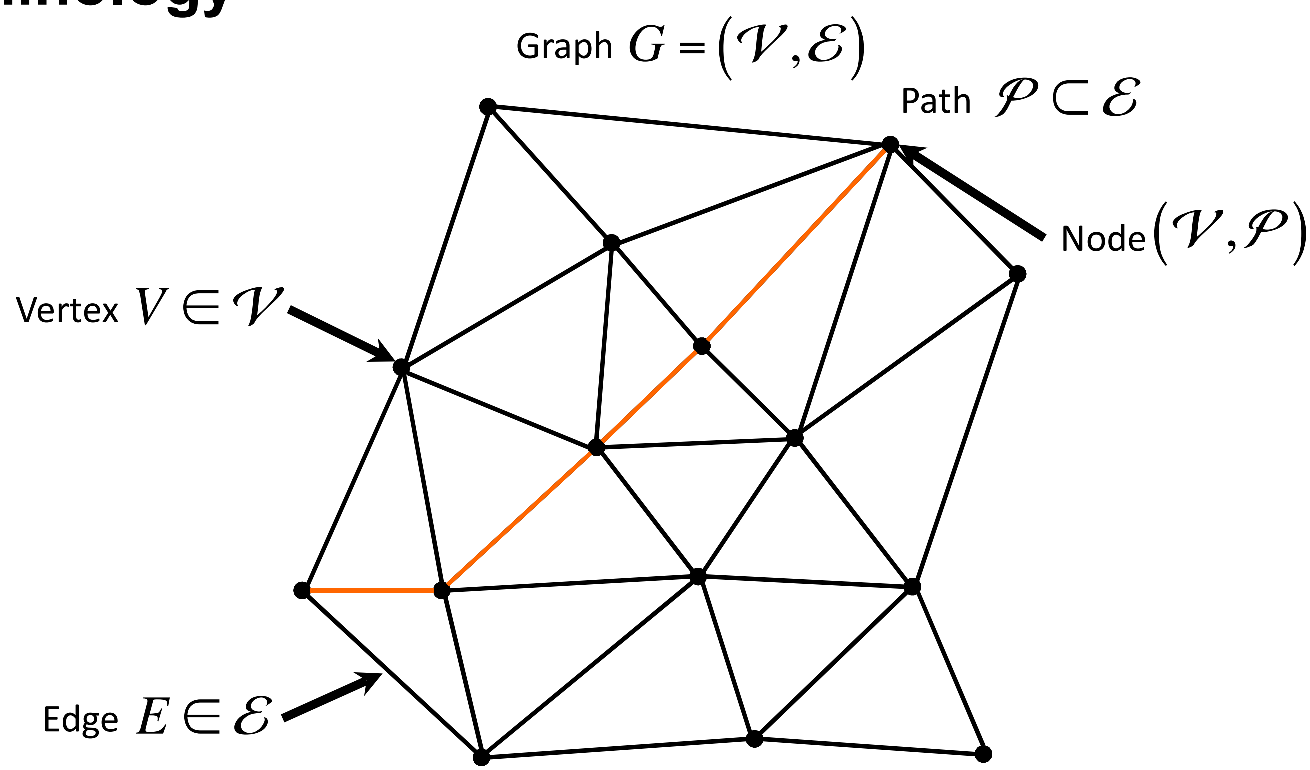
Why use configuration space?

- Positions in configuration space tend be close together for the robot
- Can be easier to solve collision checks, and join nearby poses
- Allows a level of abstraction that means solution methods can solve a wider range of problems
- Sometimes helps with wraparound conditions (rotation joints)

Continuous vs discrete state space representations

- Although continuous representations have some nice mathematical properties, they aren't always convenient
- Since computers store things digitally, there are some clever and efficient ways to create (approximate) discrete representations
- It is ***very*** common to convert a planning problem to some kind of (discrete) graph representation, then use one of a variety of existing search algorithms on the graph

Some Terminology



Directed graph: edges have direction
Weighted graph: edges have costs

Discrete state space representation

- Reduce continuous state space to a finite set of discrete **states**

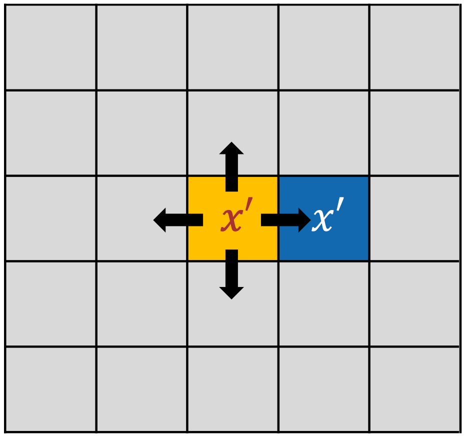
$$x \in X$$

- Also define feasible **actions** from each state

$$A(x) = \{a_0, a_1, \dots, a_n\}$$

- And an associated **transition function**

$$f(x, a) = x'$$



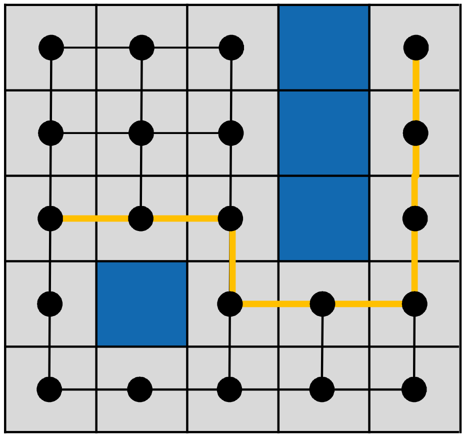
Grid → Graph

- Consider:
  - States as vertices
  - Transitions as directed edges

- The result is a graph

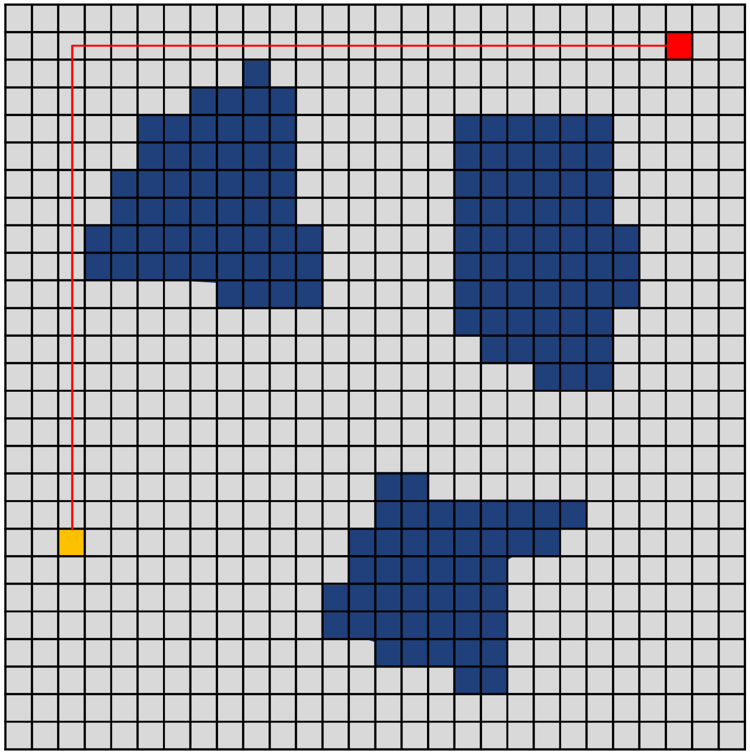
- Add:
  - Start node, x_s
  - Goal node, x_g
  - Cost function $C: X \times A \to \mathbb{R}^+$

- Finding the shortest path can be treated as a graph search problem



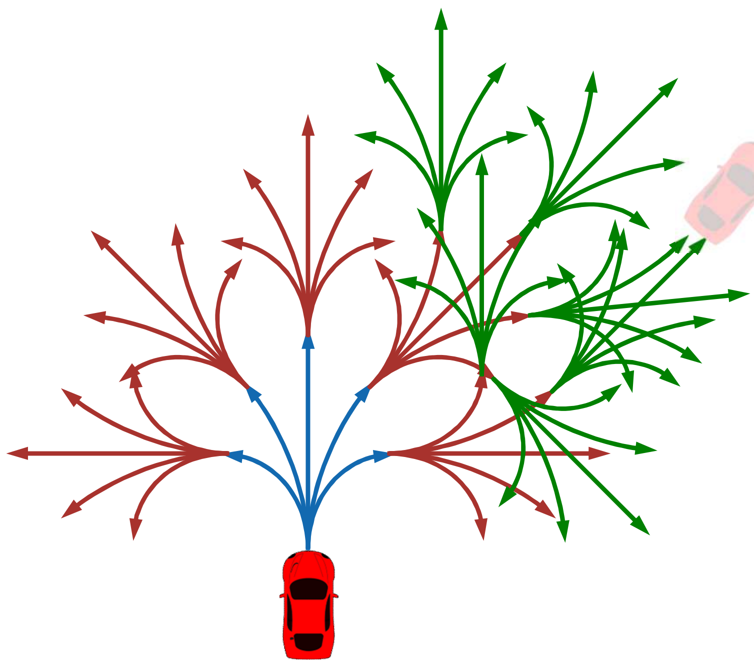
Issues with grid-based representations

- Usually suffer some loss of precision
- Selecting an appropriate grid resolution can be a challenge (multi-resolution mapping)
- Can limit the type of output path
- Suffer from poor scaling in higher dimensions



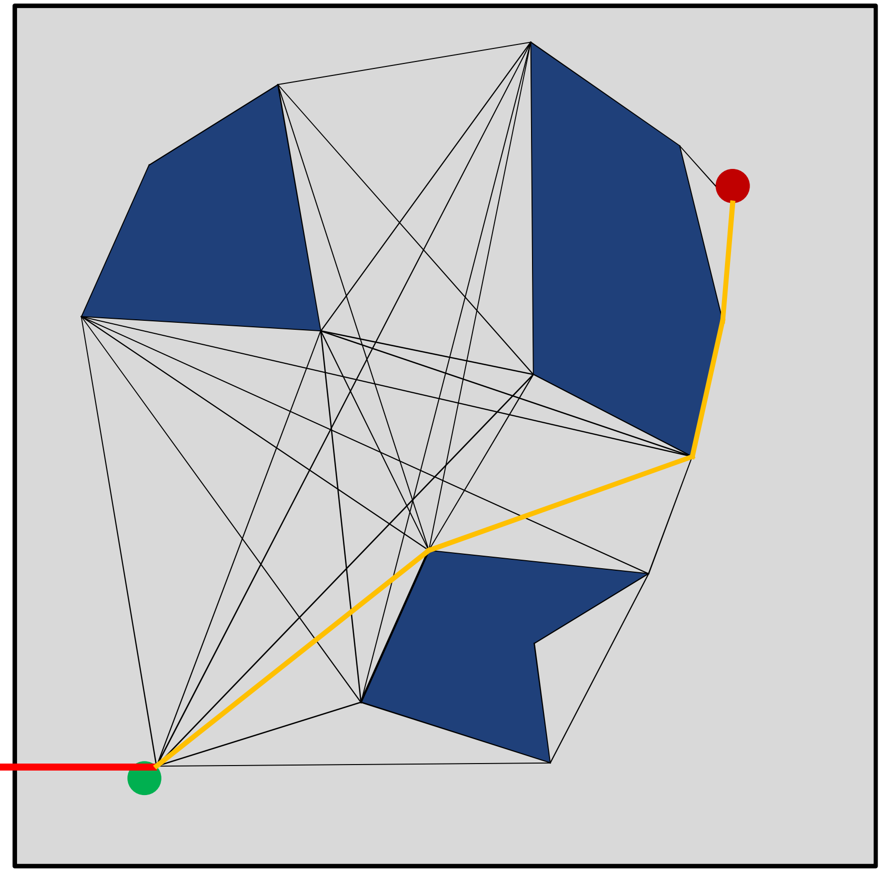
Brief aside – other graph-based representations

- Grid lattice – create a set of feasible motion primitives, and construct a tree (graph) that chains the motions into a sequence (plan)



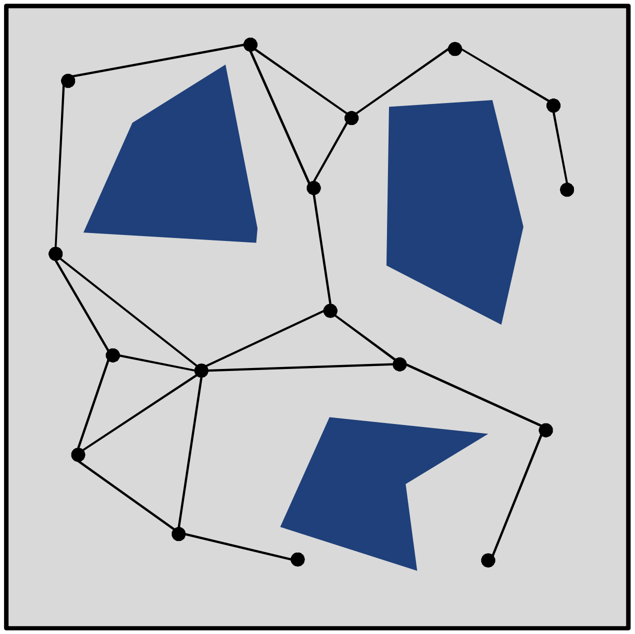
Visibility graph

- Create edges between all pairs of mutually visible vertices
- Search resulting graph
- Optimal plan!
- Limited to straight motion, 2D, polygonal obstacles



Randomly-sampled graphs

- Especially popular for sample-based methods (next lecture)
- Require careful consideration to construct graphs with guarantees



What is Path Planning?

- Path planning: detailing a motion task into discrete motions
  - Robot must get from starting location to defined waypoint
  - Robot must not run into walls, fall down stairs, etc.
  - Ideally, path taken is most efficient (shortest) path
- Inputs to algorithm:
  - Description of task (starting and ending locations)
  - Constraints (robot's physical limitations, environmental obstacles, etc.)
  - Uncertainty about robot and environment
- Outputs of algorithm:
  - Physical path
    - Set of waypoints

Planning with graphs

- Given a representation, a start, a goal, and a motion model, how do we actually generate a plan?
- Many planning approaches use graphs, because we know how to search graphs and computers are good at it
- Solve your planning problem in three easy steps:

  1. Convert problem to a graph
  2. Search the graph
  3. Profit!

Representation questions

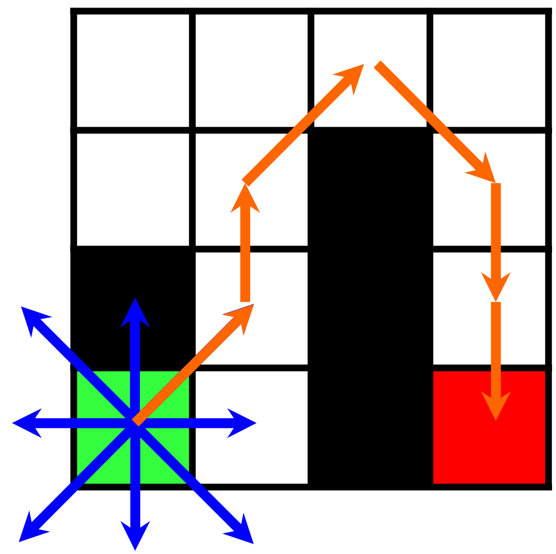
- What is the difference between work space and configuration space?
  - Work space describes the world without the robot, configuration space describes the robot's state in the world

- How do we

GRAPH SEARCH METHODS

Example 1: Discretizing the World

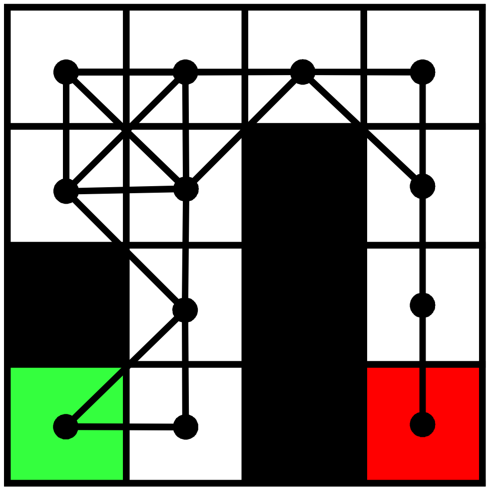
- Move from green cell to red cell
- 8-connected grid motion
- Shortest path?
- What if robot motion was restricted to 4-connected grid motion?



Assumptions about robot motion

Example 1: Discretizing the World

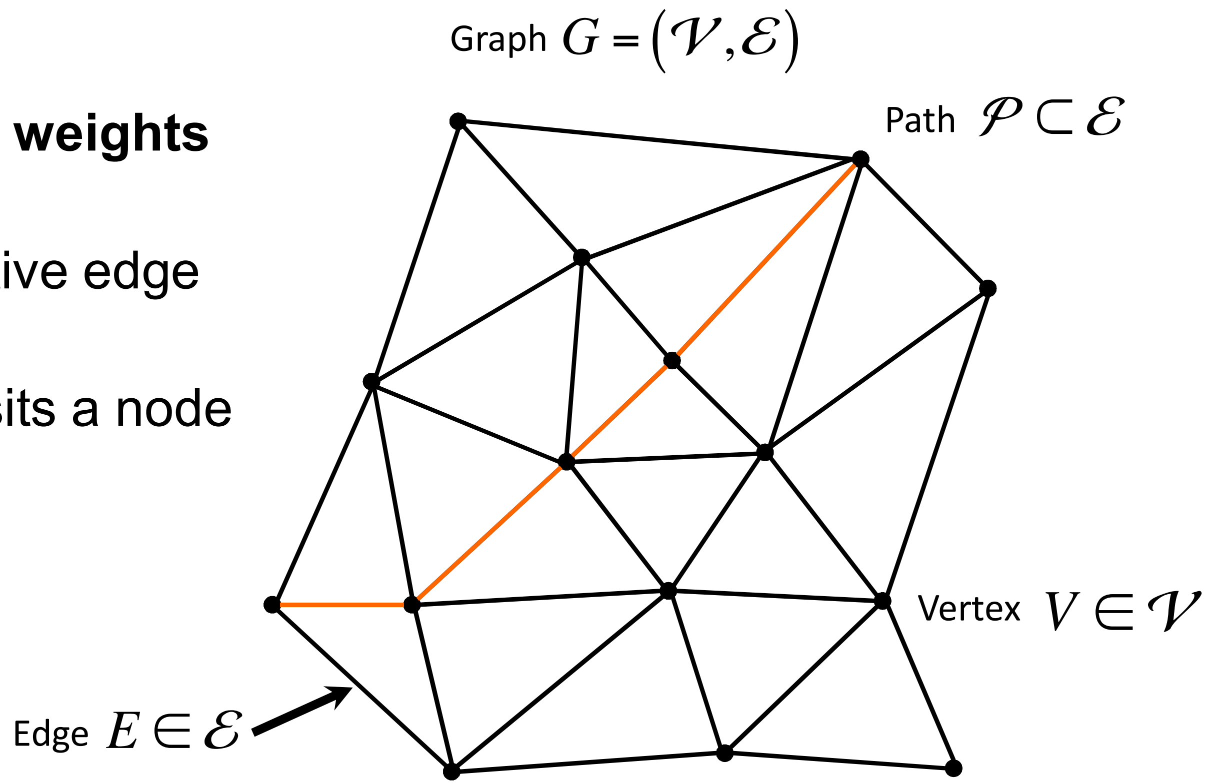
- Search over the underlying graph
- Solve for paths from any point to any other point
- Assume all edge transitions are dynamically feasible



Graph Search

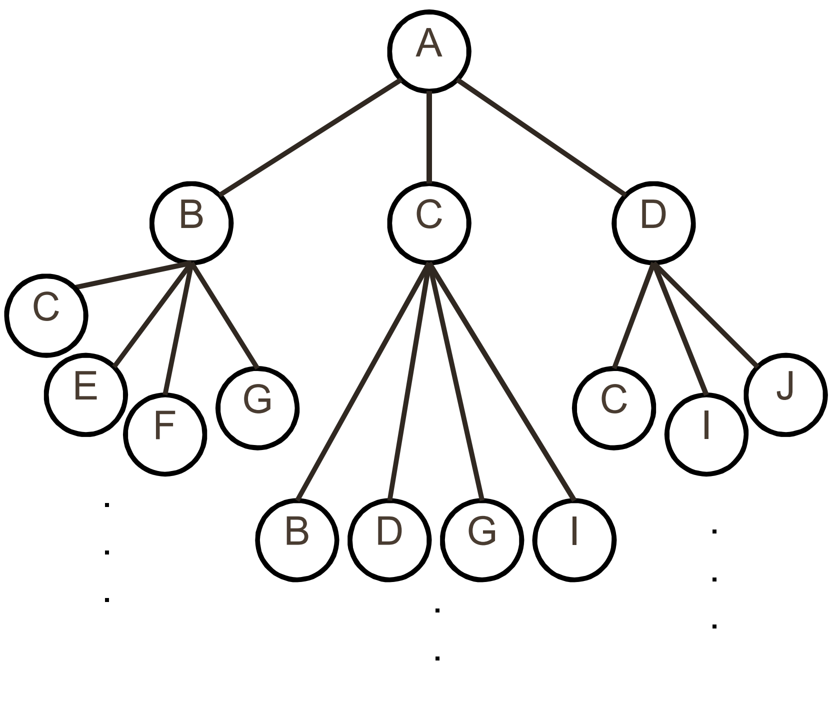
- Edges can also have associated **weights** (cost of traversing the edge)
- We generally only consider positive edge weights
- A minimum cost path never revisits a node

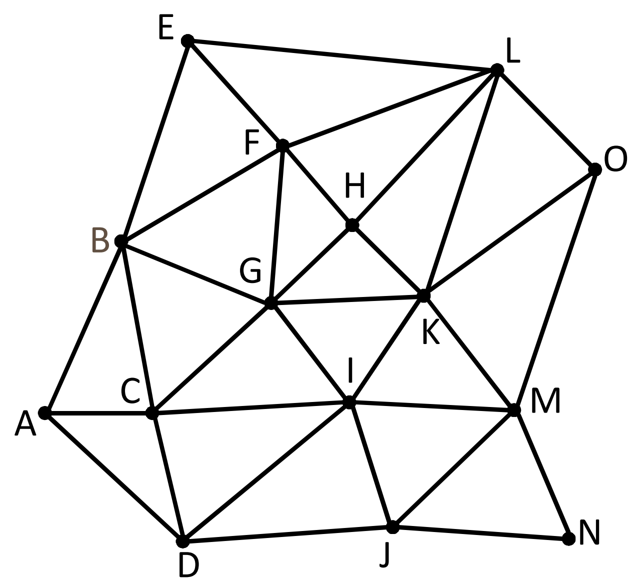
- Graph search methods
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
  - Dijkstra's Algorithm
  - A*



Search Trees

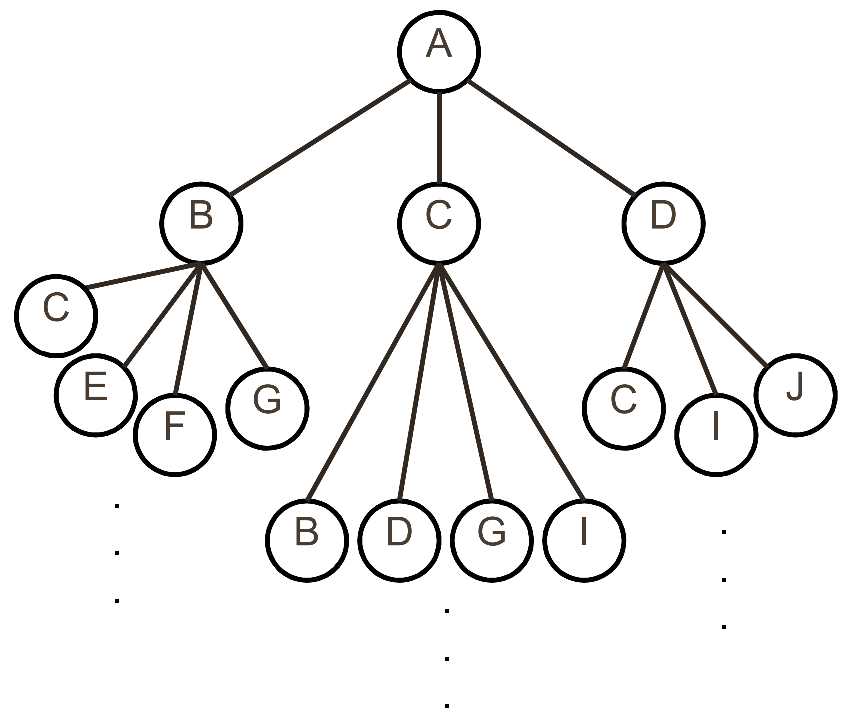
- We construct a “tree” through which we can search for optimal paths through the environment





Search Trees

- A search tree:
  - Start state at the root node
  - Children correspond to successors
  - A node corresponds to a unique **plan** from start to that state (follow up tree from node)
  - For most problems, we want to avoid building the whole tree
  - Use search algorithms to efficiently traverse tree



Basics of Forward Search

- Generally, start from the start, grow tree until you find a solution (path to goal)
- Expanding a node refers to adding children to the tree, pushing them onto the open set
- Try to expand as few tree nodes as possible

- **Open** set maintains a list of frontier (unexpanded) plans
  - Keeps track of what nodes to expand next
  - Often stored as a priority queue
  - For each node in the open list, we know of at least one path to it from the start

- **Closed** set keeps track of nodes that have been expanded
  - For each node in the closed list, we've already found the lowest-cost path to it from the start

Forward Search algorithm from LaValle

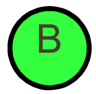
FORWARD_SEARCH

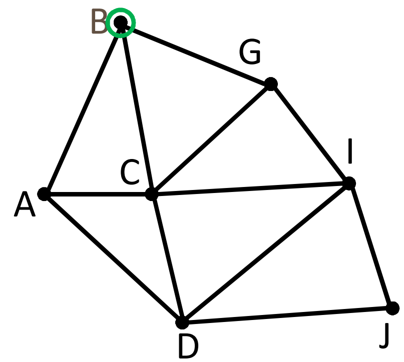
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
    $x \leftarrow Q.GetFirst()$
    if $x \in X_G$
        return SUCCESS
    forall $u \in U(x)$
        $x' \leftarrow f(x, u)$
        if x' not visited
            Mark x' as visited
            $Q.Insert(x')$
        else
        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

Figure 2.4: A general template for forward search.

LaValle, Steven M. *Planning algorithms*. Cambridge university press, 2006, p. 33

Breadth-First Search Example





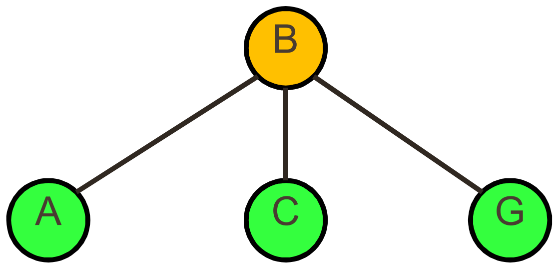

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate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

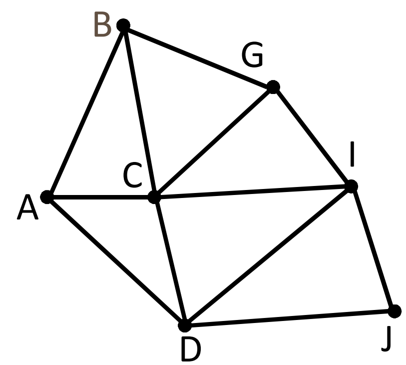
Open (Q): {B}

Closed: {}

Our (BFS) queue will be FIFO:
- push (*Q.Insert*) onto the end
- pop (*Q.GetFirst*) from the front

Breadth-First Search Example



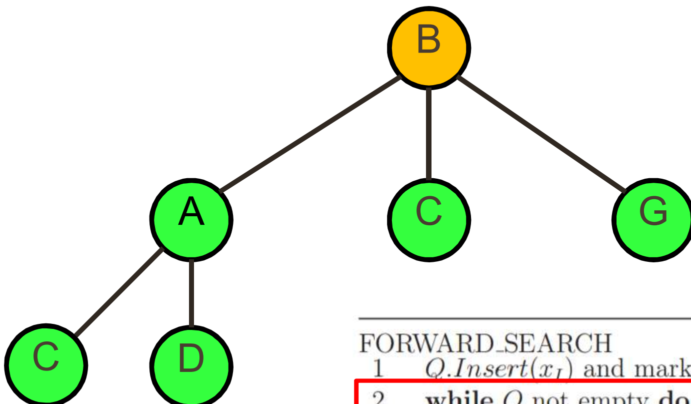


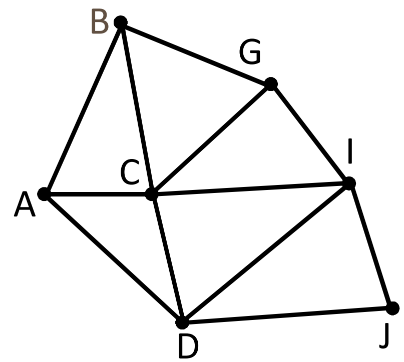

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate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

Open (Q):
{A,C,G}

Closed:
{B,A,C,G}

Breadth-First Search Example





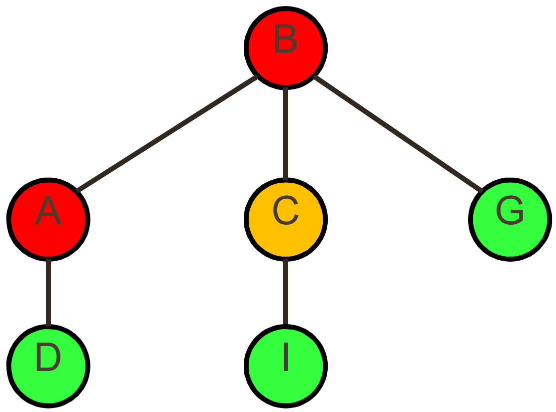
FORWARD_SEARCH

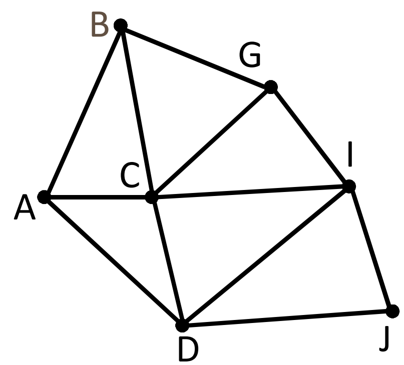
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
  $x \leftarrow Q.GetFirst()$
  if $x \in X_G$
    return SUCCESS
  forall $u \in U(x)$
    $x' \leftarrow f(x, u)$
    if x' not visited
      Mark x' as visited
      $Q.Insert(x')$
    else
      Resolve duplicate x'
return FAILURE

Open (Q):
{C,G,D}

Closed:
{B,A,C,G,D}

Breadth-First Search Example

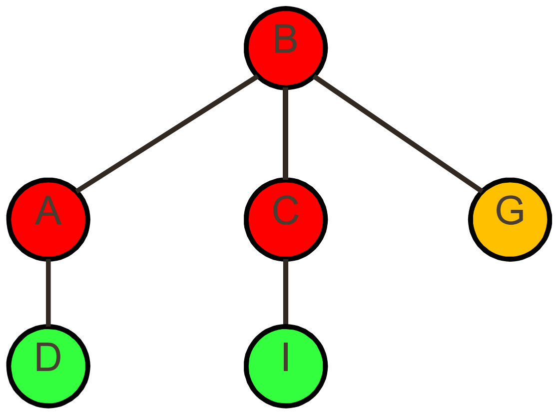


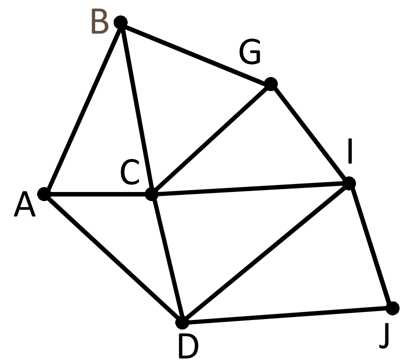


Open (Q):
{G,D,I}

Closed:
{B,A,C,G,D,I}

Breadth-First Search Example

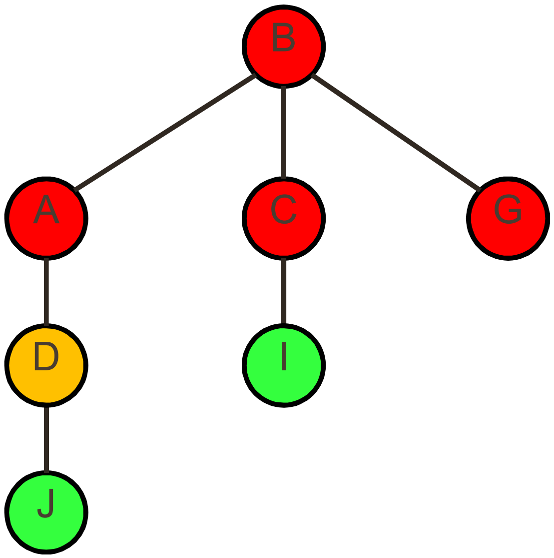


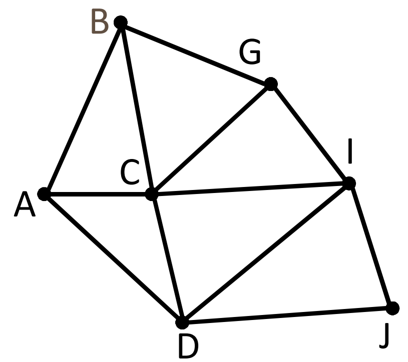


Open (Q):
{D,I}

Closed:
{B,A,C,G,D,I}

Breadth-First Search Example

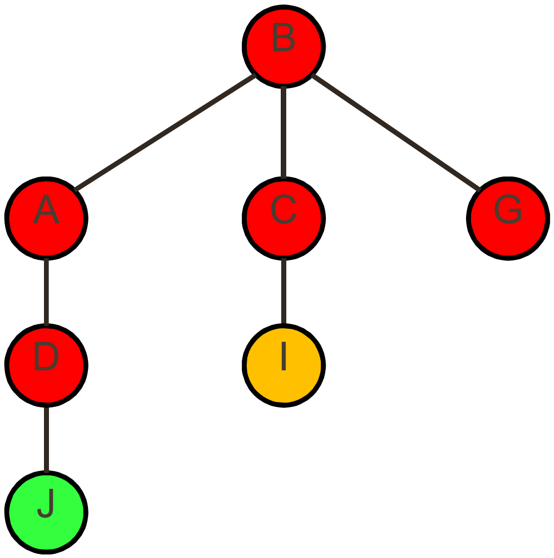


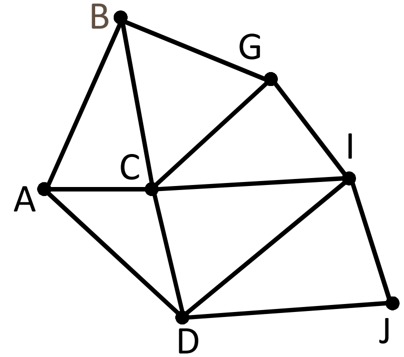


Open (Q):
{I,J}

Closed:
{B,A,C,G,D,I,J}

Breadth-First Search Example

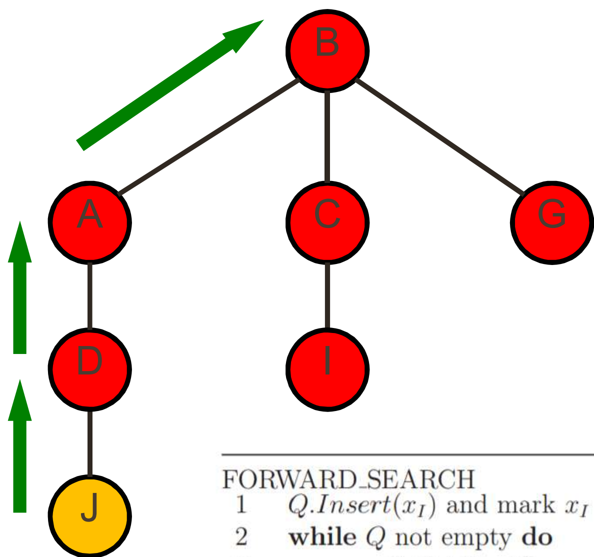




Open (Q):
{J}

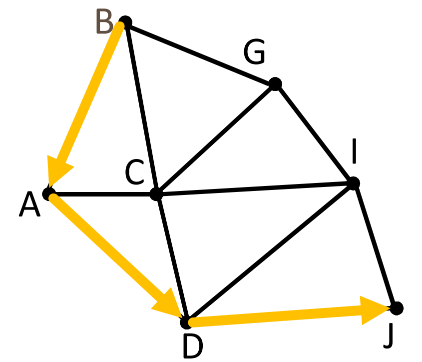
Closed:
{B,A,C,G,D,I,J}

Breadth-First Search Example



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



Open (Q):
{}

Closed:
{B,A,C,G,D,I,J}

Final path solution: B→A→D→J

Other solutions may exist but have the same number or more transitions

Breadth-First Search

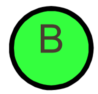
- Complete (will find the solution if it exists)
- Guaranteed to find the shortest (number of edges) path
  - First solution found is the optimal path
- What about non-uniform edge weights? (… Dijkstra)
- Time complexity O(|V|+|E|)
- Consider another approach: Depth-first search

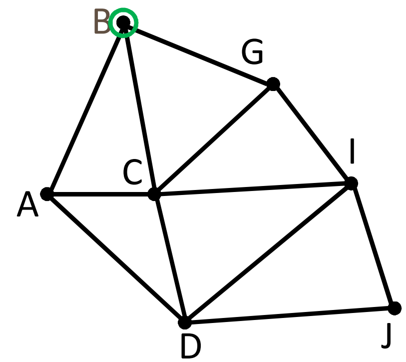
Depth-first search

- Instead of searching across levels of the tree, DFS starts at the root node and explores as far as possible along each branch before backtracking

- Similar implementation to BFS, but with a stack (last-in first-out) queue

Depth-First Search Example





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

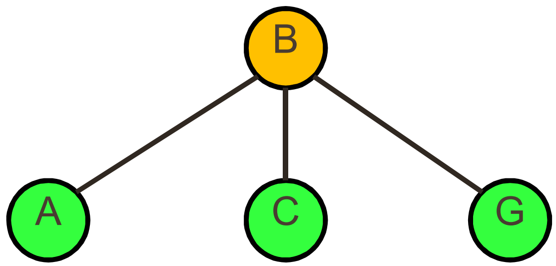
Open (Q):
{B}

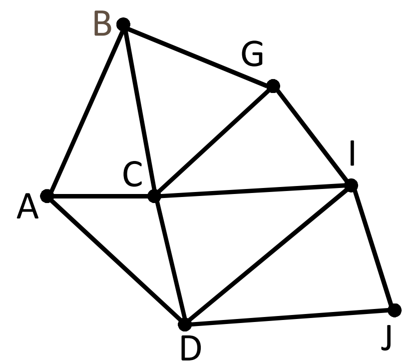
Closed:
{}

Our (DFS) queue will be LIFO:

- push ($Q.Insert$) onto the front
- pop ($Q.GetFirst$) from the front

Depth-First Search Example



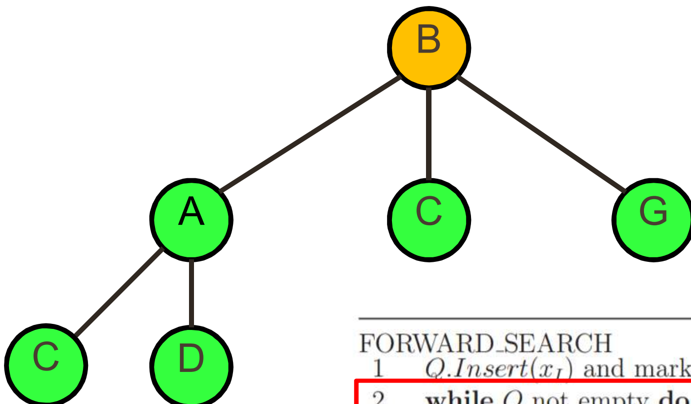


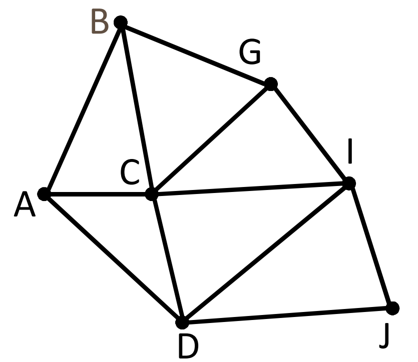
FORWARD_SEARCH
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
  $x \leftarrow Q.GetFirst()$
  if $x \in X_G$
    return SUCCESS
  forall $u \in U(x)$
    $x' \leftarrow f(x, u)$
    if x' not visited
      Mark x' as visited
      $Q.Insert(x')$
    else
      Resolve duplicate x'
return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

Open (Q):
{A,C,G}

Closed:
{B,A,C,G}

Depth-First Search Example



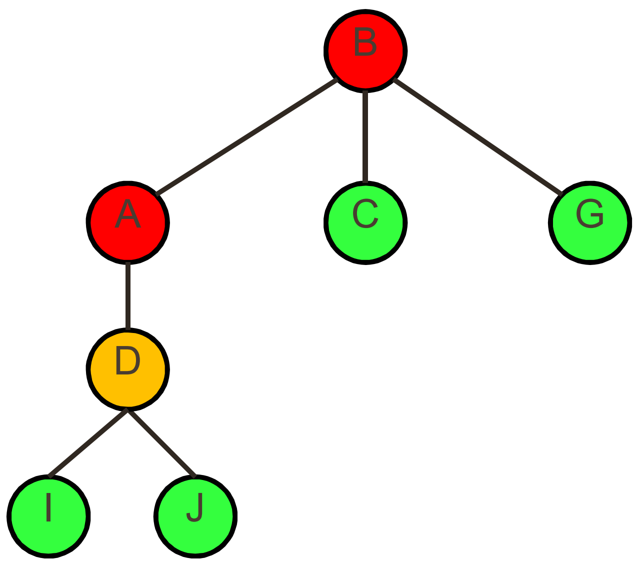


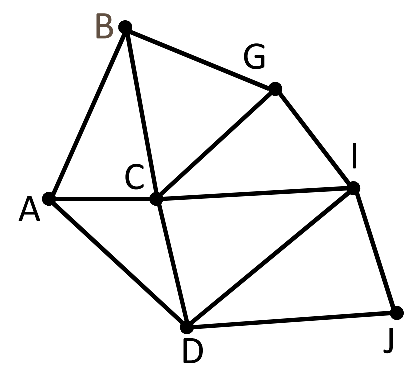

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate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

Open (Q):
{D,C,G}

Closed:
{B,A,C,G,D}

Depth-First Search Example

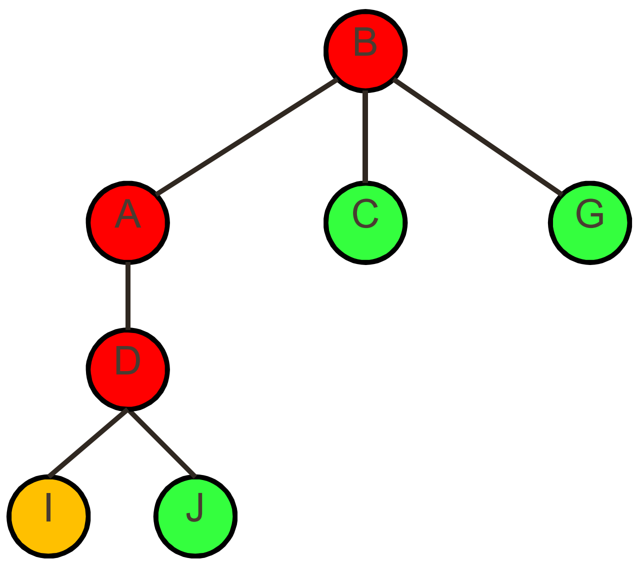


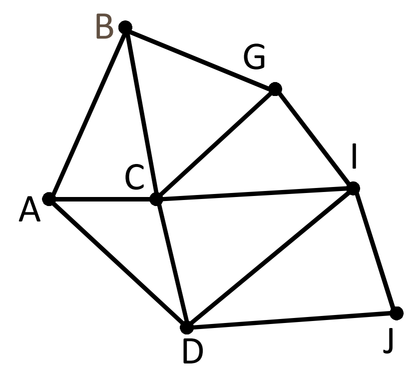


Open (Q):
{I,J,C,G}

Closed:
{B,A,C,G,D,I,J}

Depth-First Search Example

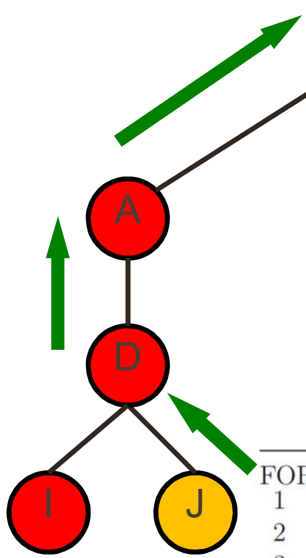




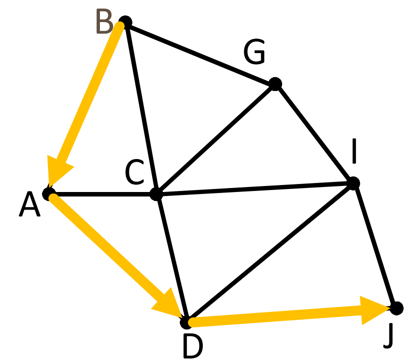
Open (Q):
{J,C,G}

Closed:
{B,A,C,G,D,I,J}

Breadth-First Search Example




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

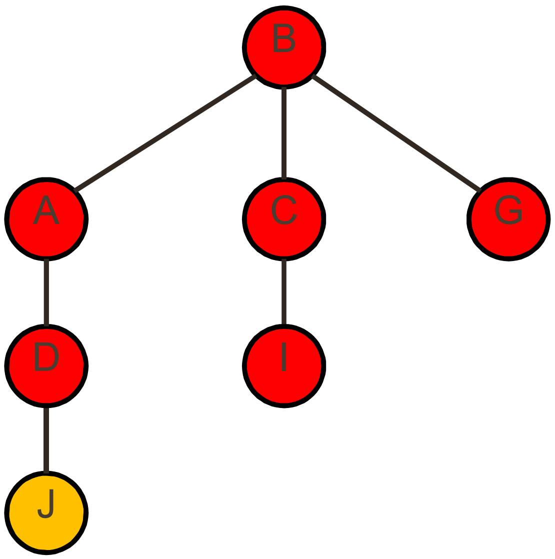


Open (Q):
{}

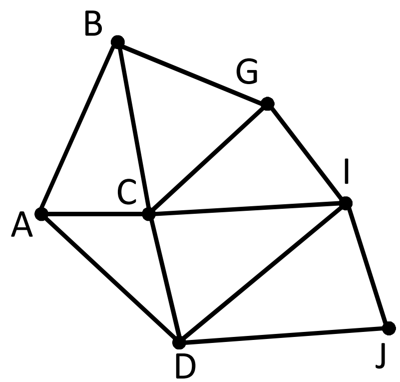
Closed:
{B,A,C,G,D,I,J}

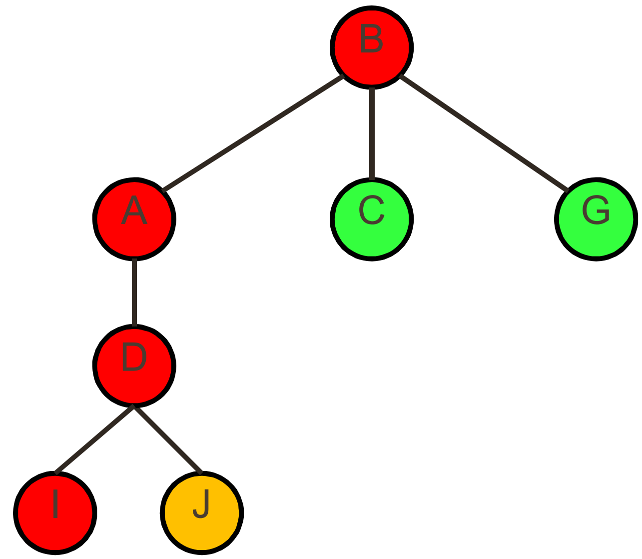
Final path solution: B→A→D→J

Search tree comparison



BFS





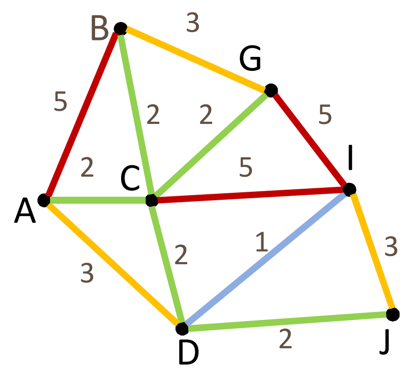
DFS

Depth-First Search

- Lower memory footprint than BFS with high-branching
- Not often used for path search, sometimes used to completely explore a graph
- Both BFS and DFS are simple to implement, but might be inefficient. More complex algorithms are faster, but generally more difficult to implement
- Seems like we want a compromise, search promising paths while we can, then go back up if they aren't working out
- DFS not complete for infinite trees (may explore an incorrect branch infinitely deep, never come back up, BFS *is* complete)

Costs on Actions

- What about non-uniform edge weights (costs)?

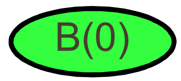


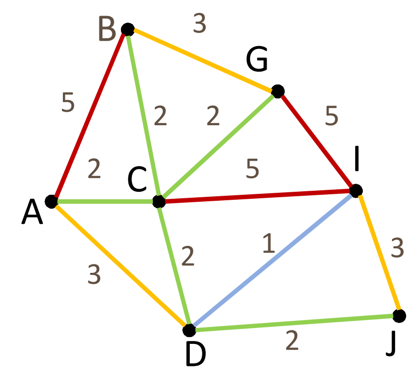
- Dijkstra's Algorithm and A* search

Dijkstra's Algorithm

- Published by Edsger Dijkstra in 1959
- Basic idea of expanding in order of closest to start (BFS with edge costs)
- One of the most commonly used routing algorithms in graph traversal problems
- Asymptotically the fastest known single-source shortest path algorithm for arbitrary directed graphs
- **Open** queue is ordered according to currently known best cost to arrive

Dijkstra’s Algorithm Example





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

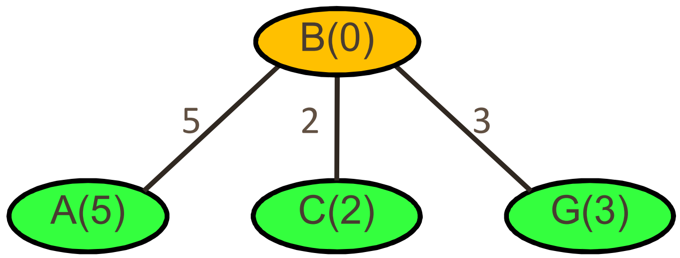
Open (Q): {B(0)}

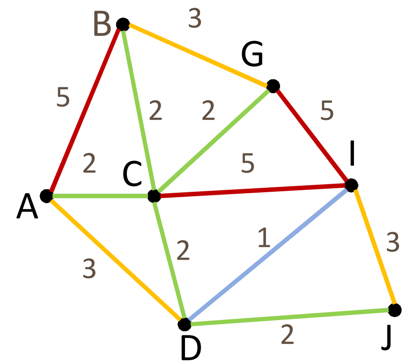
Closed: {B(0)}

Our Dijkstra queue will be ordered by cost to arrive:

- push ($Q.Insert$) by cost
- pop ($Q.GetFirst$) from the front, and add it to the closed list

Dijkstra's Algorithm Example



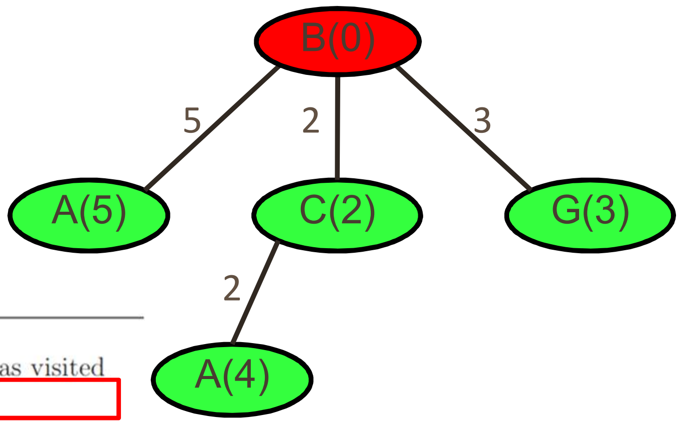


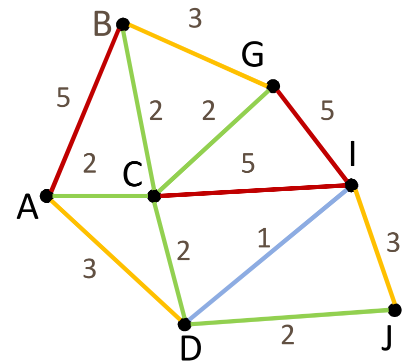

```
FORWARD_SEARCH
  Q.Insert(x_I) and mark x_I as visited
  while Q not empty do
      x ← Q.GetFirst()
      if x ∈ X_G
          return SUCCESS
      forall u ∈ U(x)
          x' ← f(x, u)
          if x' not visited
              Mark x' as visited
              Q.Insert(x')
          else
     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

Open (Q):
{ C (2),
G (3),
A (5) }

Closed:
{ B (0) }

Dijkstra's Algorithm Example



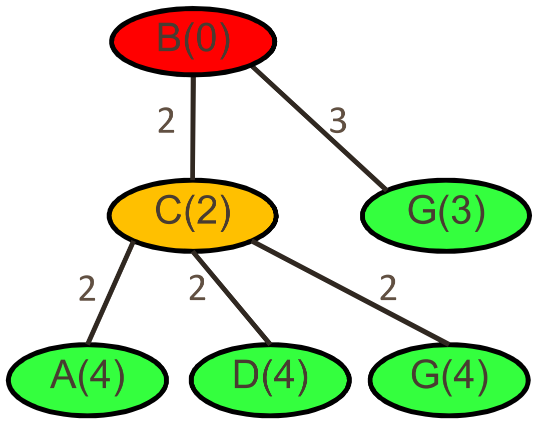


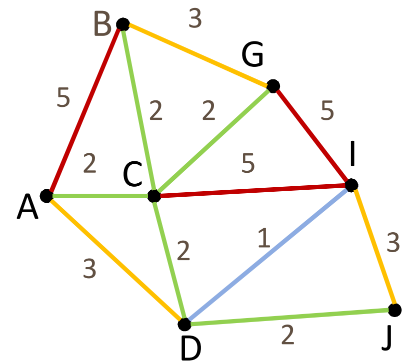

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x′ ← f(x, u)
        if x′ not visited
            Mark x′ as visited
            Q.Insert(x′)
        else
            Resolve duplicate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

Open (Q):
{ G (3),
A (4) }

Closed:
{ B (0),
C (2) }

Dijkstra's Algorithm Example



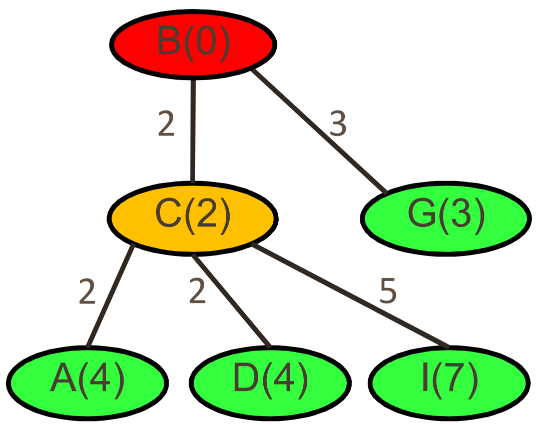


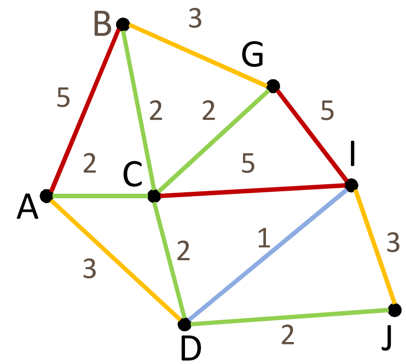

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x′ ← f(x, u)
        if x′ not visited
            Mark x′ as visited
            Q.Insert(x′)
        else
            Resolve duplicate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

Open (Q):
{ G (3),
A (4),
D (4) }

Closed:
{ B (0),
C (2) }

Dijkstra’s Algorithm Example



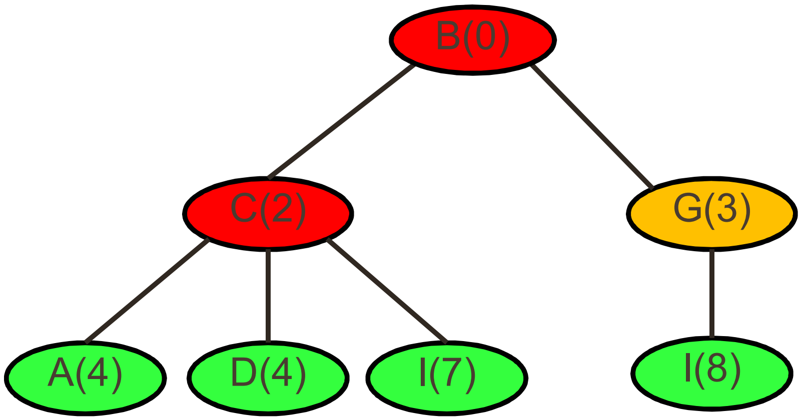


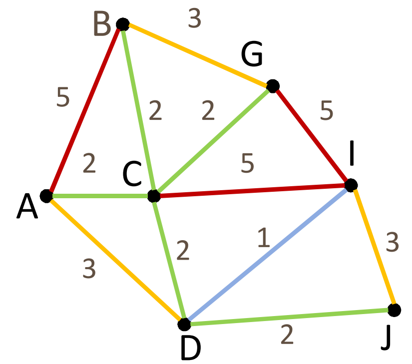

```
FORWARD_SEARCH
  Q.Insert(x_I) and mark x_I as visited
  while Q not empty do
      x ← Q.GetFirst()
      if x ∈ X_G
          return SUCCESS
      forall u ∈ U(x)
          x' ← f(x,u)
          if x' not visited
              Mark x' as visited
              Q.Insert(x')
          else
      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

Open (Q):
{ G (3),
A (4),
D (4),
I (7) }

Closed:
{ B (0),
C (2) }

Dijkstra's Algorithm Example

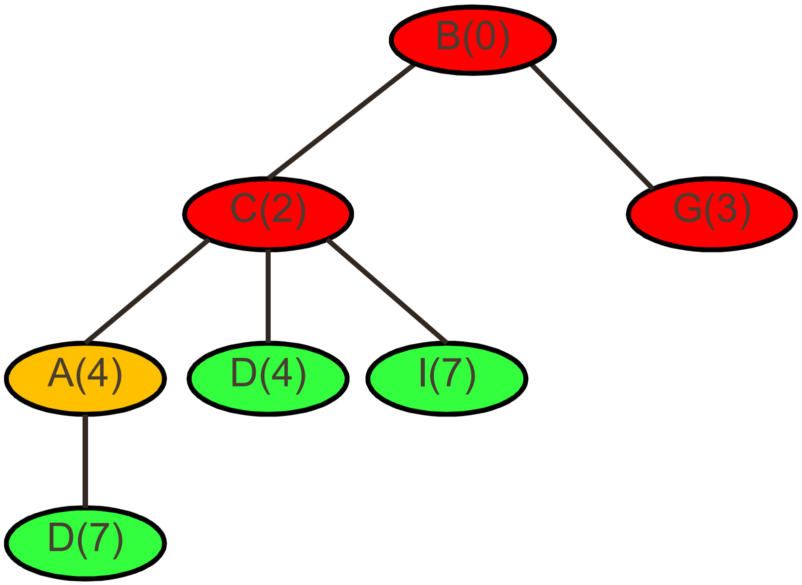


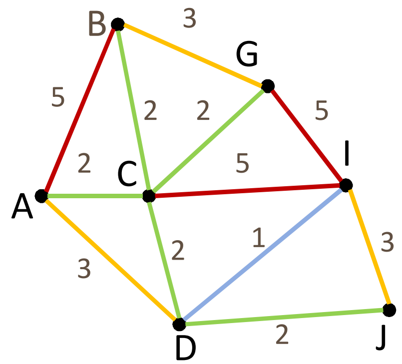


Open (Q):
{ A (4),
D (4),
I (7) }

Closed:
{ B (0),
C (2),
G (3) }

Dijkstra's Algorithm Example

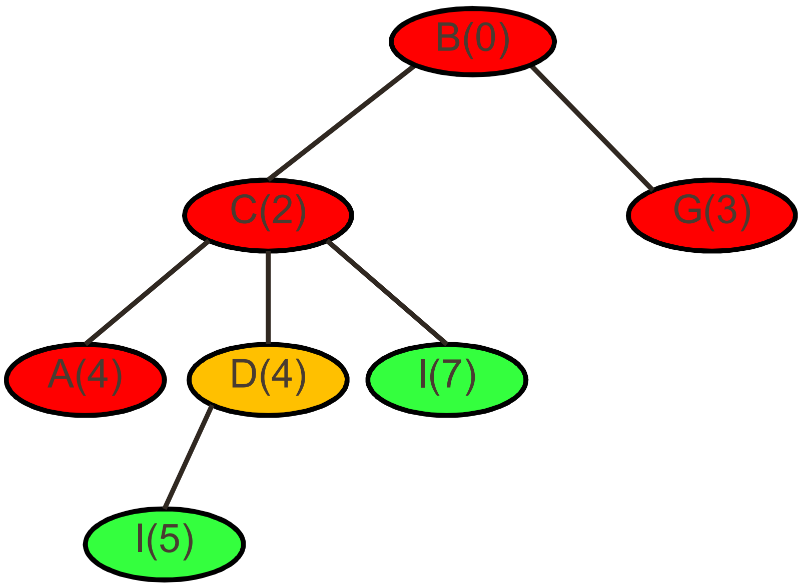


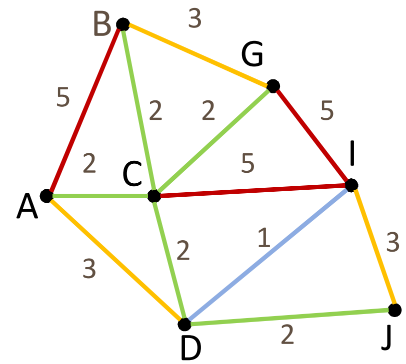


Open (Q):
{ D (4),
 I (7) }

Closed:
{ B (0),
 C (2),
 G (3),
 A (4) }

Dijkstra's Algorithm Example

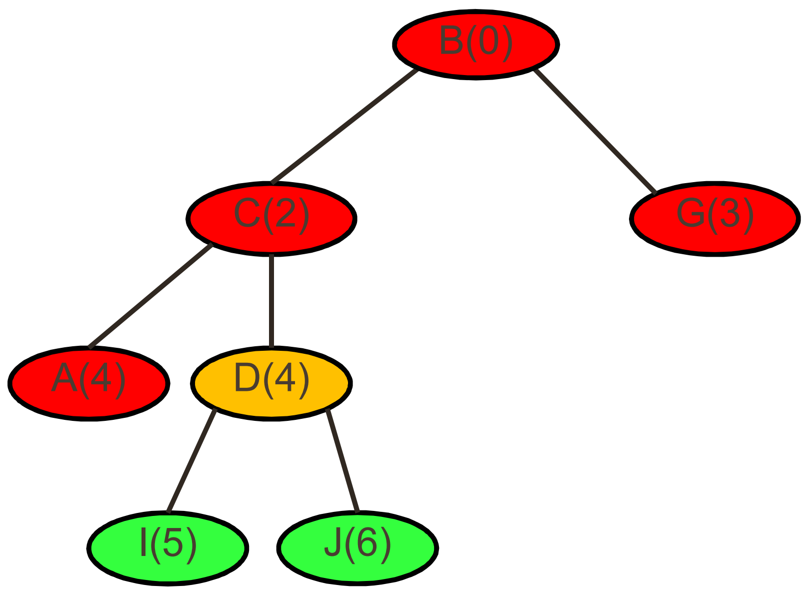


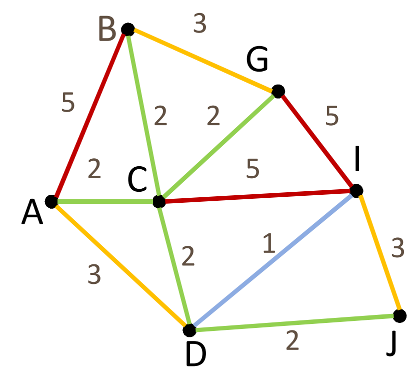


Open (Q):
{ I (5) }

Closed:
{ B (0),
C (2),
G (3),
A (4),
D (4) }

Dijkstra’s Algorithm Example

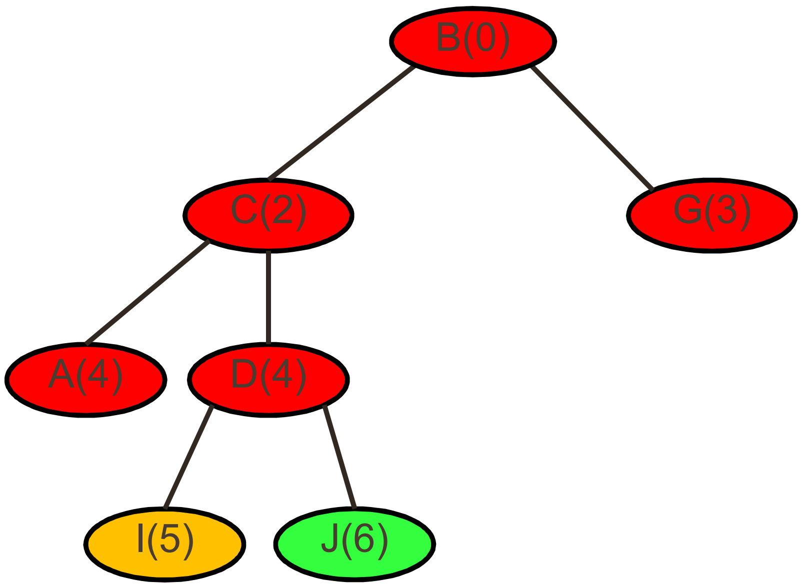


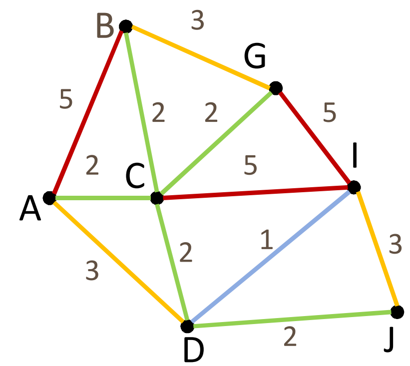


Open (Q):
{ I (5) ,
J (6) }

Closed:
{ B (0),
C (2),
G (3),
A (4),
D (4) }

Dijkstra's Algorithm Example

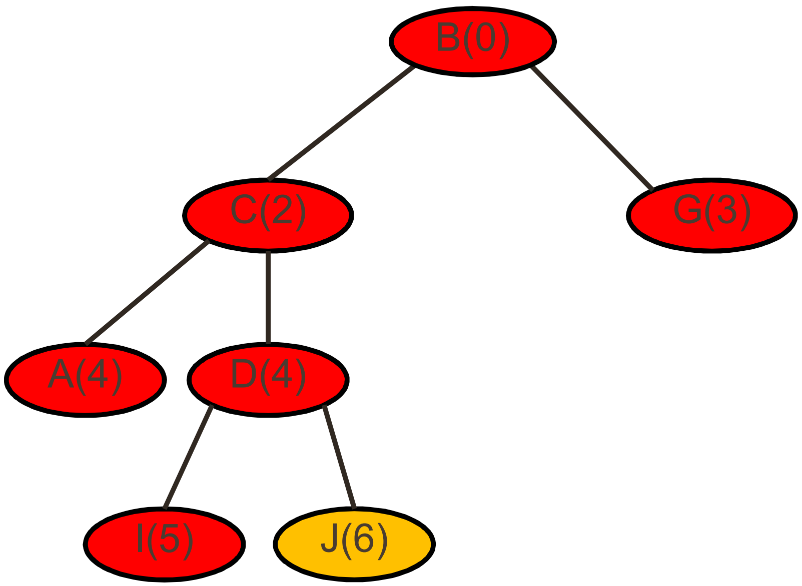


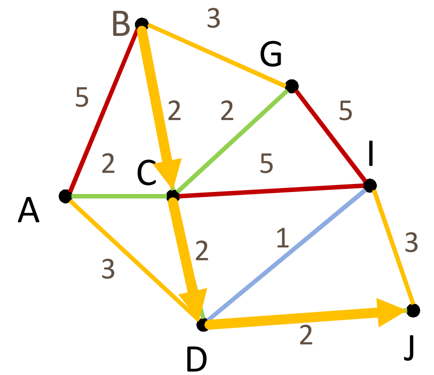


Open (Q):
{ J (6) }

Closed:
{ B (0),
C (2),
G (3),
A (4),
D (4),
I (5) }

Dijkstra's Algorithm Example





Open (Q):
{ }

Closed:
{ B (0),
C (2),
G (3),
A (4),
D (4),
I (5),
J (6) }

Final path solution: B→C→D→J
with path cost 6

Dijkstra's Algorithm

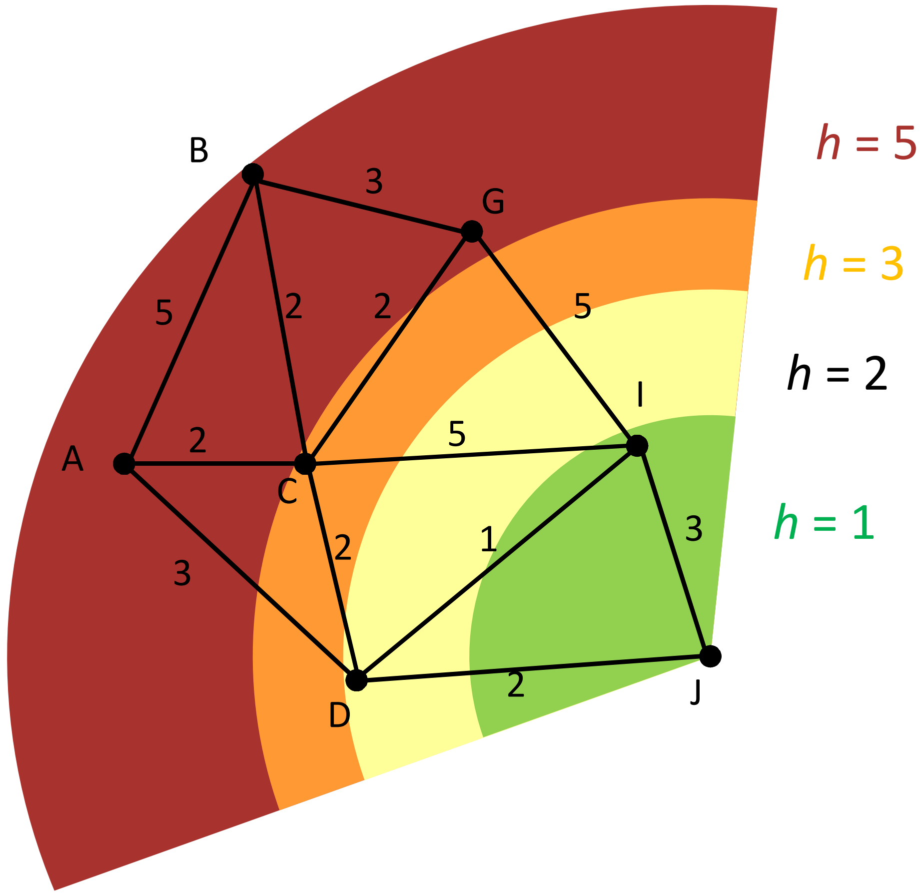
- At the end, we can recover the lowest-cost route from the start to any node (or any node with cost < goal if we terminate at a goal)
- Quite easy to implement, but requires a little bit of careful management with the priority queue
- Doesn't really know the goal exists until it reaches it
  - Could we guide the search to expand nodes that are closer to the goal earlier?
  - Can we do it without breaking the condition that a node is only accepted with its lowest cost of arrival?

A* Heuristic Search

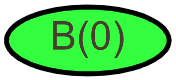
- Heuristic:
  - Any *optimistic estimate* of how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance

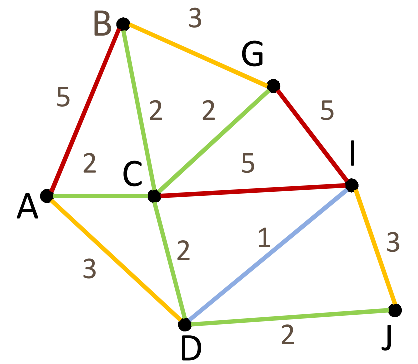
- A* Priority: $$f(n) = \underbrace{g(n)}_{\text{Cost to arrive}} + \underbrace{h(n)}_{\text{Heuristic cost to goal}}$$

A* Heuristic



A* Algorithm Example





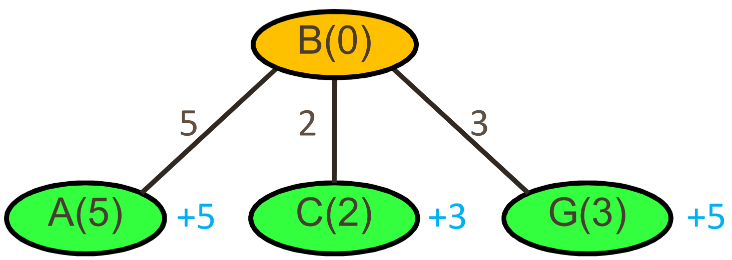
Open (Q): {B(0)}

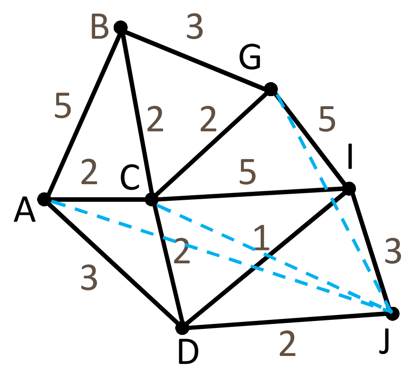
Closed: {B(0)}

Our A* queue will be ordered by cost to arrive + heuristic:

- push (*Q.Insert*) by A* priority, *f(n)*
- pop (*Q.GetFirst*) from the front, and add it to the closed list

A* Algorithm Example

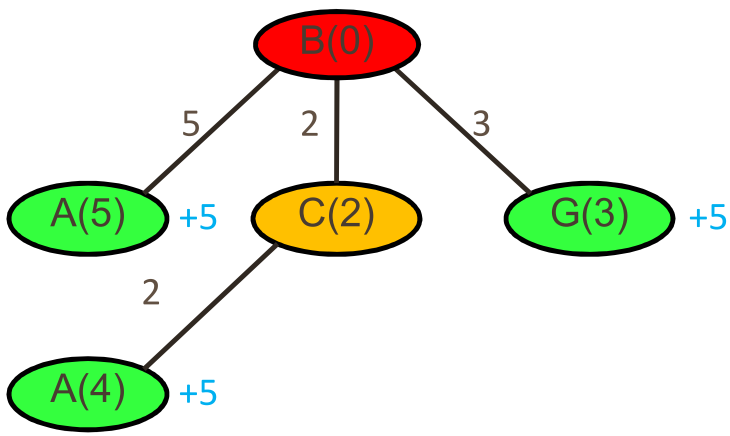


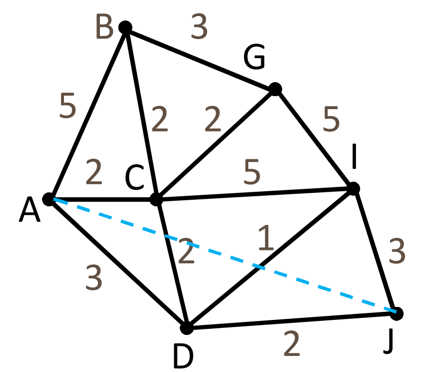


Open (Q):
{ C (2+3),
G (3+5),
A (5+5) }

Closed:
{ B (0) }

A* Algorithm Example

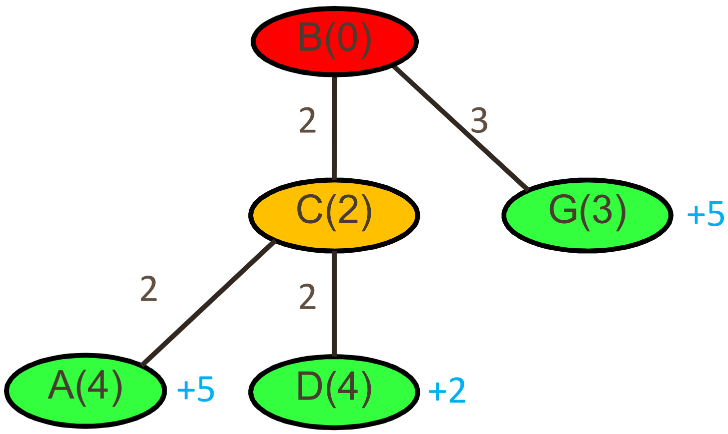


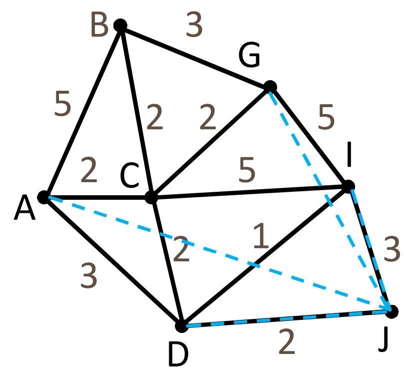


Open (Q):
{ G (3+5),
A (4+5) }

Closed:
{ B (0) ,
C (2) }

A* Algorithm Example

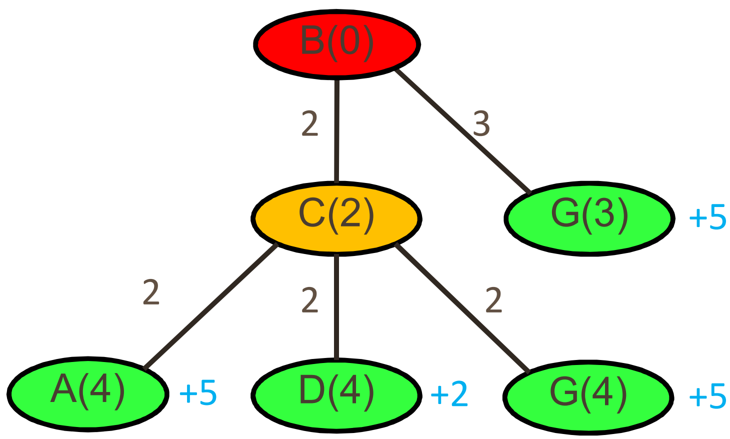


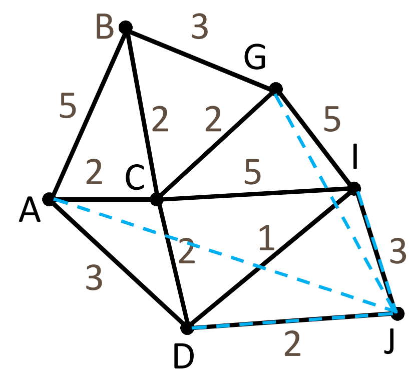


Open (Q):
{ D (4+2),
G (3+5),
A (4+5) }

Closed:
{ B (0) ,
C (2) }

A* Algorithm Example

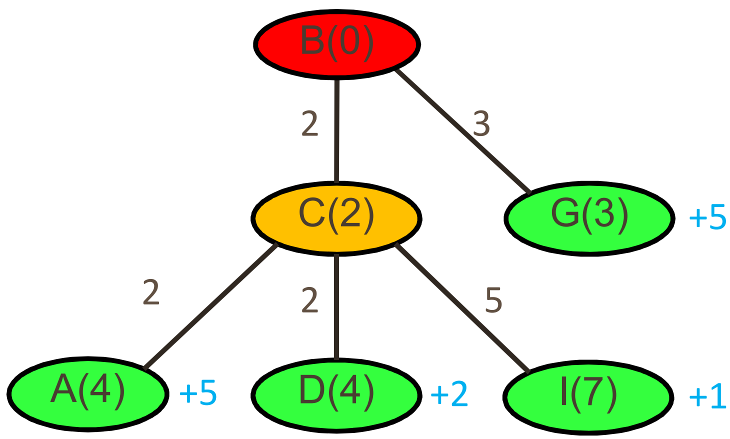


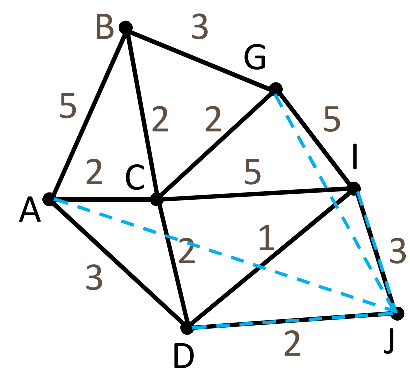


Open (Q):
{ D (4+2),
G (3+5),
A (4+5) }

Closed:
{ B (0) ,
C (2) }

A* Algorithm Example

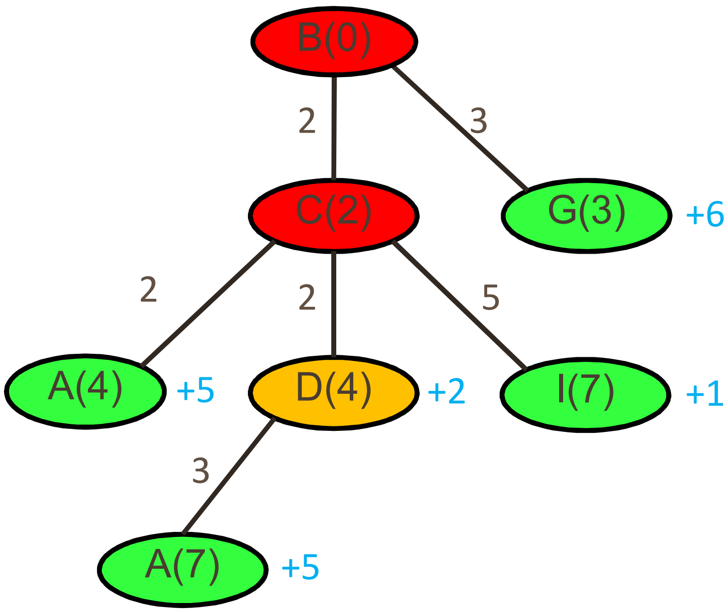


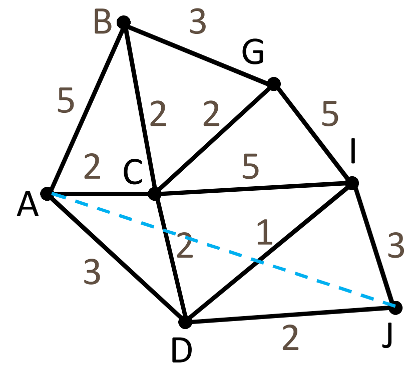


Open (Q):
{ D (4+2),
I (7+1),
G (3+5),
A (4+5) }

Closed:
{ B (0) ,
C (2) }

A* Algorithm Example

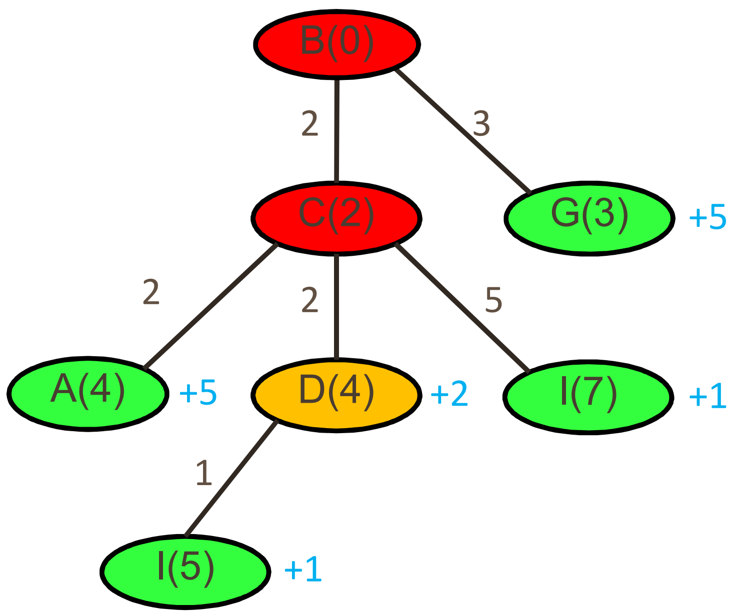


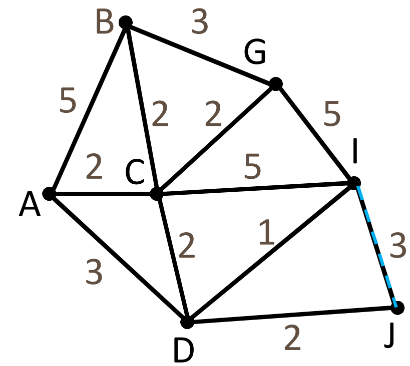


Open (Q):
{ I (5+1),
G (3+5),
A (4+5) }

Closed:
{ B (0) ,
C (2),
D (4) }

A* Algorithm Example

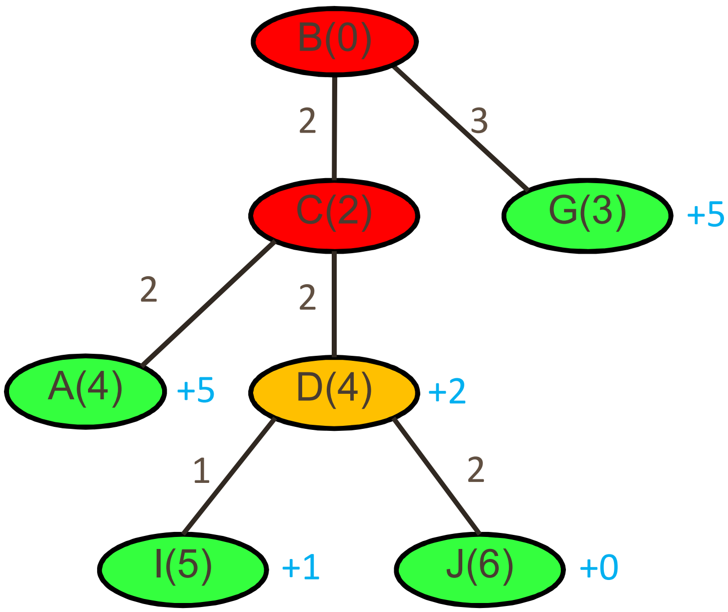


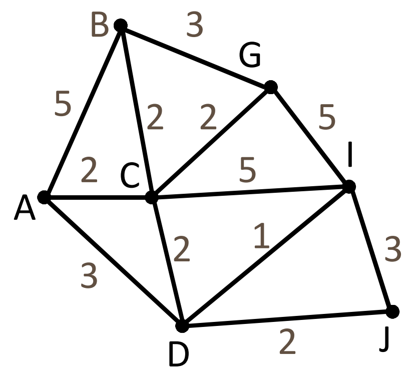


Open (Q):
{ I (5+1),
G (3+5),
A (4+5) }

Closed:
{ B (0) ,
C (2),
D (4) }

A* Algorithm Example

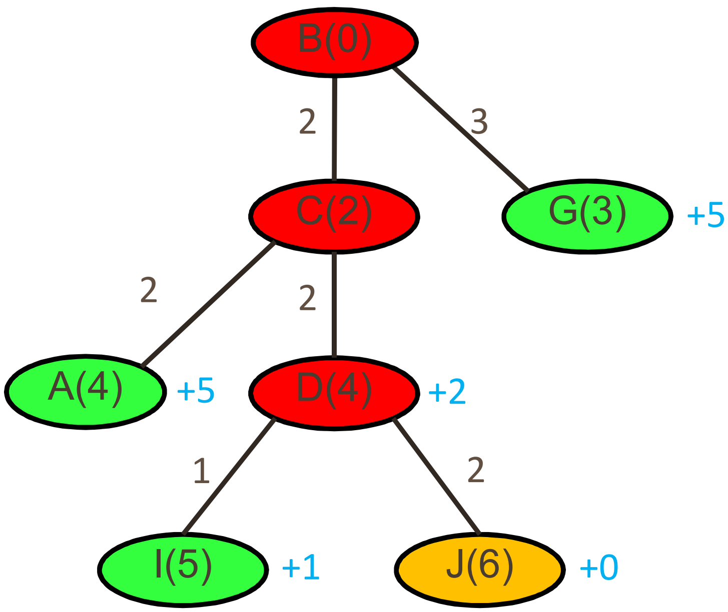


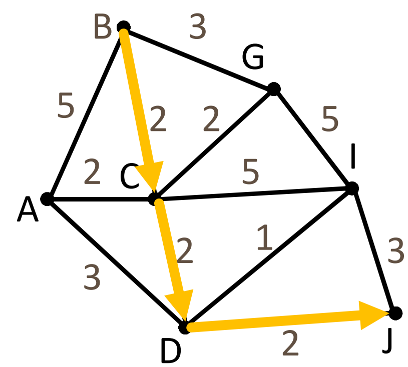


Open (Q):
{ J (6+0),
I (5+1),
G (3+5),
A (4+5) }

Closed:
{ B (0) ,
C (2),
D (4) }

A* Algorithm Example





Open (Q):
{ I (5+1),
G (3+5),
A (4+5) }

Closed:
{ B (0) ,
C (2),
D (4),
J (6) }

Final path solution: B→C→D→J
with path cost 6

A* Heuristic

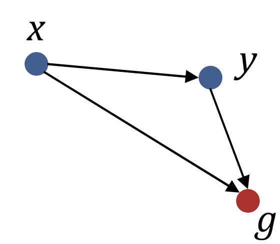
- The heuristic must be ***admissible***
  - It never overestimates the cost

True cost to goal

$$h(x) \leq d(x, goal)$$

- The heuristic must be ***consistent***
  - For any pair of adjacent nodes x and y, where $d(x,y)$ is the cost of edge between them

$$h(x) \leq d(x, y) + h(y)$$



- Typical valid heuristics:
  - Euclidean distance
  - Manhattan distance
  - Zero (Dijkstra's algorithm)

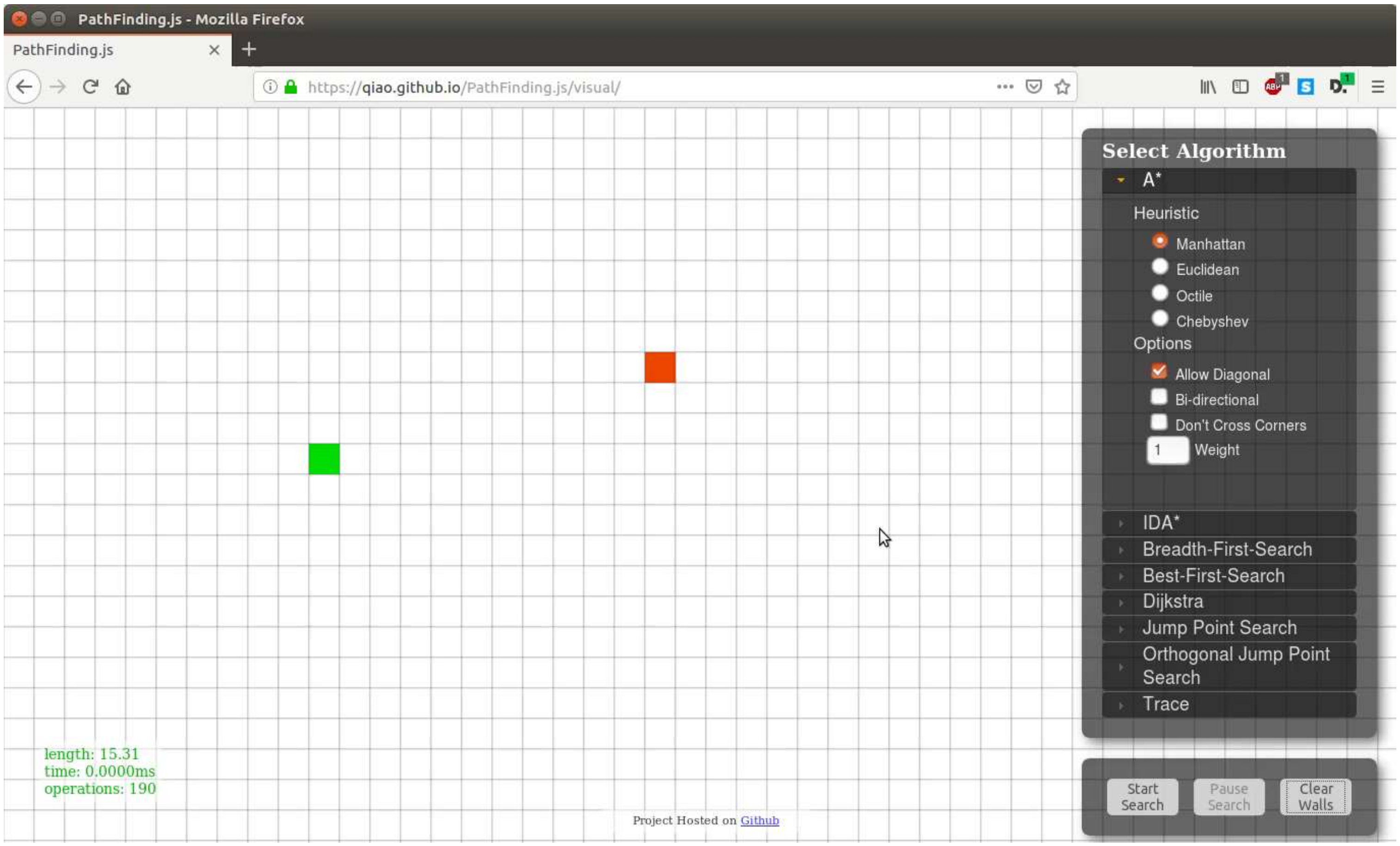
A* Search Algorithm

- A* is an extension of Dijkstra's algorithm, and achieves faster performance by using heuristics

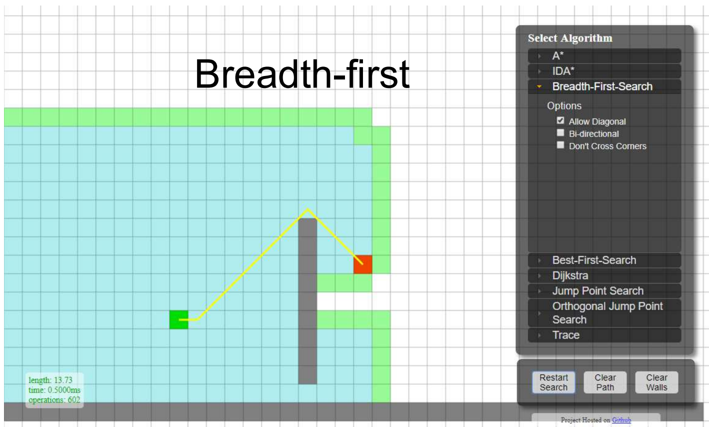
- **Best-first search**: A* traverses a graph following a path of lowest expected total cost or distance

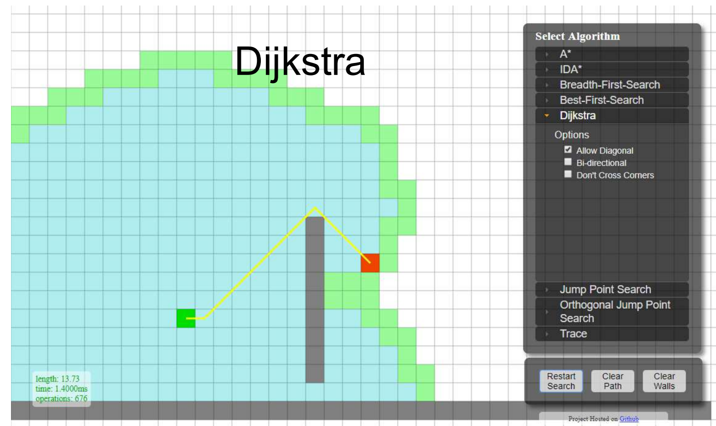
- The cost function is a sum of two functions:
  - Past path-cost function, which is a **known** cost from the starting node to the current node
  - Future path-cost function, which is a "heuristic estimate" of the distance from the current node to the goal

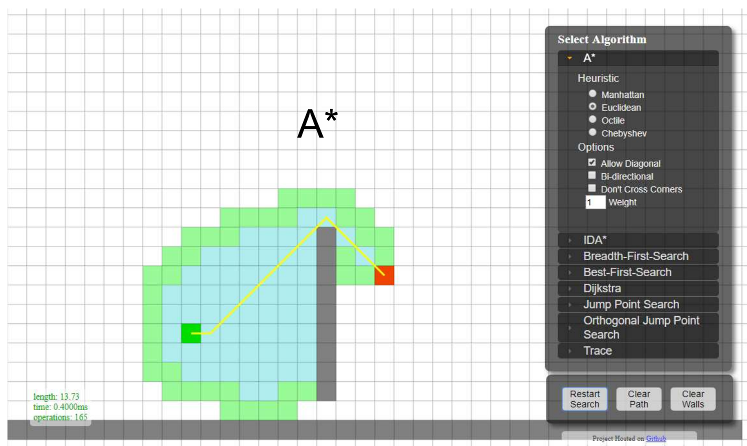
https://qiao.github.io/PathFinding.js/visual/



https://qiao.github.io/PathFinding.js/visual/



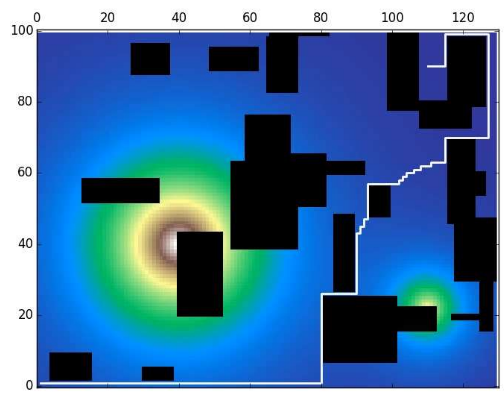


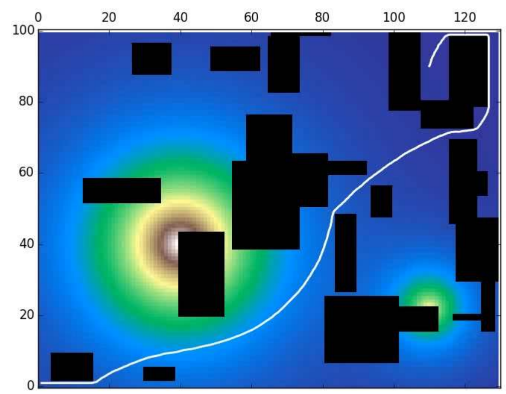


Limitations

- A* is very commonly used in robot planning, especially for low-dimensional state spaces

- Limitations:
  - You need to construct a graph
  - Sometimes an admissible heuristic function is difficult to find (as hard as the problem)
  - A grid may not be a good representation of your problem





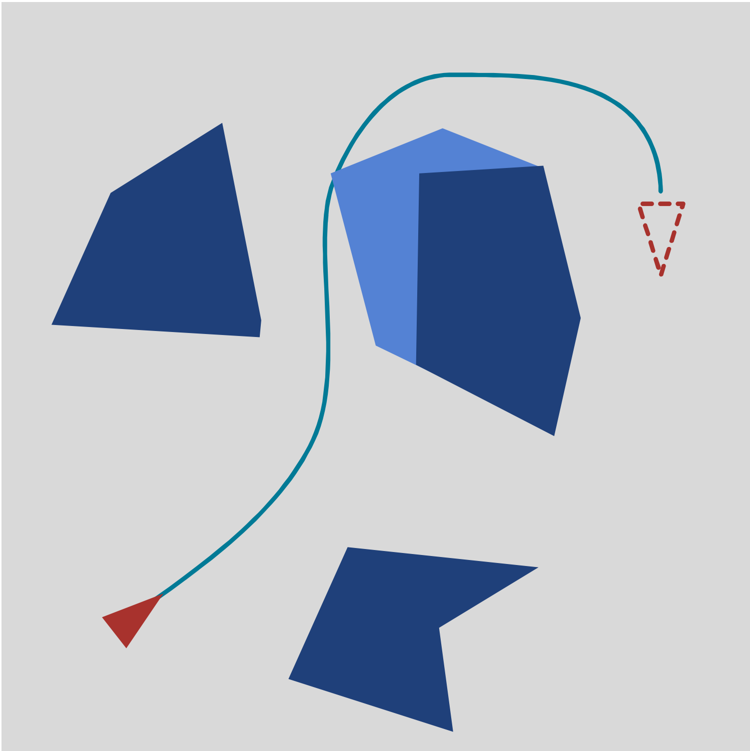
What happens when it doesn't all go to plan?

- So far, we've basically ignored the plan **execution**

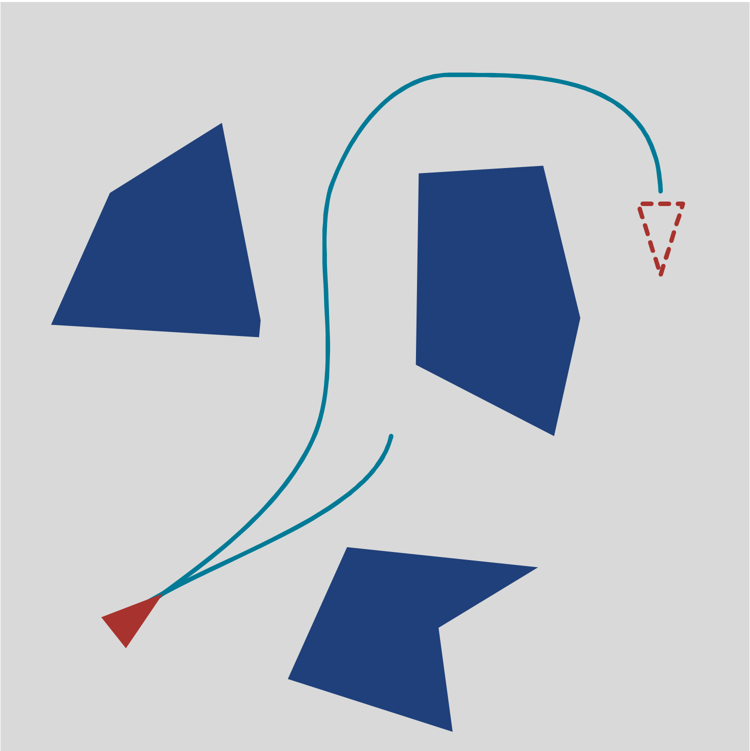
- Our plans are a series of states that we assume the robot is capable of visiting sequentially and reliably

- However, there are many reasons that, in practice, this may not always be the case

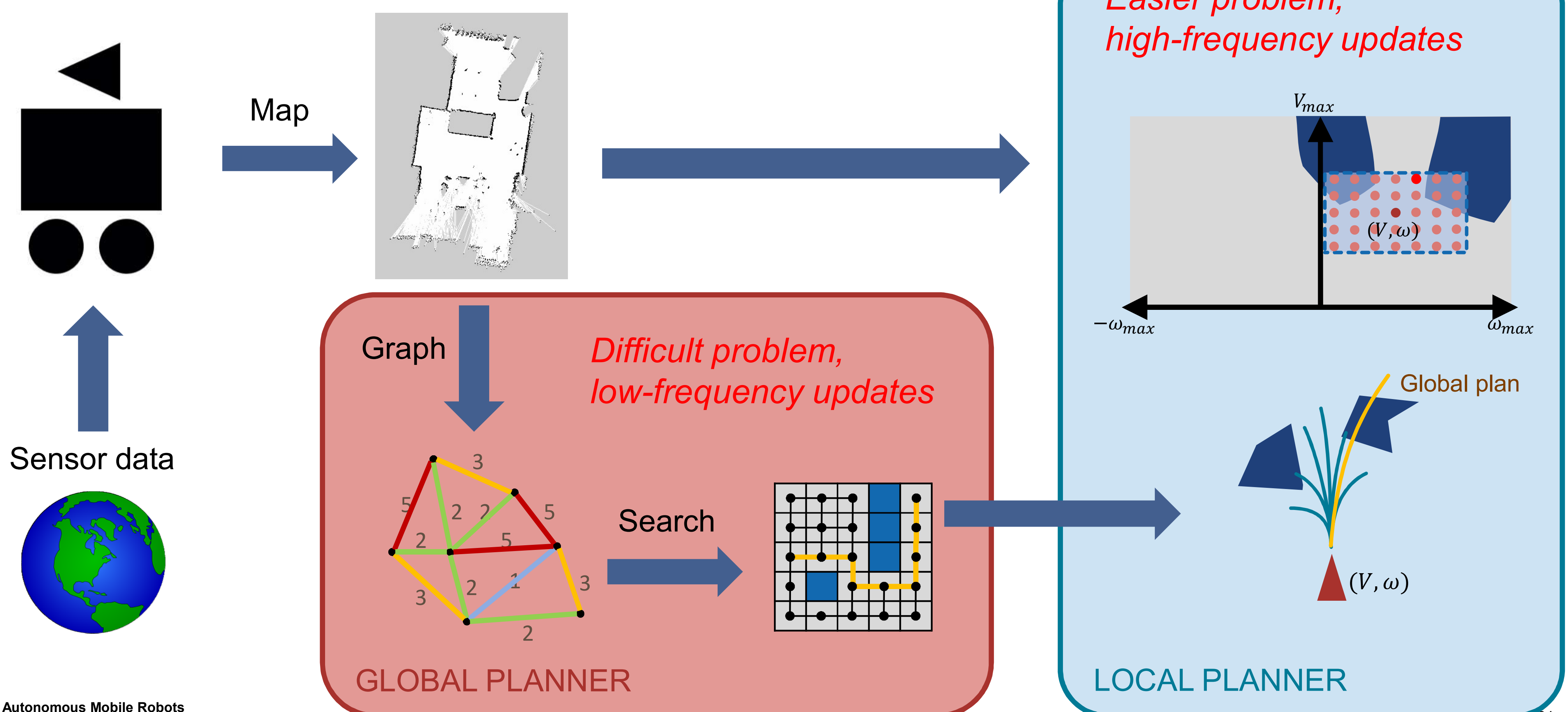
- Environment uncertainty



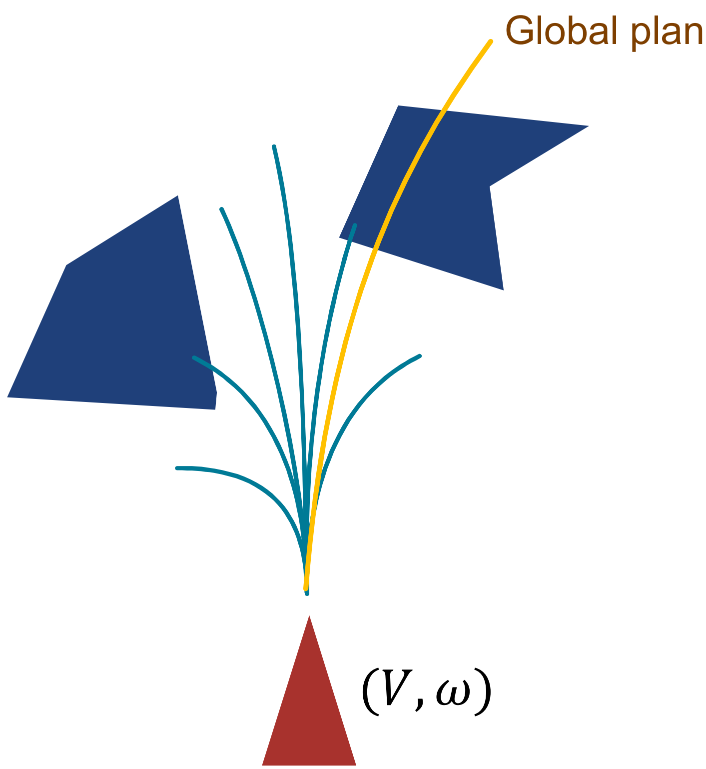
- Motion uncertainty



Planning hierarchy



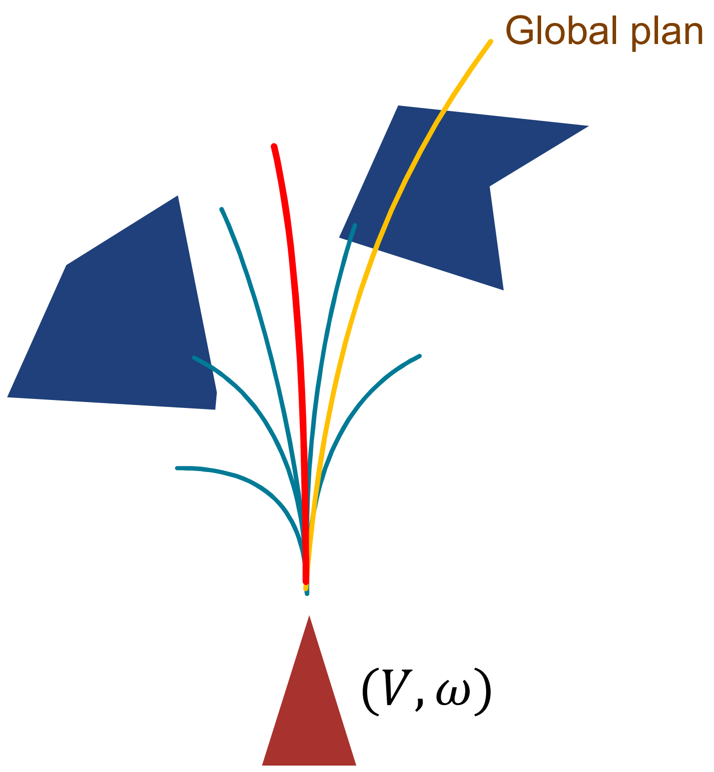
Dynamic Window Approach Collision Avoidance

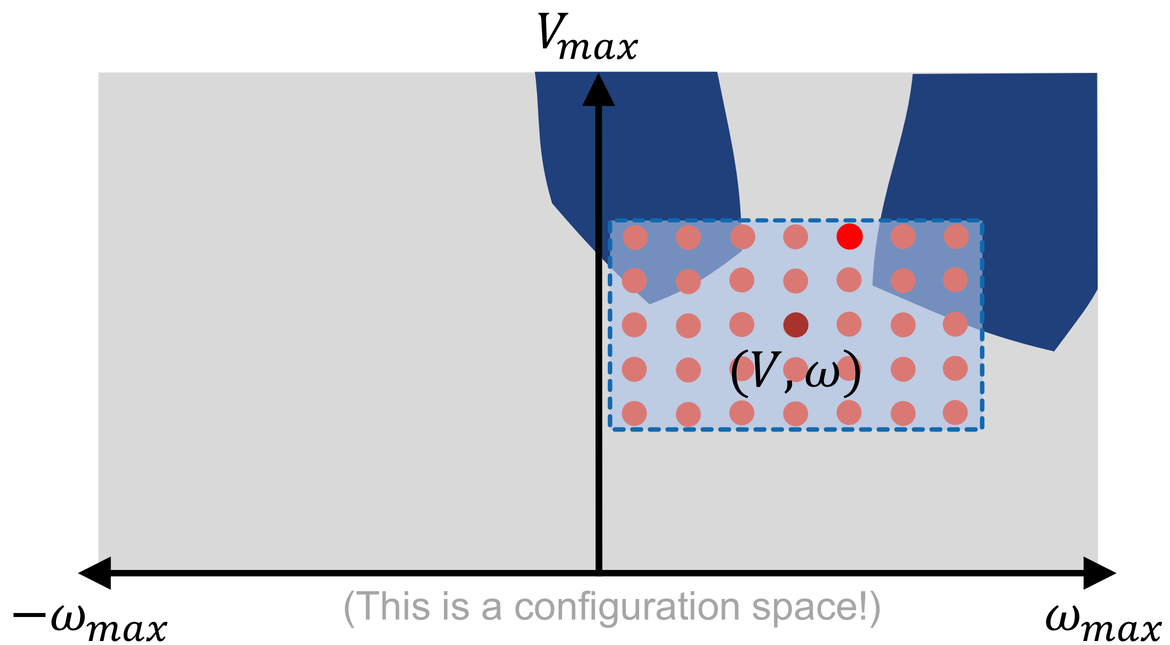


- Generating full time-varying trajectories for $V(t)$ and $\omega(t)$ is still very challenging
- If we assume (V, ω) are constant for a fixed Δt, each local path in the future is a circular arc segment
- This can be easily considered as a type of *velocity configuration space*

Fox, D., Burgard, W., & Thrun, S. (1997). The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, *4*(1), 23-33.

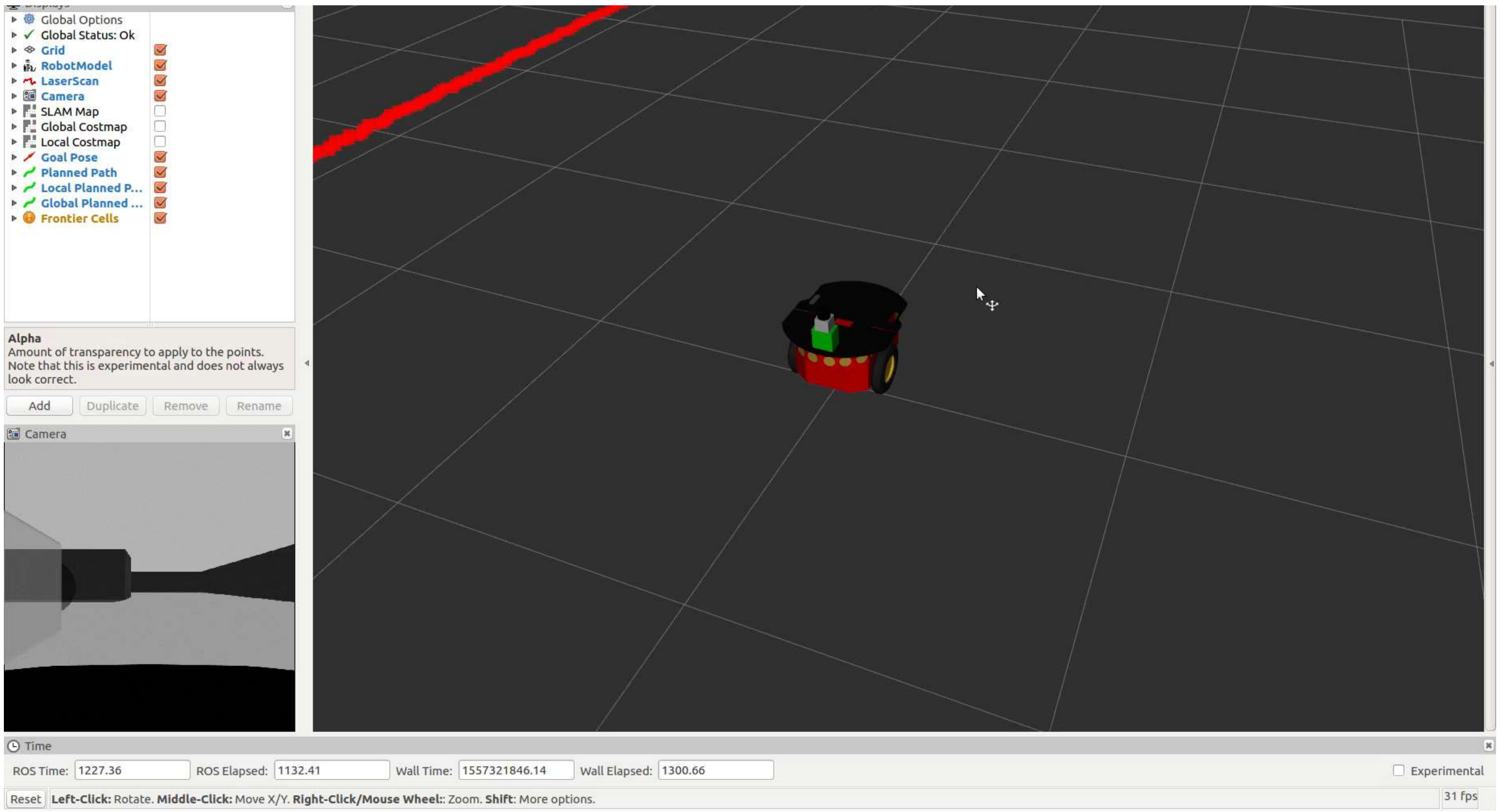
Dynamic Window Collision Avoidance





- Maximise utility metric (usually maximise speed, minimise distance to goal, maximise distance from obstacles) across configuration samples

ROS Navigation Stack *(Dijkstra global + DWA local)*



Collision Avoidance

- Sometimes, you may prefer to avoid the hierarchy and consider both global and local planning simultaneously

- A conceptually simple and relatively common approach for this type of problem are **potential field** methods

Potential Field Methods – Global potential

- Imagine we had a function that related an **elevation** with some kind of distance to the goal (a ‘potential’ function, as in potential energy)

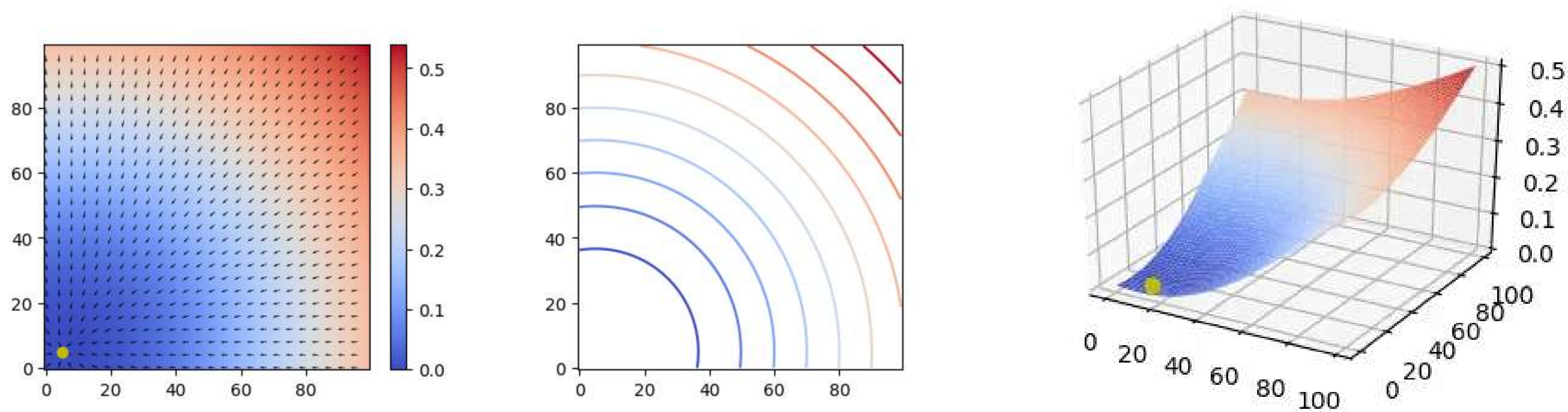
- By taking control actions that direct the robot in the direction of **maximum gradient (down),** the robot should ‘fall’ towards the goal (minimum energy state)

- The global potential function should ‘attract’ the robot towards the goal, from any valid state

Potential Field Methods – Global potential

- We want a smooth, differentiable function so that it is easy to calculate the target vector

$$U_{goal}(x) = \begin{cases} \frac{1}{2}\zeta\left\|x - x_{goal}\right\|^2, & \left\|x - x_{goal}\right\| < d^* \\ d^*\zeta\left(\left\|x - x_{goal}\right\|^2 - \frac{1}{2}d^*\right), & otherwise \end{cases}$$

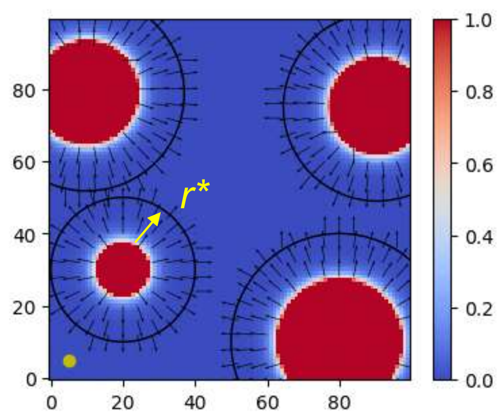


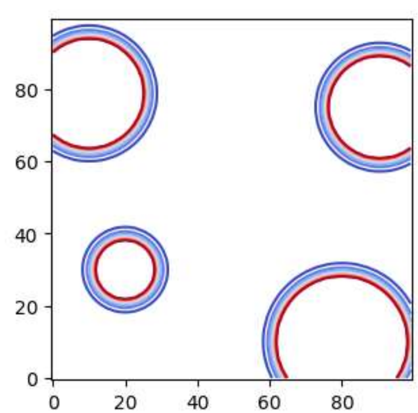
Potential Field Methods – Obstacles

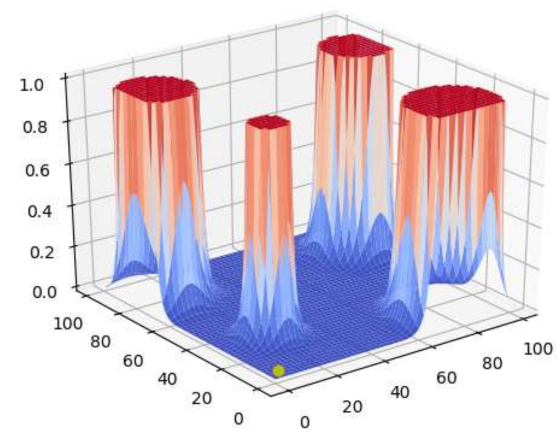
- We also want to avoid obstacles, so we add an additional component that ‘repels’ from obstacles

$$U_{obs}(x) = \begin{cases} \frac{1}{2}\eta\left(\frac{1}{D(x)} - \frac{1}{r^*}\right)^2, & D(x) \le r^* \\ 0, & otherwise \end{cases}$$

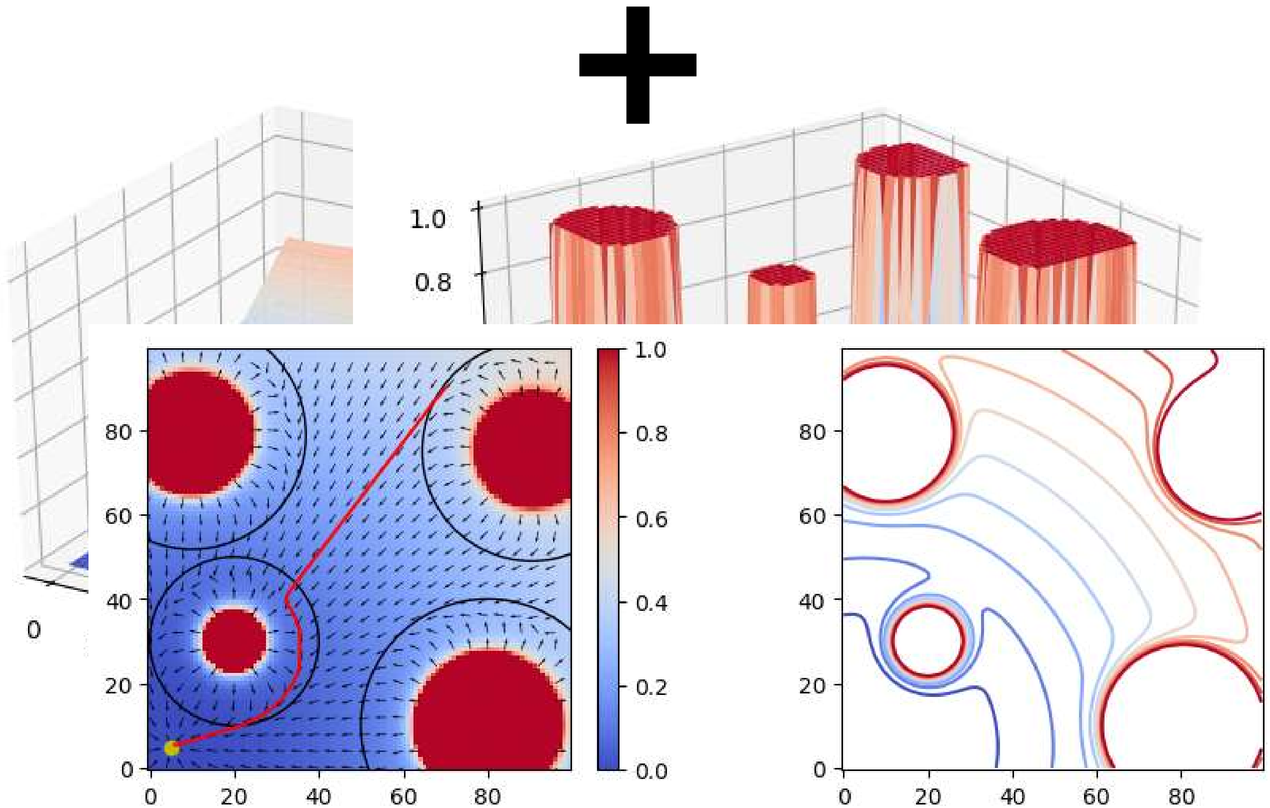
*$D(x)$ is the distance to the nearest obstacle boundary



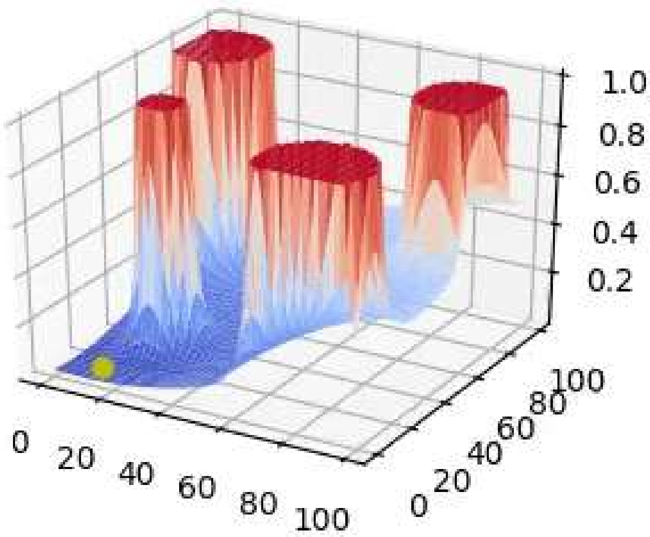


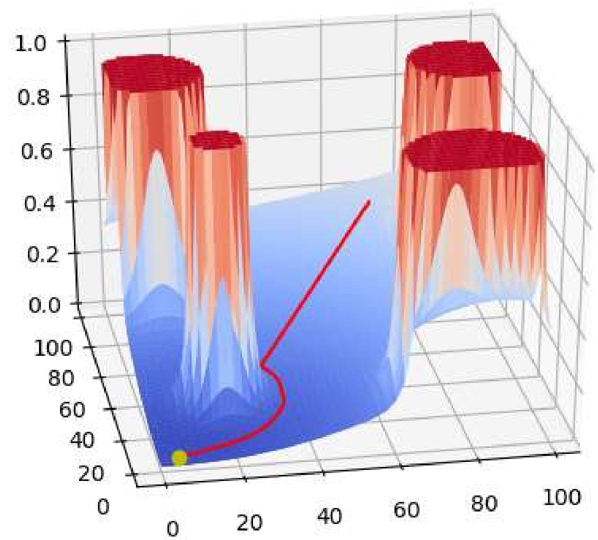


Potential Field

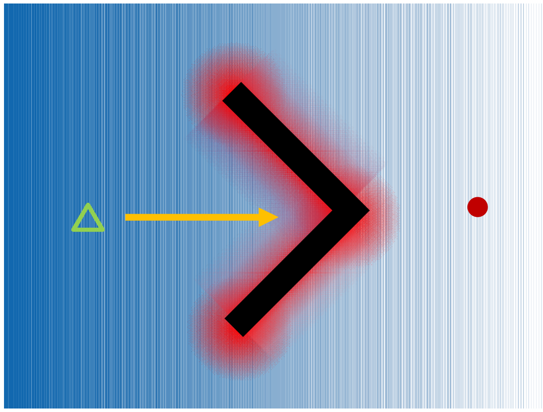


Potential Field





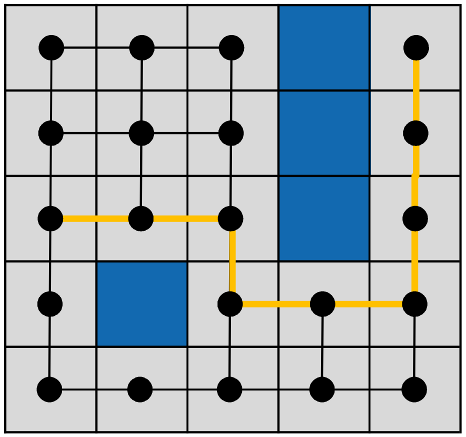
Potential Field Methods



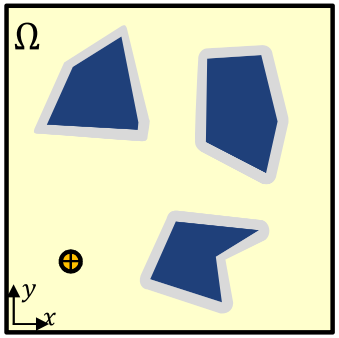
- Relatively simple to implement
- Simplest versions can have issues with stationary points or local minima
- Modifying the potential functions can allow these conditions to be avoided (harmonic potentials, homework!)
- Dealing with higher-order state spaces (arms etc.) can be difficult to grasp/visualise, but potential functions are applicable

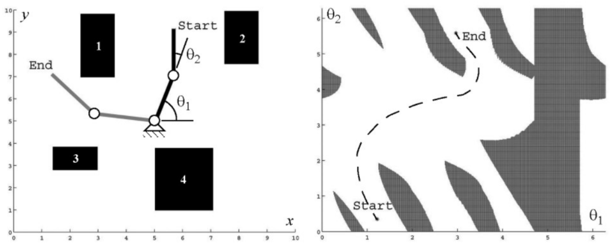
Summary

- Motion planning
  - Representation – how to define the robot's understanding of the world, and ensure that it is sufficient to complete the task



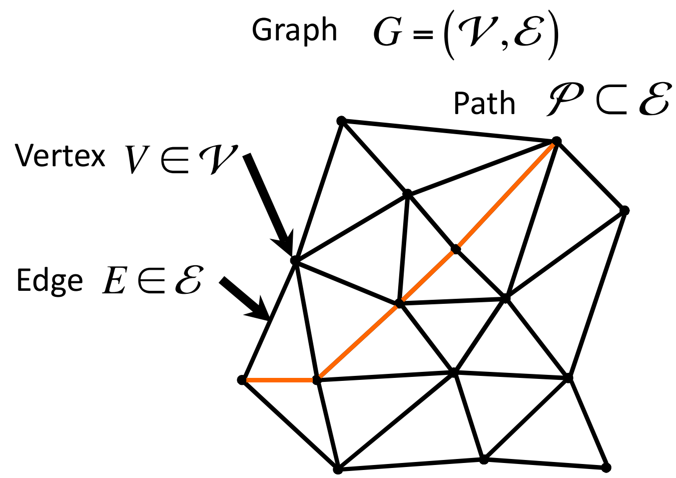
  - Work space – the world without the robot
  - Configuration space – the robot's configuration (joint angles etc.) in the world





Summary

- Graph search methods
  - Graphs are constructions of vertices and connecting edges
  - Graph search techniques are used to find low-cost paths through graphs
  - Breadth-first and depth-first search – complete searches from start (unweighted graphs)
  - Djikstra – search outwards in order of cost from start (weighted graphs)
  - A* – focused search that prioritises searching towards the goal using an admissible heuristic



Summary

- Hierarchical planning
  - Global planner over the whole search space
  - Local planner to respond to changes in environment, avoid collisions, stay on global path
- Potential fields
  - Design a function such that descending the gradient leads to a collision-free path to the goal
- Additional references
  - Course text book and online lectures
  - Howie Choset (CMU) motion planning lecture notes:
    - https://www.cs.cmu.edu/~motionplanning/lecture/Chap4-Potential-Field_howie.pdf
  - Steven LaValle's Planning Algorithm Textbook
    - http://planning.cs.uiuc.edu/