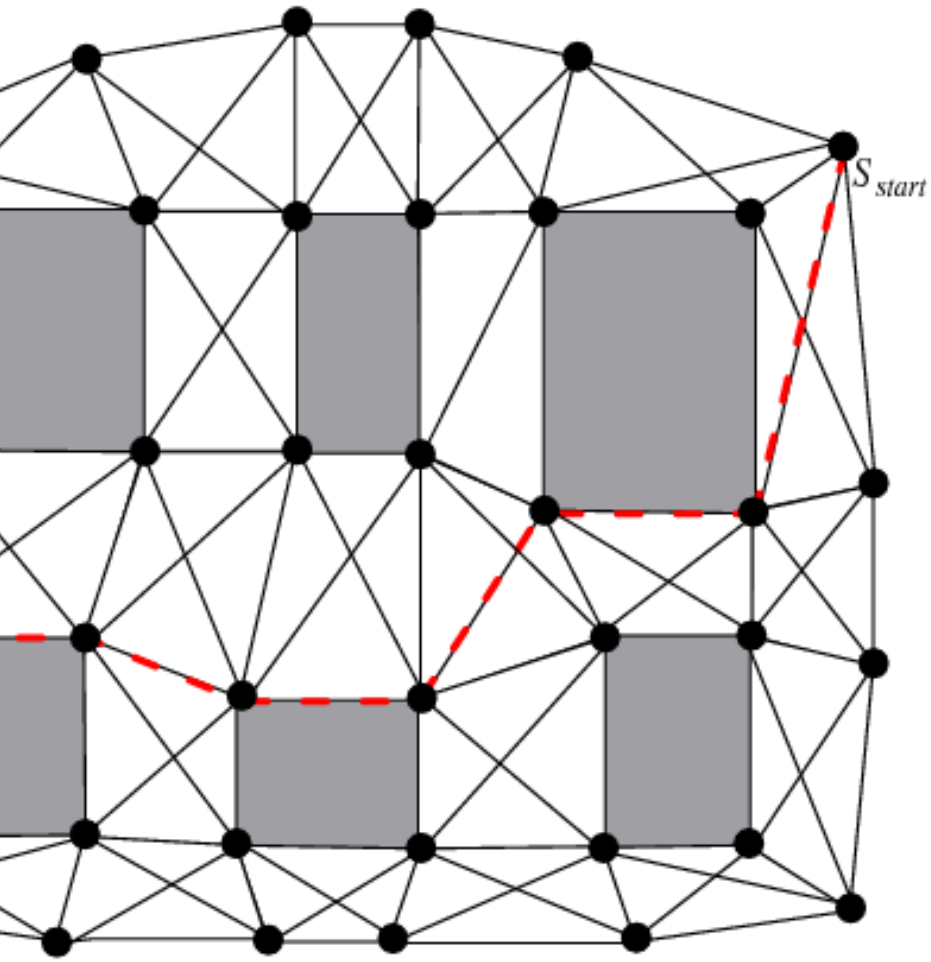
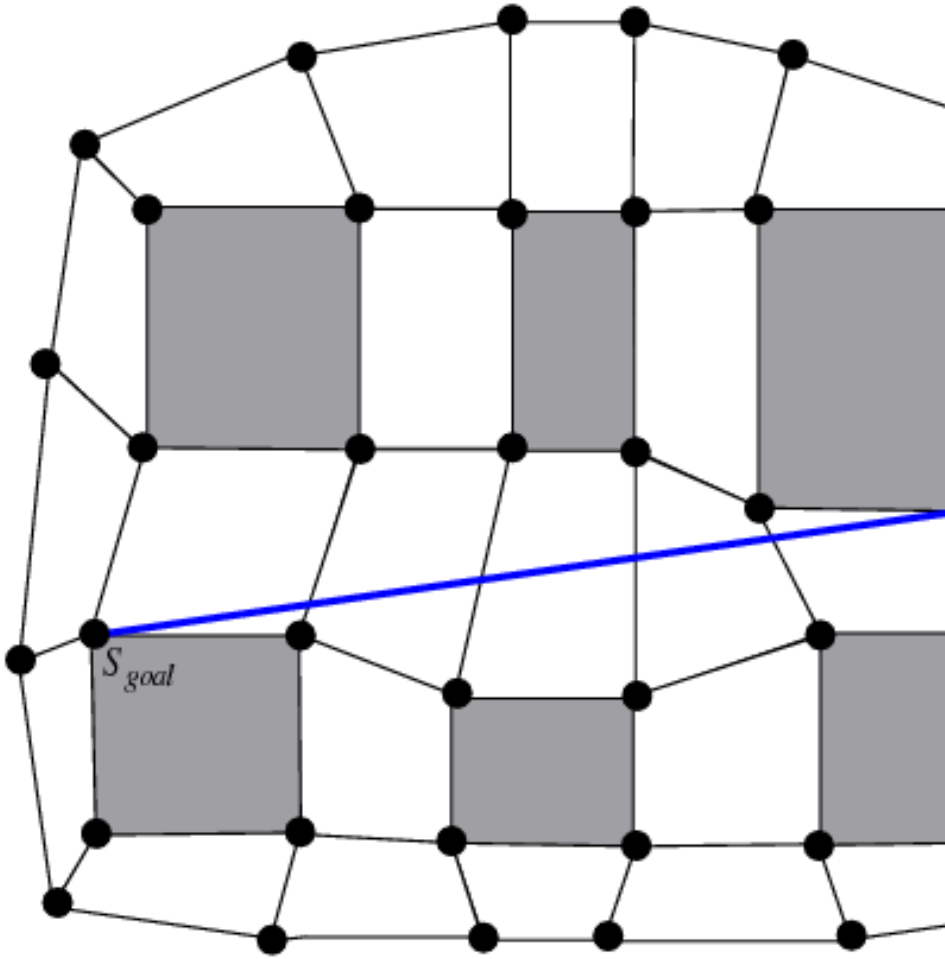




INTRODUCTION TO SEARCH

ECS170 Spring 2018
Josh McCoy, @deftjams

Why Search?

R I K L S

Strong statements:

All AI problems can be seen as search.

All AI techniques can be seen as search.

Why search?

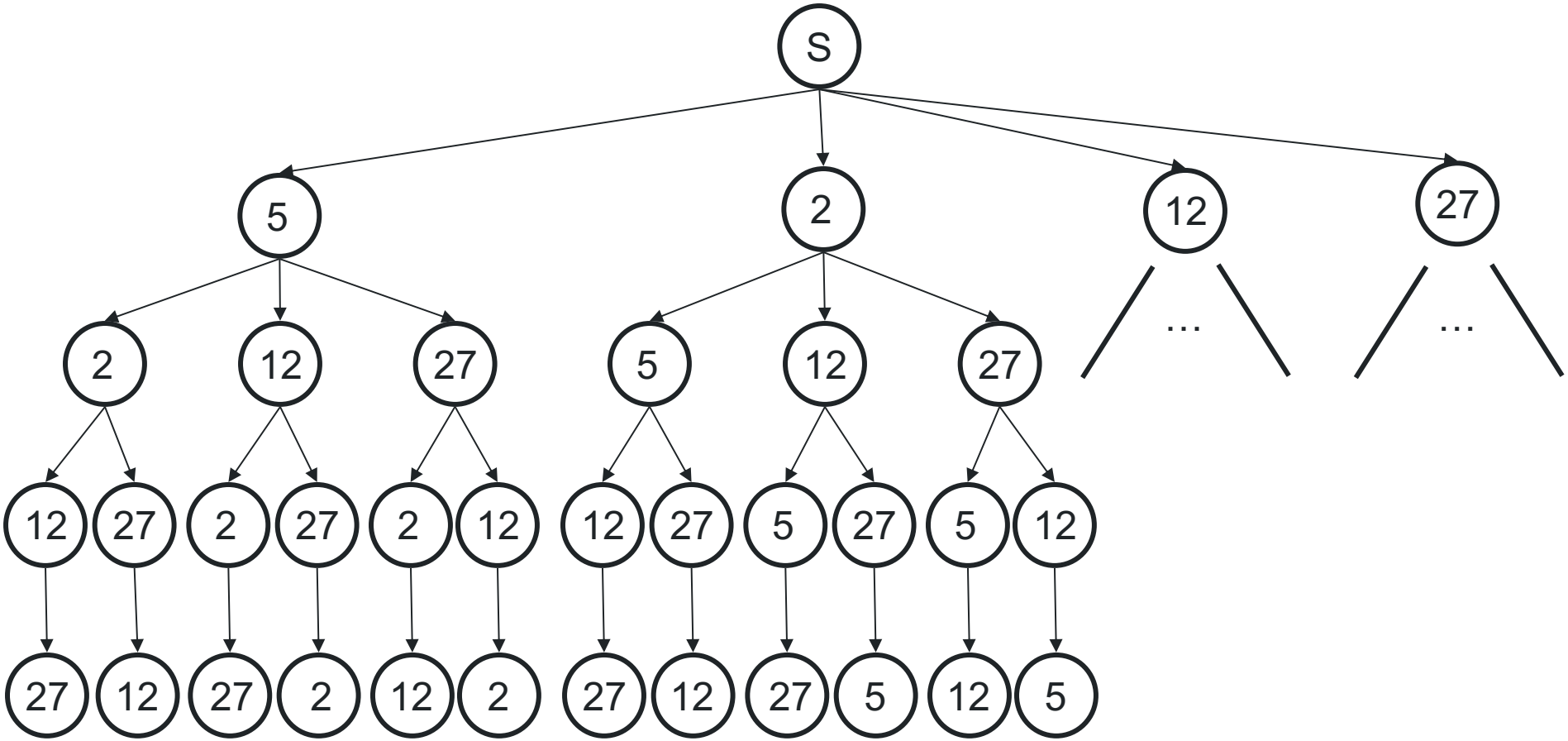
Consider this problem – given a set of integers, we want to output them in sorted order, smallest number first

- Say we want to sort {5, 2, 27 12} – the answer is {2, 5, 12, 27}

First of all, do we have a quick way of checking an answer? Yes.

```
currentNumber := first element of SA (SA is a list containing the answer)
while (not at the end of SA) do
        nextNumber := the next element in SA
        if (currentNumber > nextNumber) signal no
        else currentNumber := nextNumber
signal yes
```

So here's a way to sort



Is this a good way to sort numbers?

Intriguing facts about computation

There exist problems which can be solved in a relatively small number of computations, in fact a *polynomial number* (P), like sort

There exist (many many interesting) problems which can not be solved in a polynomial number of computations – these are *non-deterministic polynomial* complete (NP-complete) problems

The NP-complete problems are the ones that can only be solved by search

Formalizing Search

From the book:

- Initial State
- Actions
- Transition Model
- Goal Test
- Path Cost

MadLibs Search

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Next Level of Detail

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Formalizing search

A search problem has five components: ***Q*** , ***S, G***, **succs**, **cost:**

Q is a finite set of states.

$\boldsymbol{S} \subseteq Q$ is a non-empty set of start states.

$\boldsymbol{G} \subseteq Q$ is a non-empty set of goal states.

succs : $Q \rightarrow P(Q)$ is a function which takes a state as input and returns a set of states as output. **succs**(*s*) means "the set of states you can reach from *s* in one step".

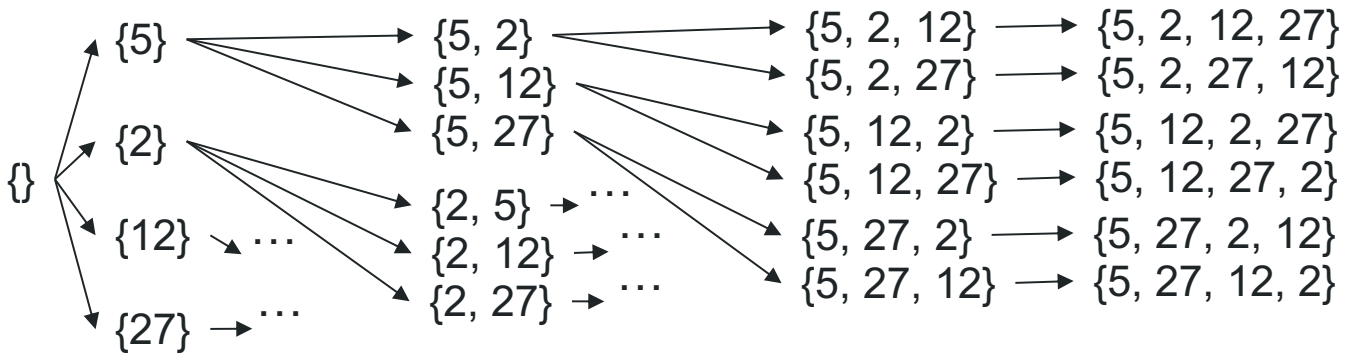
cost : Q , $Q \rightarrow$ *Positive Number* is a function which takes two states, *s* and *s'*, as input. It returns the one-step cost of traveling from *s* to *s'*. The cost function is only defined when *s'* is a successor state of *s*.

Representing the search space

Search space represented as a graph of state transitions

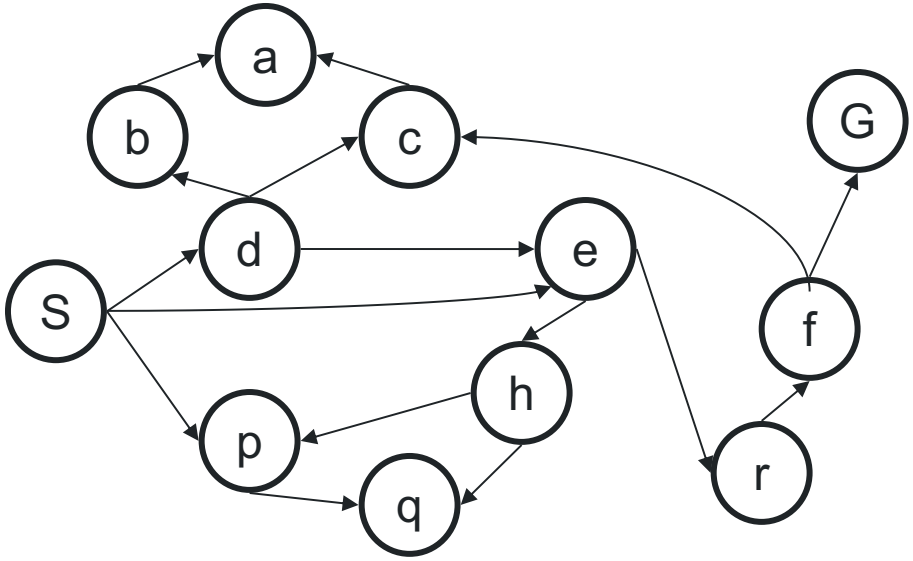
Two typical representations

- A graph of states, where arrows represent **succ** and nodes represent states – a state appears only once in the graph
- A tree of states, where a path in the tree represents a sequence of search decisions – the same state can appear many times in the tree



Here's our sort problem written out as a search graph

An abstract search problem



Q = {START, *a* , *b* , *c* , *d* , *e* , *f* , *h* , *p* , *q* , *r* , GOAL}
S = { S }
G = { G }
succs(*b*) = { *a* }
succs(*e*) = { *h* , *r* }
succs(*a*) = NULL … etc.
cost(*s*,*s'*) = 1 for all transitions

Some example search problems

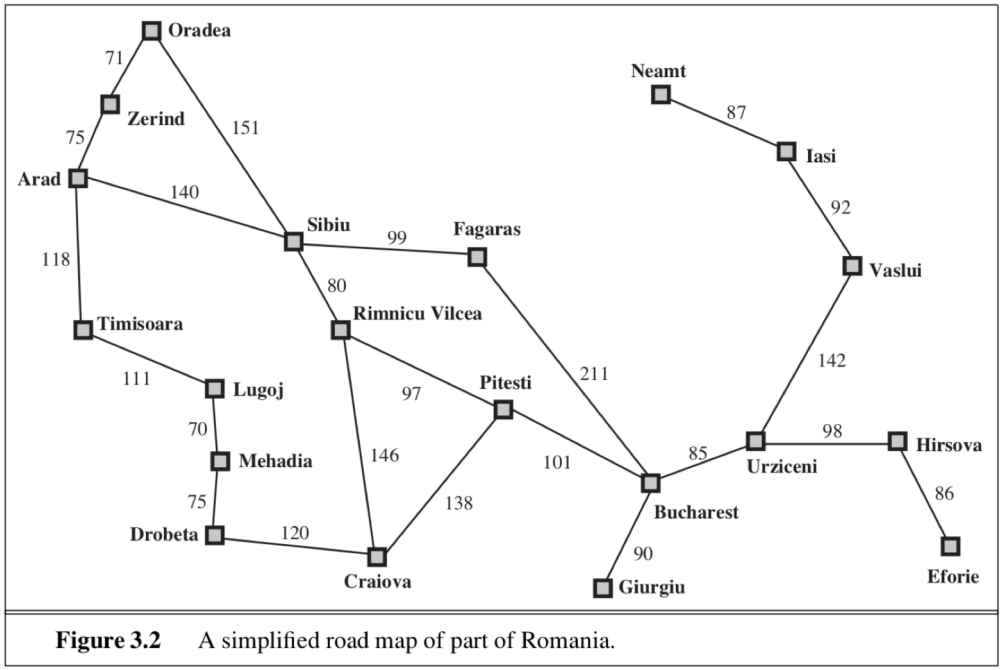
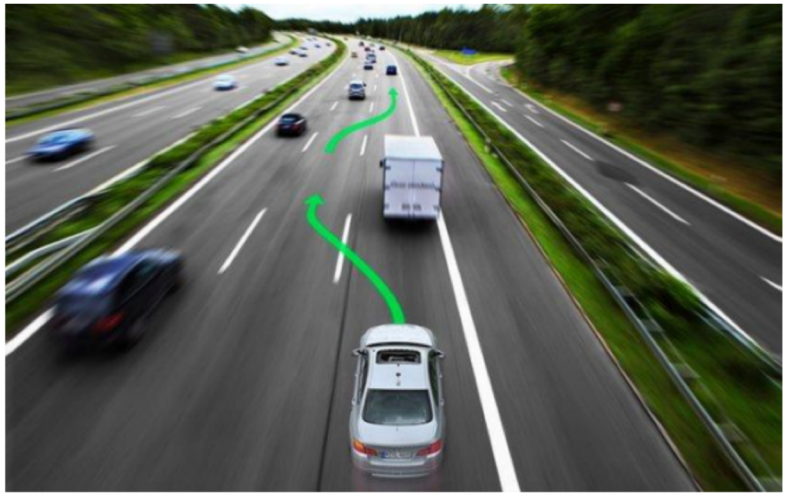
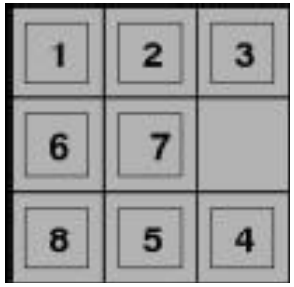


Figure 3.2 A simplified road map of part of Romania.





Aside: Why do we have to search?

- "We may regard the present state of the universe as the effect of its past and the cause of its future. An intellect which at any given moment knew all of the forces that animate nature and the mutual positions of the beings that compose it, if this intellect were vast enough to submit the data to analysis, could condense into a single formula the movement of the greatest bodies of the universe and that of the lightest atom; for such an intellect nothing could be uncertain and the future just like the past would be present before its eyes."

-Marquis Pierre Simon de Laplace

Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

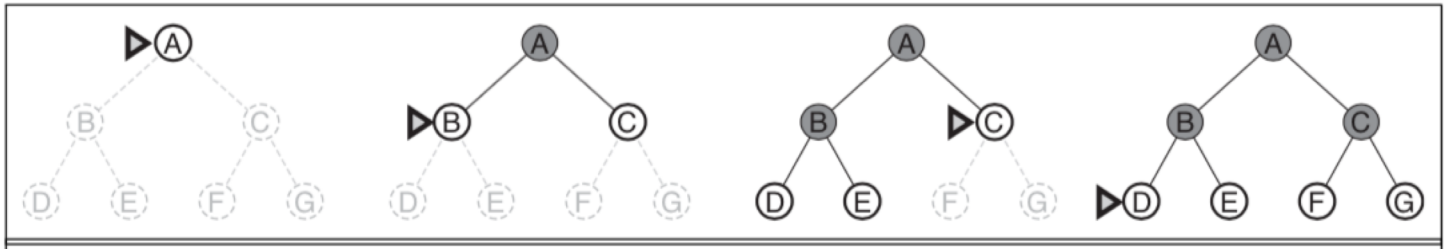
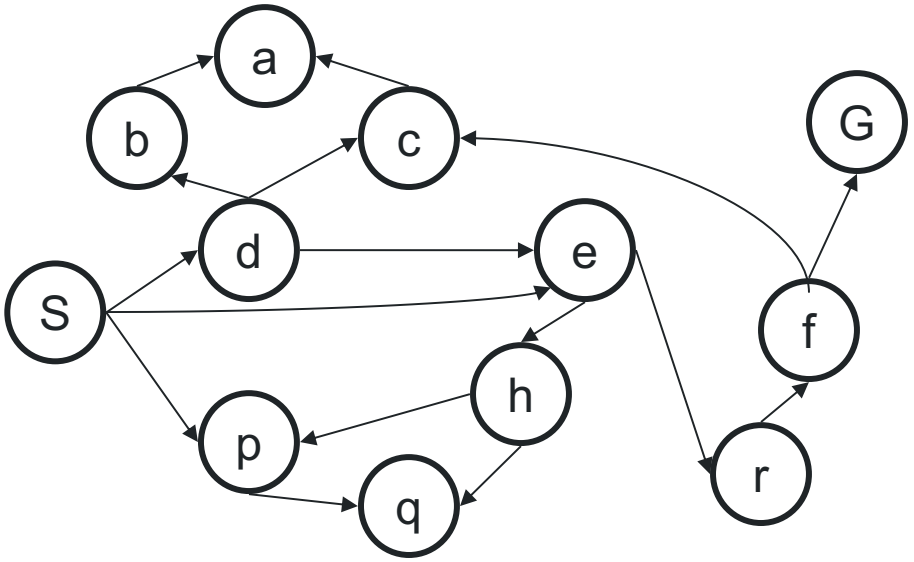


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Breadth-first search

1. Label all states that are reachable from S in 1 step but aren't reachable in less than 1 step.
2. Then label all states that are reachable from S in 2 steps but aren't reachable in less than 2 steps.
3. Then label all states that are reachable from S in 3 steps but aren't reachable in less than 3 steps.
4. Etc… until Goal state reached.
5. Try this out for our abstract search problem



Depth-first search

An alternative to BFS. Always expand from the most recently-expanded node, if it has any untried successors. Else backup to the previous node on the current path.

We use a data structure we'll call a Path to represent the path from the START to the current state:

- Path P = <START, d, e, r >
  - The path from START to r.

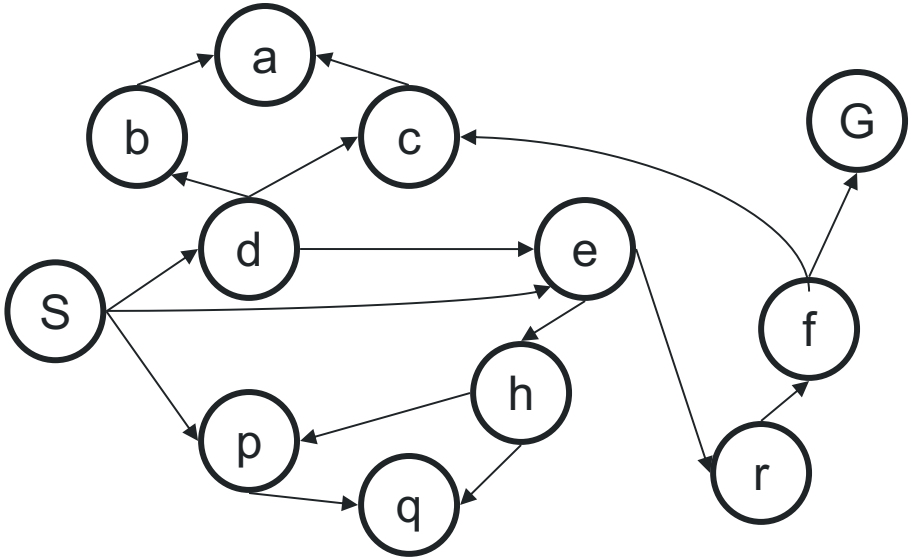
Along with each node on the path, we must remember which successors we still have available to expand:

P = <START (expand=e,p) ,
d (expand = b,c) ,
e (expand = h),
r (expand = f) >

DFS Algorithm

```
Let P = <START (expand = succs(START))>
While (P not empty and top(P) not a goal)
      if expand of top(P) is empty then
            remove top(P) ("pop the stack")
      else
            let s be a member of expand of top(P)
            remove s from expand of top(P)
            make a new item on the top of path P:
            s (expand = succs(s))
If P is empty then
      return FAILURE
Else
      return the path consisting of states in P
```

DIY: Search Tree (lexical)



Uniform Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Uniform cost (Dijkstra) search

A conceptually simple BFS approach when there are costs on transitions.

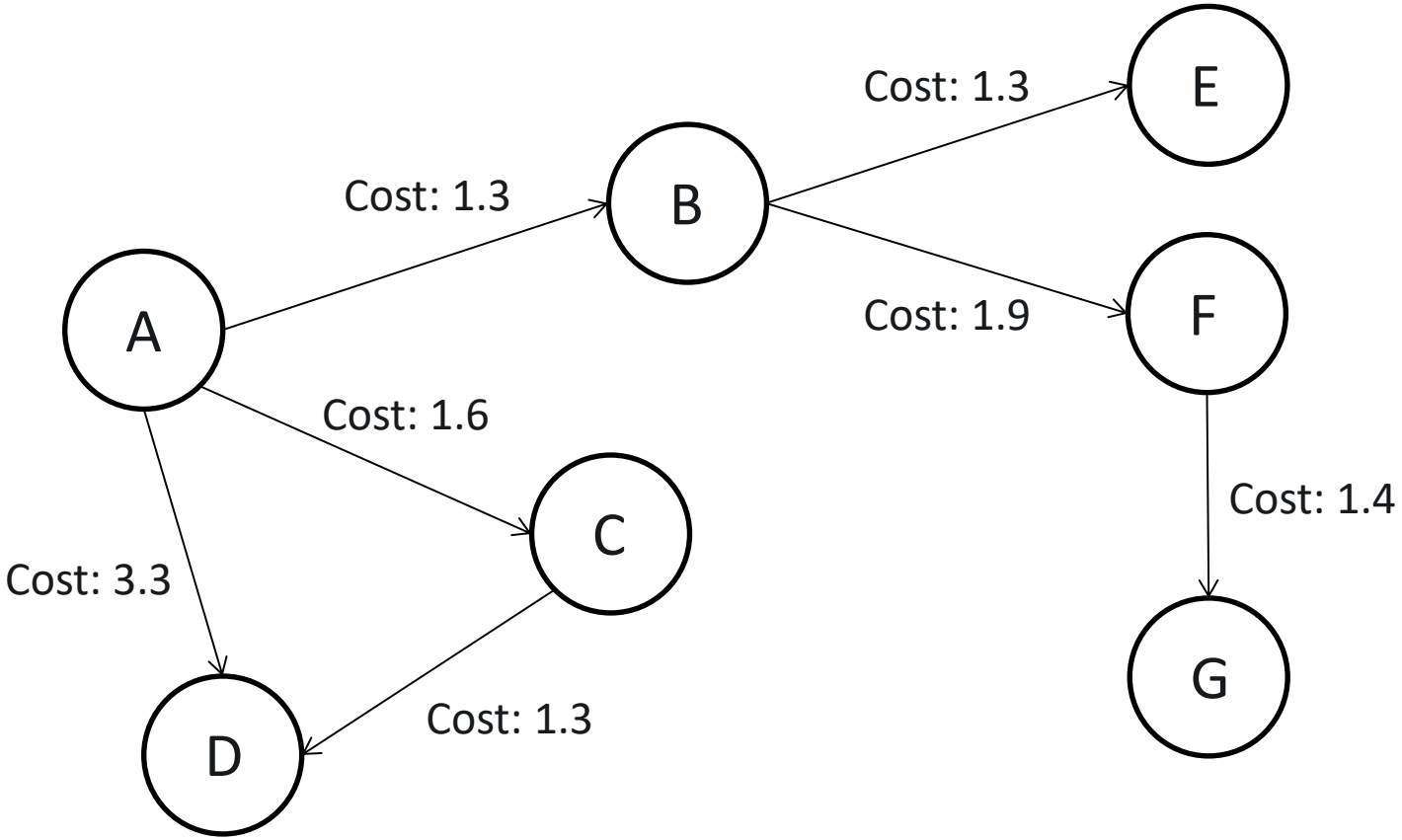
It uses a priority queue

- PQ = Set of states that have been expanded or are awaiting expansion
- Priority of state *s* = *g(s)* = cost of getting to *s* using path implied by backpointers.

Main loop

- Pop least cost state from PQ (the open list)
- Add the successors
- Stop when the goal is popped from the PQ (the open list)

DIY: Uniform Cost



Searching for a path from A to G

Data structure for the open list

- We need a data structure for the open list that makes the following operations inexpensive:
  - Adding an entry to the list
  - Removing an entry from the list
  - Finding the smallest element
  - Finding an entry corresponding to a particular node
- Let's analyze how well a linked list works

Data structure for the open list

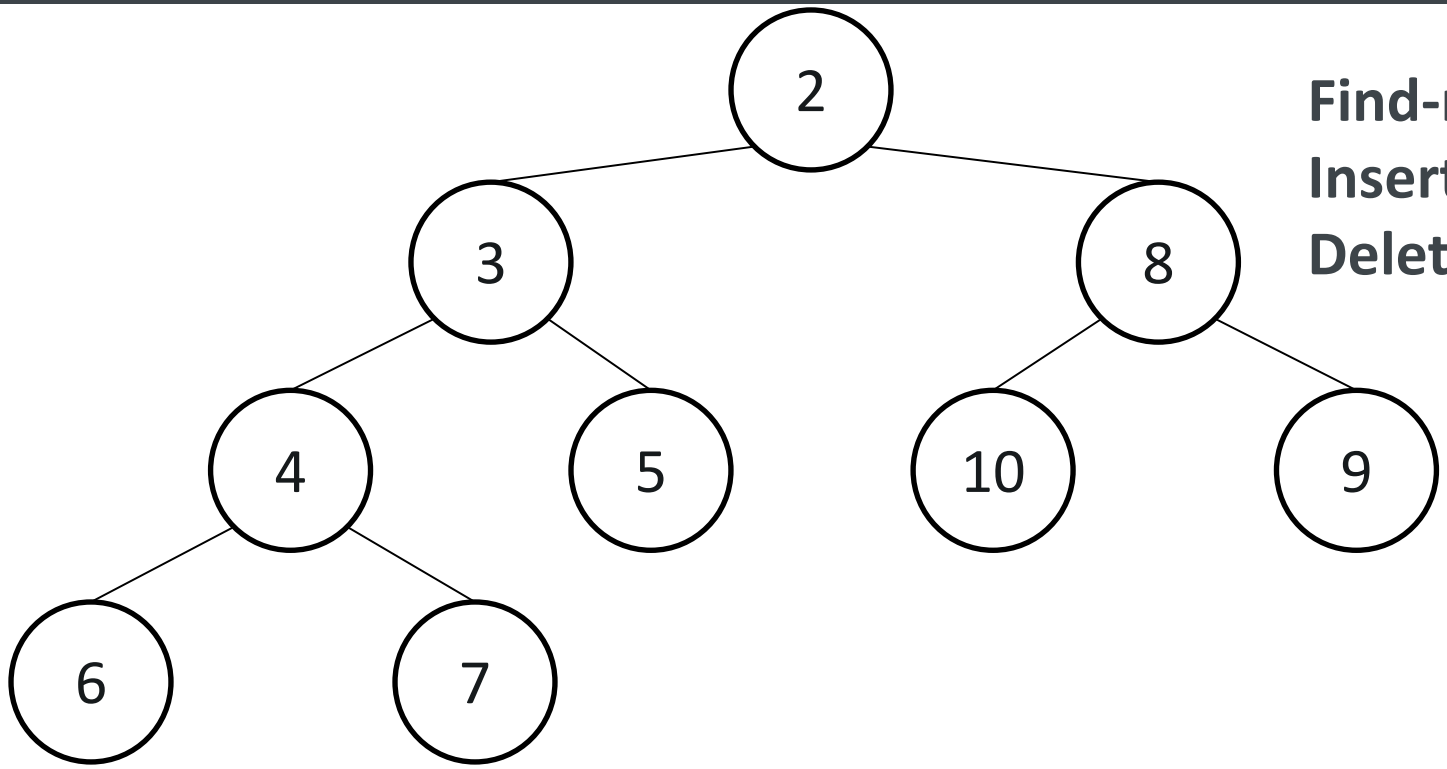
- We need a data structure for the open list that makes the following operations inexpensive:
  - Adding an entry to the list
  - Removing an entry from the list
  - Finding the smallest element
  - Finding an entry corresponding to a particular node

- Let's analyze how well a linked list work

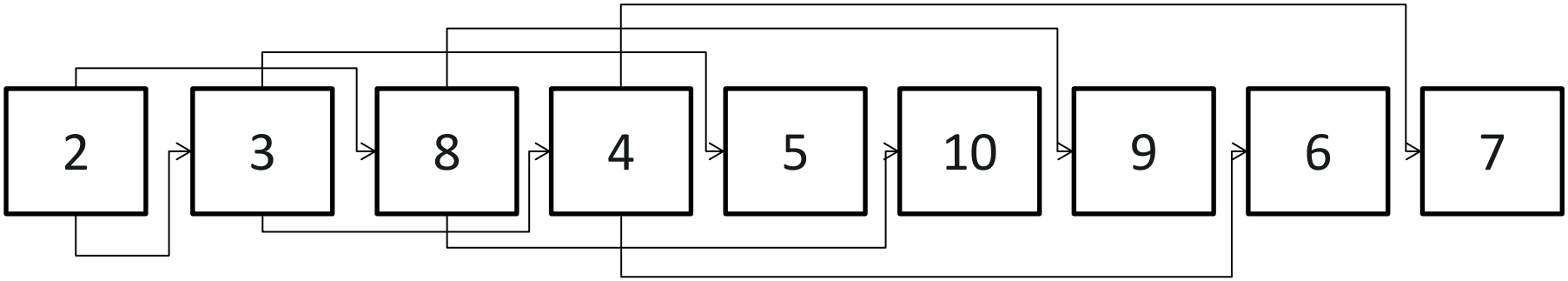
Access: O(n)
Insertion: O(1)
Search: O(n)

We can do better!

Priority heap



Find-min: O(1)
Insertion: O(log n)
Delete: O(log n)



Heuristic search

- Suppose in addition to the standard search specification we also have a *heuristic*.
- ***A heuristic function maps a state onto an estimate of the cost to the goal from that state.***
- What's an example heuristic for path planning?
- Denote the heuristic by a function $h(s)$ from states to a cost value.

Best first “greedy” search

```
insert-PriQueue(PQ,START,h(START))
while (PQ is not empty and PQ does not contain a goal state)
    (s, h) := pop-least(PQ)
    foreach s′ in succs(s)
        if s′ is not already in PQ and s′ never previously visited
            insert-PriQueue(PQ, s′, h(s′))
```

Greedy ignores the actual costs.

An improvement to this algorithm becomes A*!

A* search

Uniform-cost (Dijkstra): on expanding n, take each successor n' and place it on priority queue with priority g(n') (cost of getting to n')

Best-first greedy: on expanding n, take each successor n' and place it on priority queue with priority h(n')

A*: When you expand n, place each successor n' on priority queue with priority f(n') = g(n') + h(n')

A* Function

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

A* Function

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

Greedy

Heuristic

Let's Compare

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

A* algorithm

Priority queues **Open** and **Closed** begin empty. Both queues contain records of the form <**state**, **f**, **g**, **backpointer**>.

Put **S** into **Open** with priority f(s) = g(s) + h(s)

Is **Open** empty?

Yes? No solution.

No. Remove node with lowest f(n). Call it **n**.

If **n** is a **goal**, stop and report **success** (for pathfinding, return the path)

Otherwise, expand **n**. For each **n'** in **successors(n)**

Let **f'** = g(n') + h(n') = g(n) + cost (n, n') + h(n')

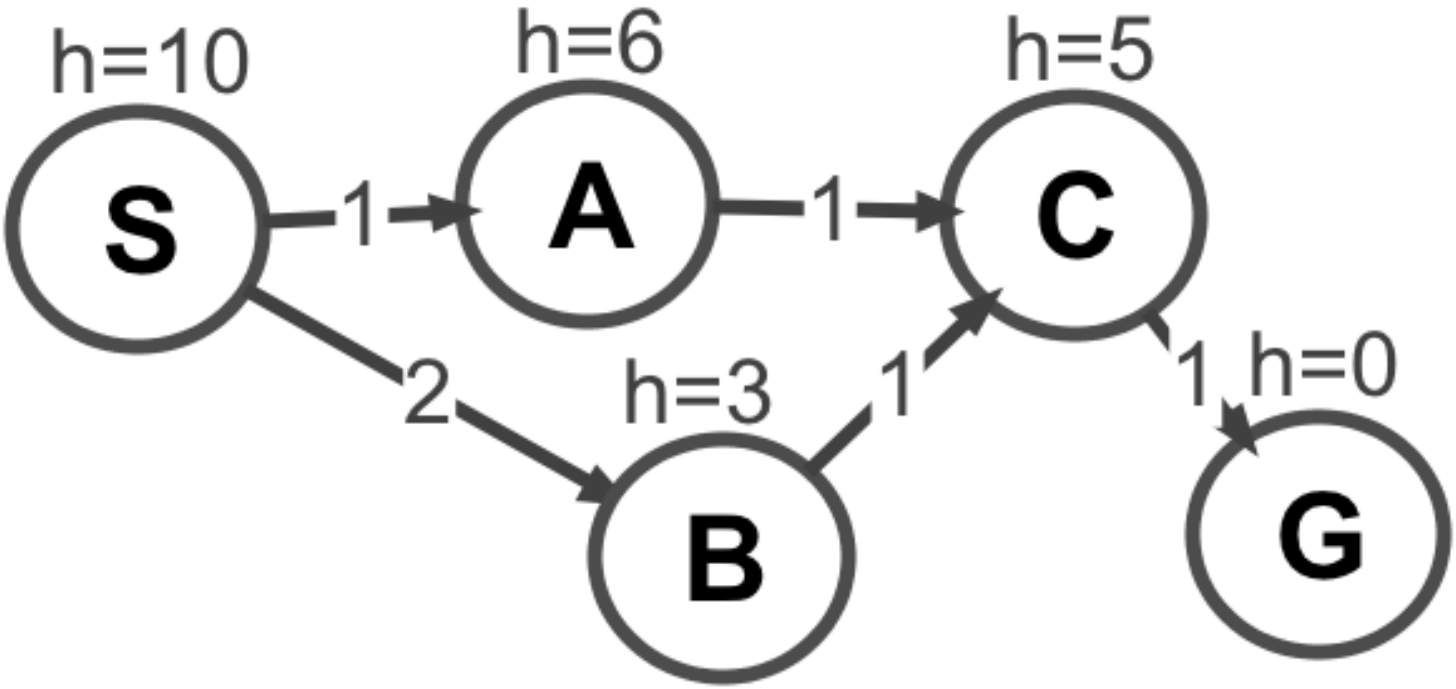
If **n'** not seen before or **n'** previously expanded with f(**n'**) > f' or **n'** currently in Open with f(**n'**) > f'

Then place or promote **n'** on **Open** with priority **f'**

Else ignore **n'**

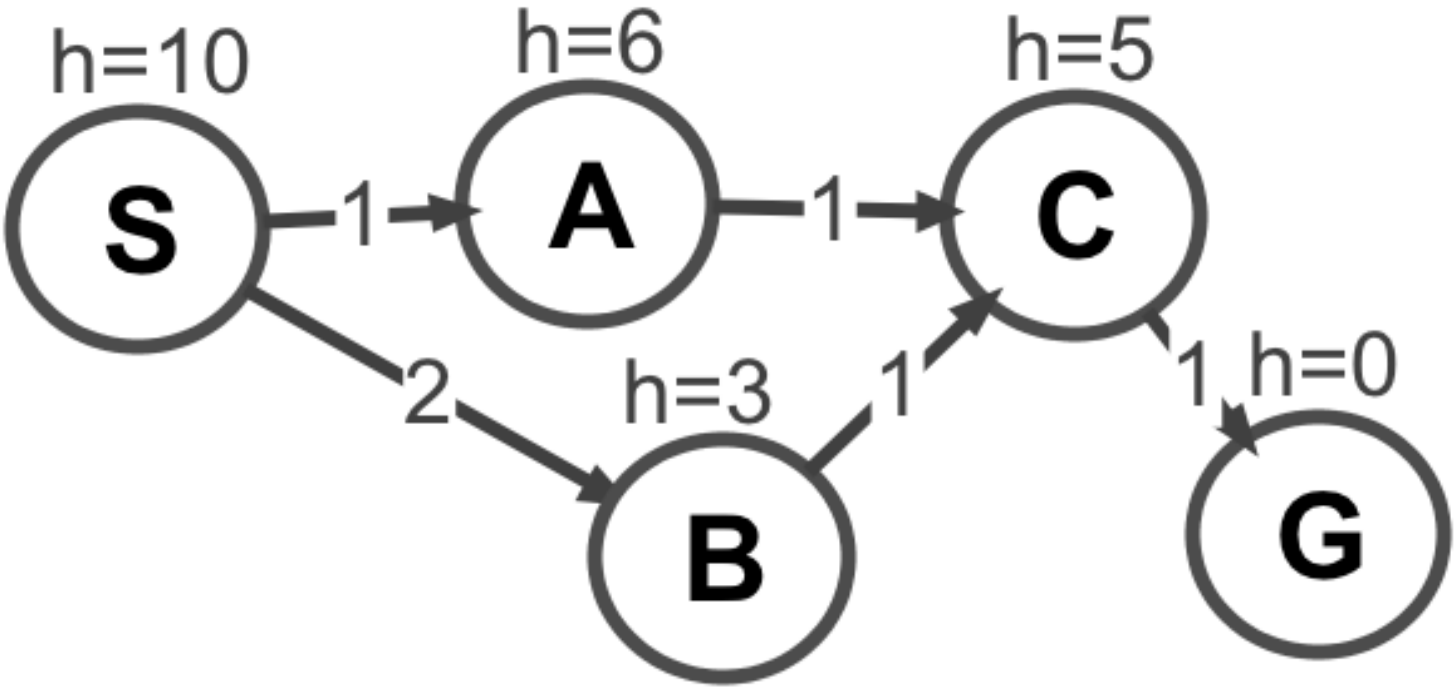
Place record for **n** in **Closed**

Let's try it out!



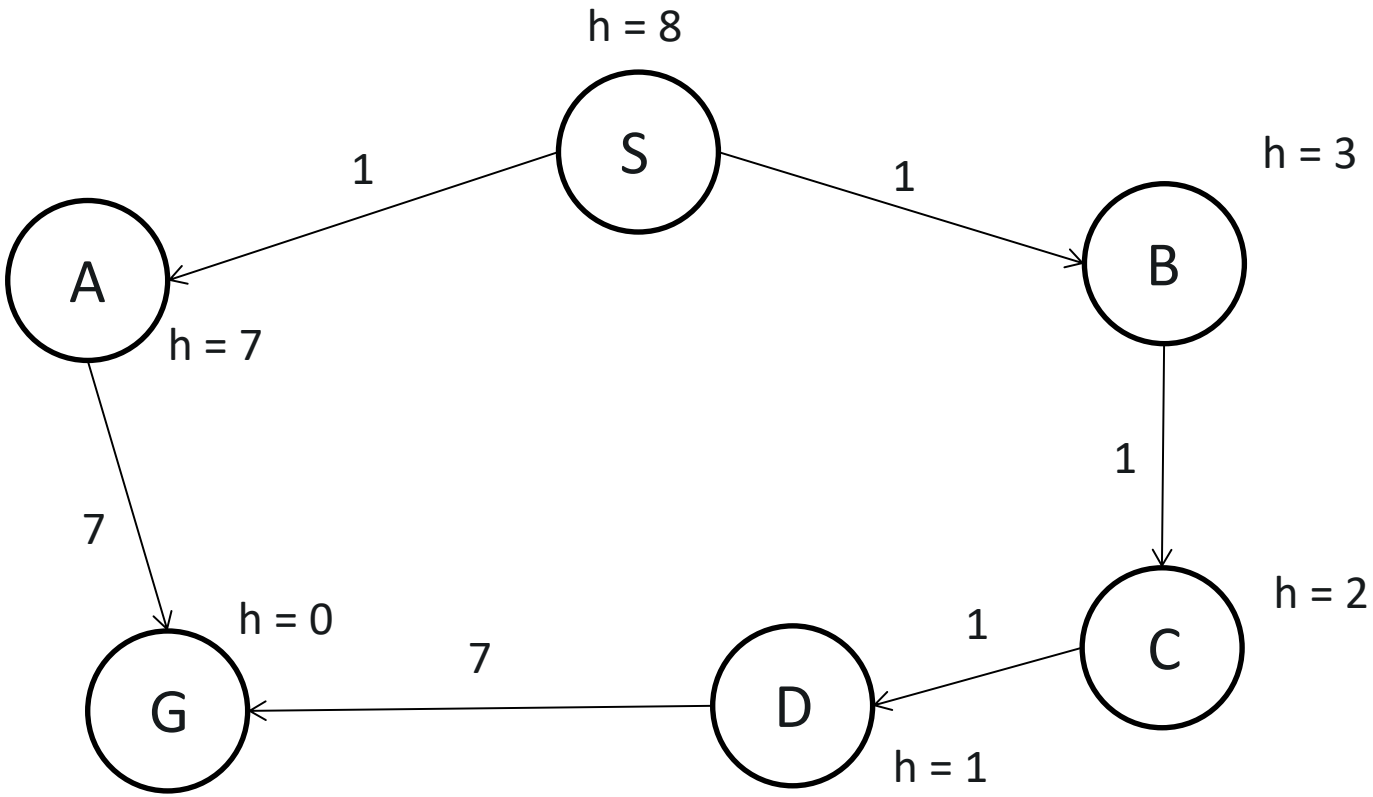
<state, f, g, backpointer>

Let's try it out!



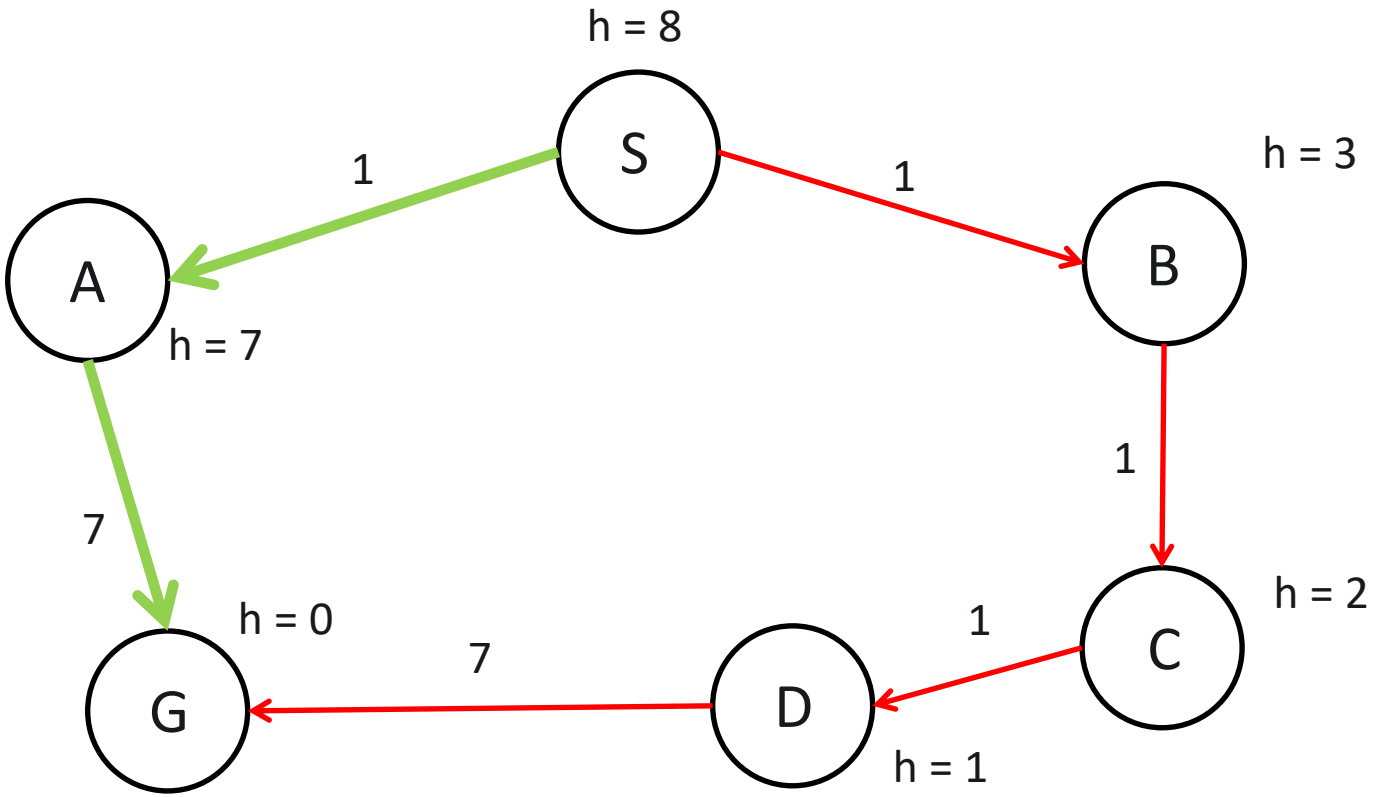
<state, f, g, backpointer>

When should A* terminate?



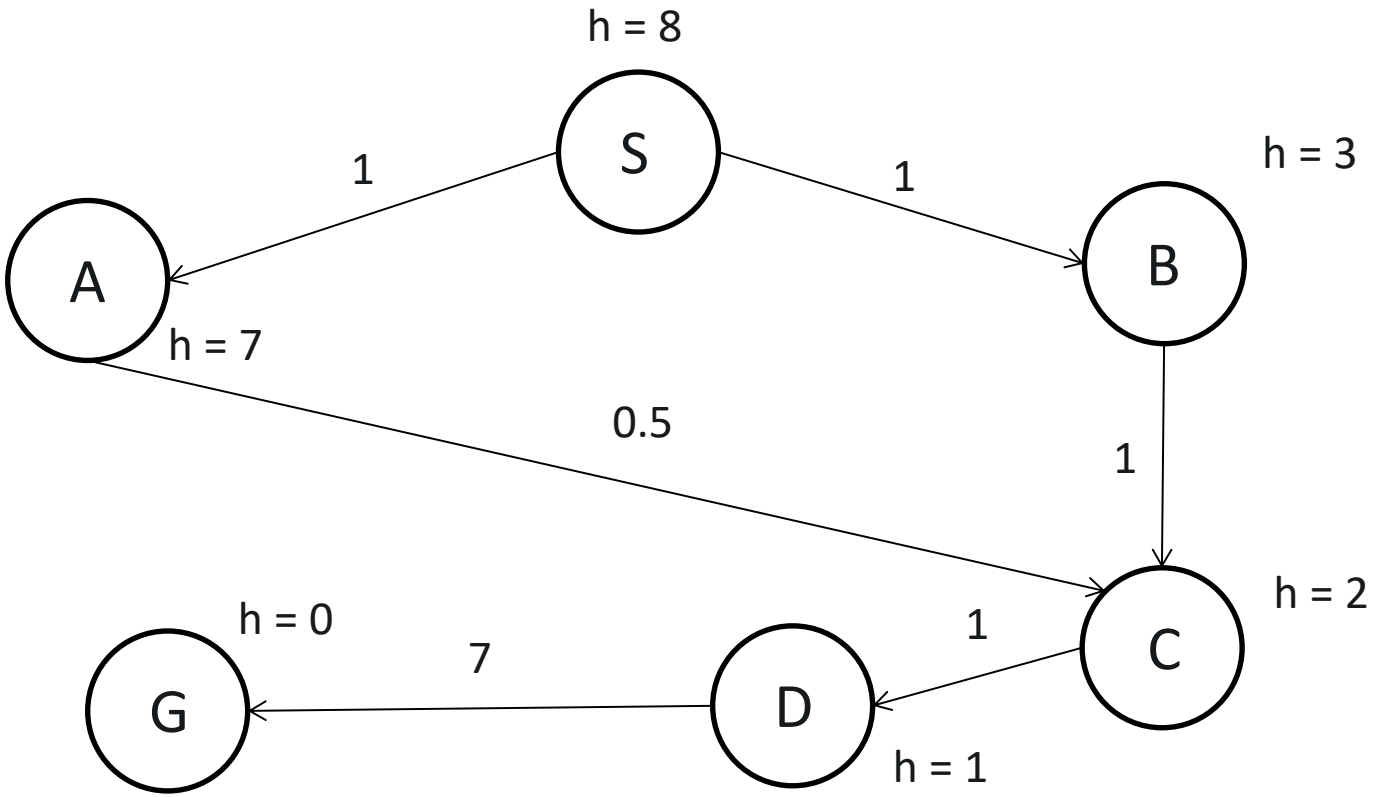
Should it terminate as soon as we generate the goal state?

Correct A* termination rule



A* terminates when a goal state is popped from the priority queue (open list)

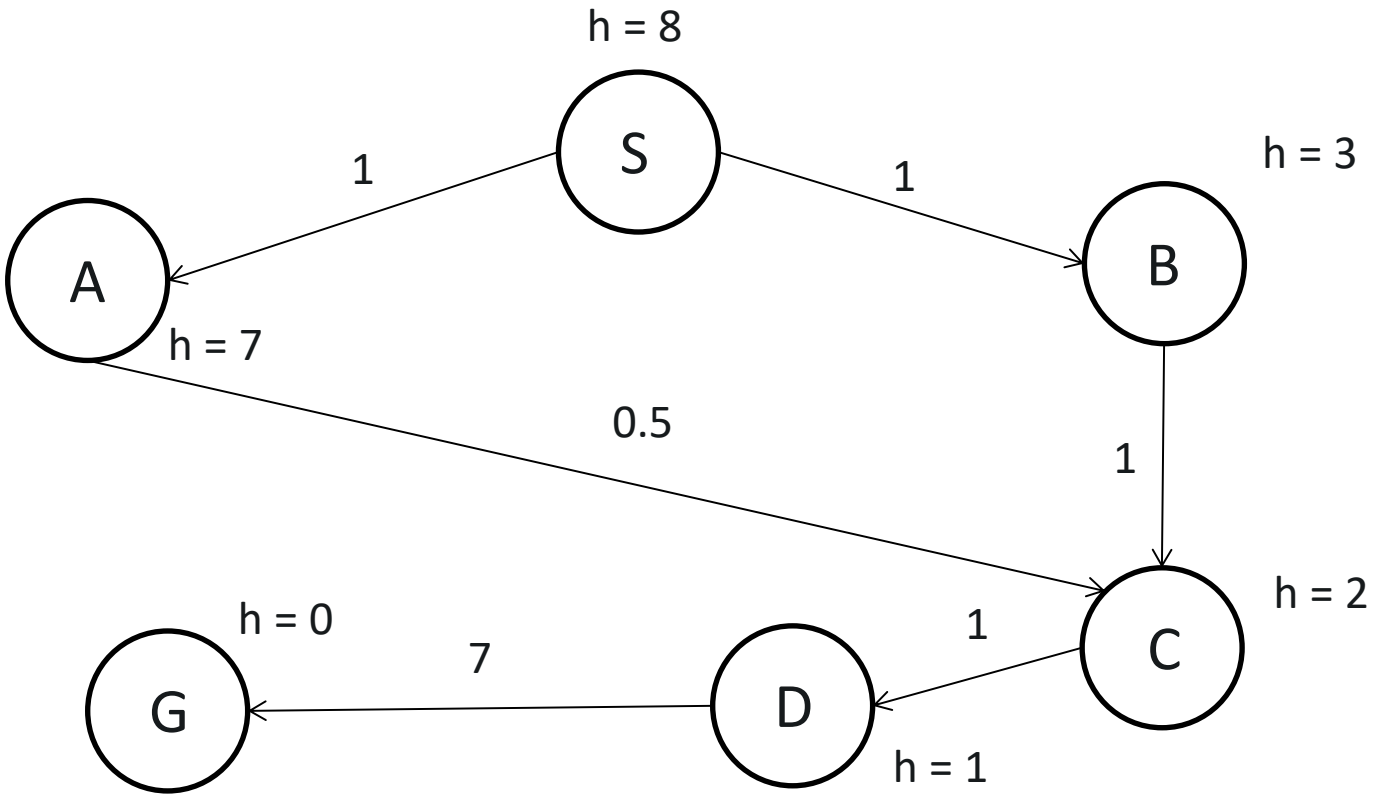
A* revisiting states



What if A* revisits a state that was already expanded and discovers a shorter path?

In this example, a state that was expanded gets re-expanded.

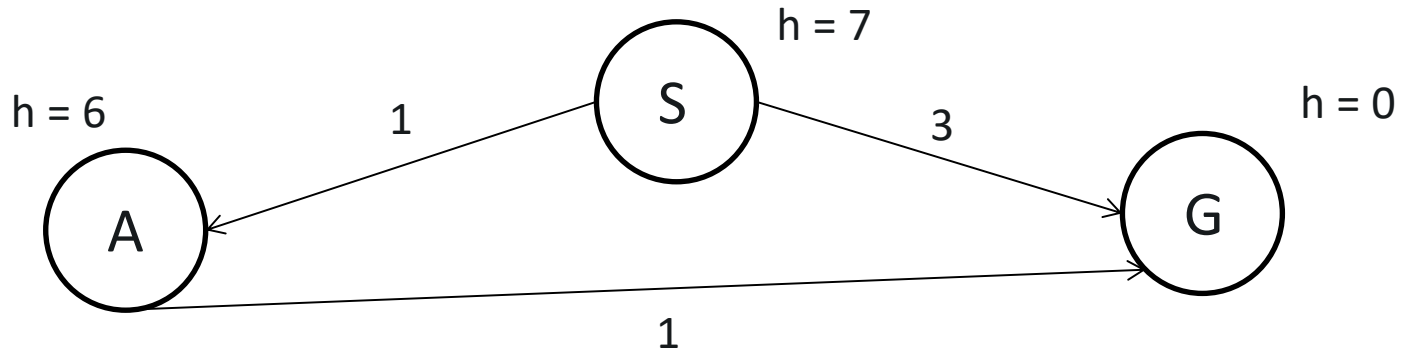
A* revisiting states



What if A* revisits a state that was already expanded and discovers a shorter path?

In this example, a state that was expanded gets re-expanded.

Is A* guaranteed to find the optimal path?



Nope. Not if the heuristic overestimates the cost.

Admissible heuristics

h*(n) = the true minimal cost to goal from n.

A heuristic is admissible if h(n) <= h*(n) for all states n.

An admissible heuristic is guaranteed to never overestimate the cost to the goal.

An admissible heuristic is optimistic.

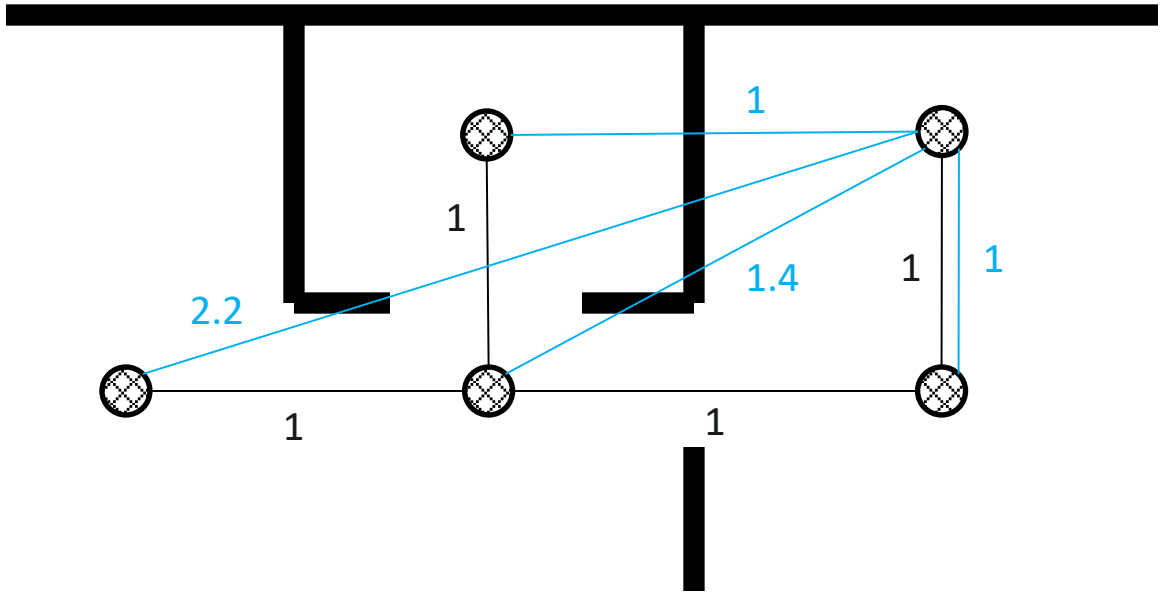
What's a common admissible heuristic for pathplanning?

Monotonicity or consistency for A* on graphs:

$$h(n) \leq c(n, a, n') + h(n')$$

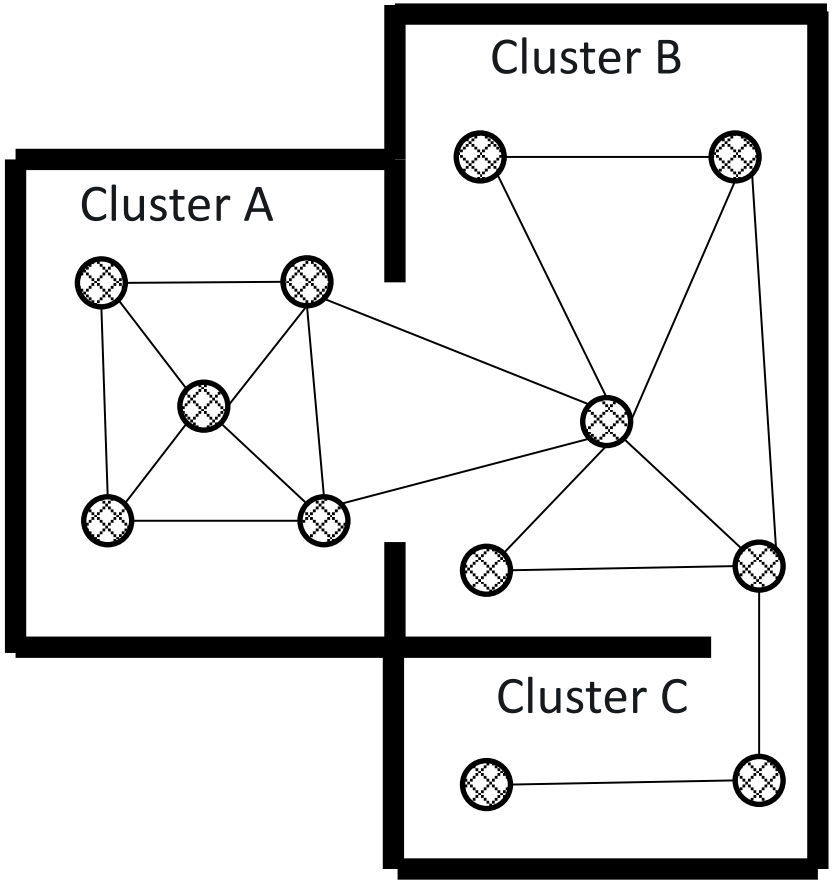
(The estimated cost for the successor must be less than or equal to the estimated cost of the parent plus the transition cost from the parent to the successor.)

Euclidian distance



- Guaranteed to be underestimating
- Can provide fast path planning on outdoor levels with few constraints
- May create too much fill on complex indoor environments

Cluster heuristic

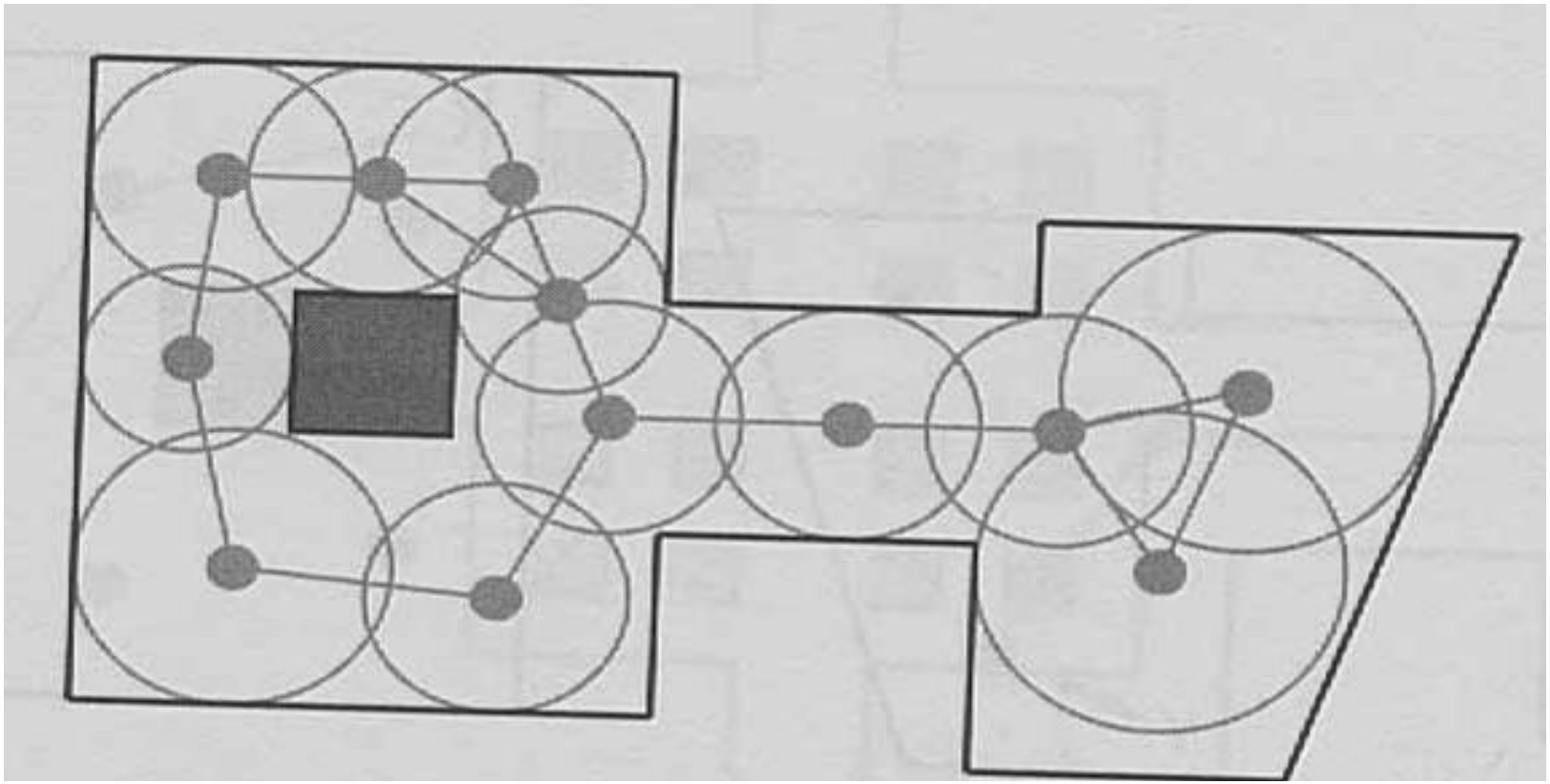


	A	B	C
A		13	29
B	13		7
C	29	7	

- Can be good for indoor environments
- Pre-compute accurate heuristic between clusters
- Use pre-computed values for points in two clusters, Euclidean (or something else simple) within clusters

Circular Waypoint Graphs

- Good in open terrain
- Not good in angular areas



A* pathfinding review

In A* pathfinding, nodes are locations, links are connections between locations

Actual costs are given by terrain (e.g. swamp harder to move through than plains)

Heuristic (projected) costs are typically manhattan distance (grid) or straight-line distance (non-grid movement)

Search space

For any search algorithm, the characteristics of the search space determine performance

Given a search problem (start and goal), the smallest search space that contains both the start and the goal gives the best performance

- Though solutions generated in different search spaces will have different properties
- For pathfinding, the aesthetics of the path will be greatly influence by search space

Let's look at different search spaces for pathfinding

Regular Grids

http://www.codeofhonor.com/blog/the-starcraft-path-finding-hack

Regular Grids

- Won't work for 3D game worlds without some modification
- Mostly used in strategy games (typically with a top-down perspective)
- Disadvantage: High resolution grids have large memory footprint
- Advantage: Provide random access look-up

Regular Grids

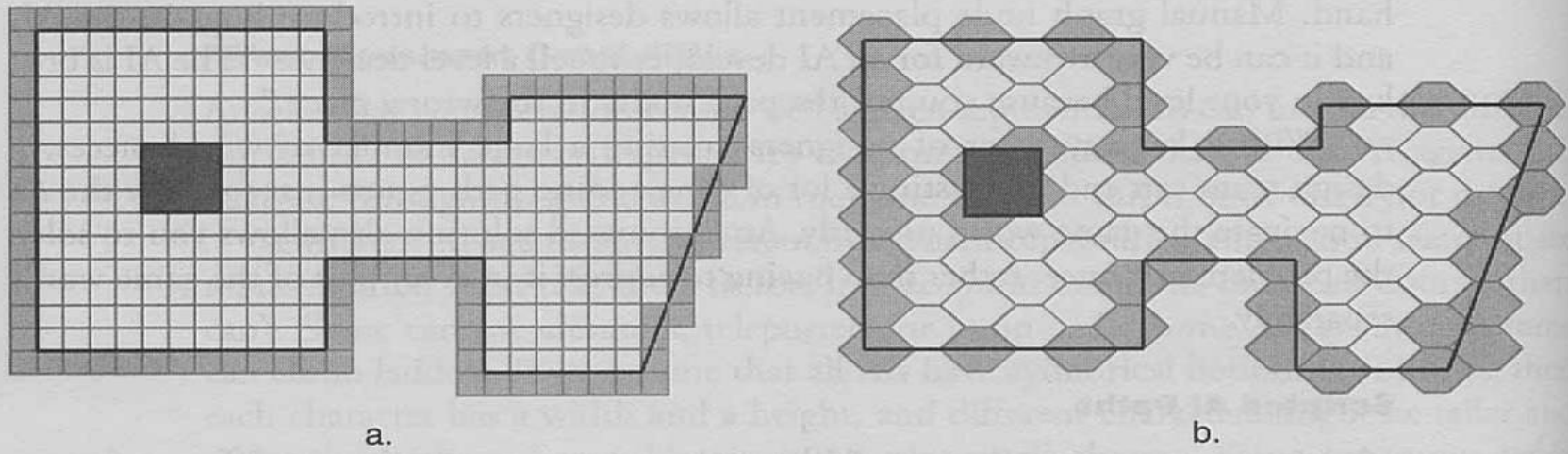
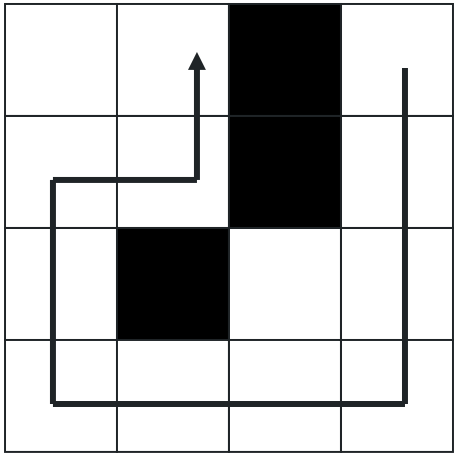


FIGURE 2.1.3 *Grid representations based on square and hexagonal cells.*

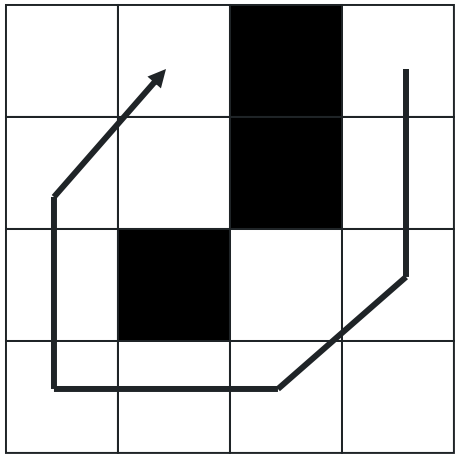
Grids and Movement

- Moving in only 4 cardinal directions gives you unattractive angular paths



Grids and Movement

Add in the diagonals and you improve the movement somewhat

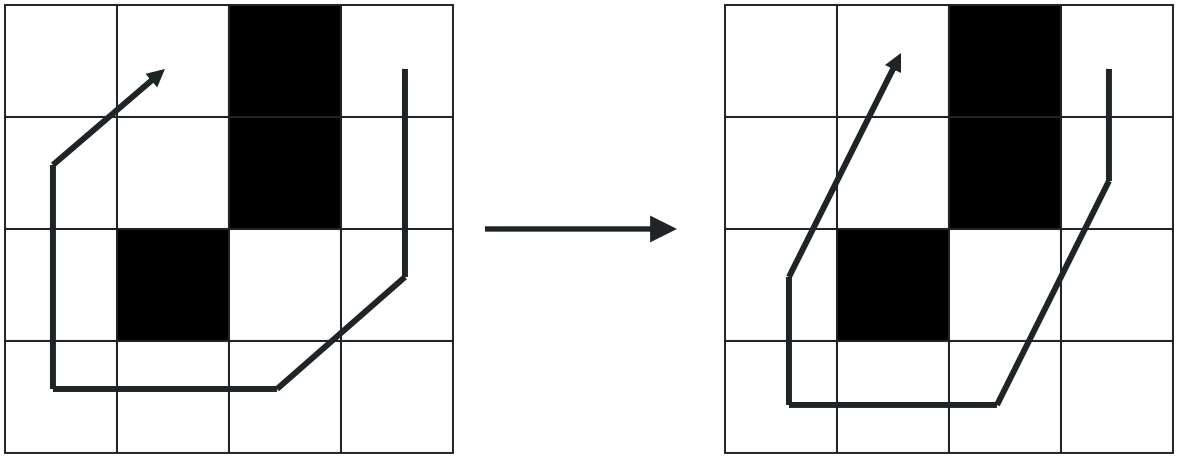


An Optimization

String-pulling or Line-of-sight testing can be used to improve this further.

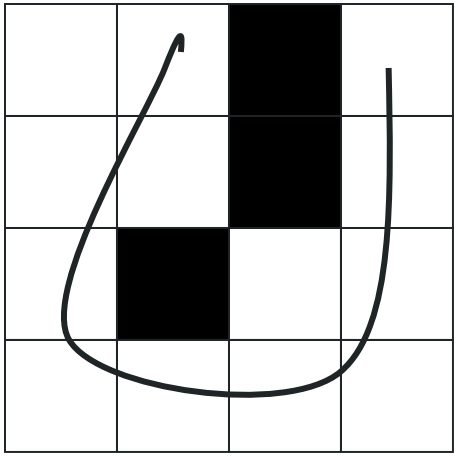
Delete any point Pn from path when it is possible to get from Pn-1 to Pn+1 directly.

Don’t need to travel through node centers.



Another Optimization

Use Catmull-Rom splines to create a smooth curved path.

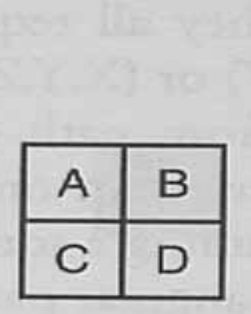


Graphs

The rest of the search space representations are graphs

You can think of grids as graphs

- Could be useful to have directed graphs (cliffs)



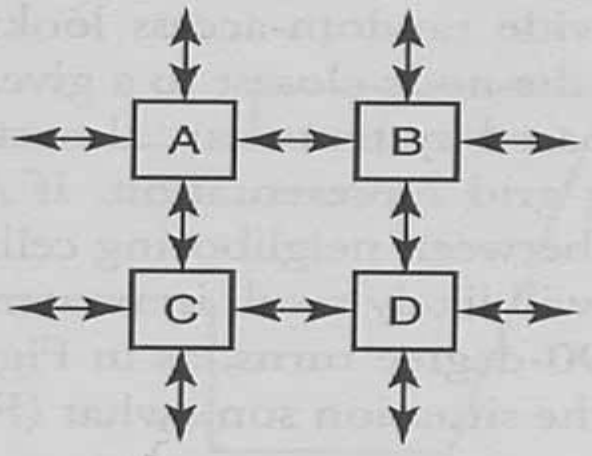


FIGURE 2.1.5 *The implicit "edges" between adjacent grid tiles.*

Slides after this are for those interested in delving further into path planning. You will not be assessed on this material.

END OF CONTENT FOR ECS170

Corner graphs

- Place waypoints on the corners of obstacles
- Place edges between nodes where a character could walk in a straight line between them

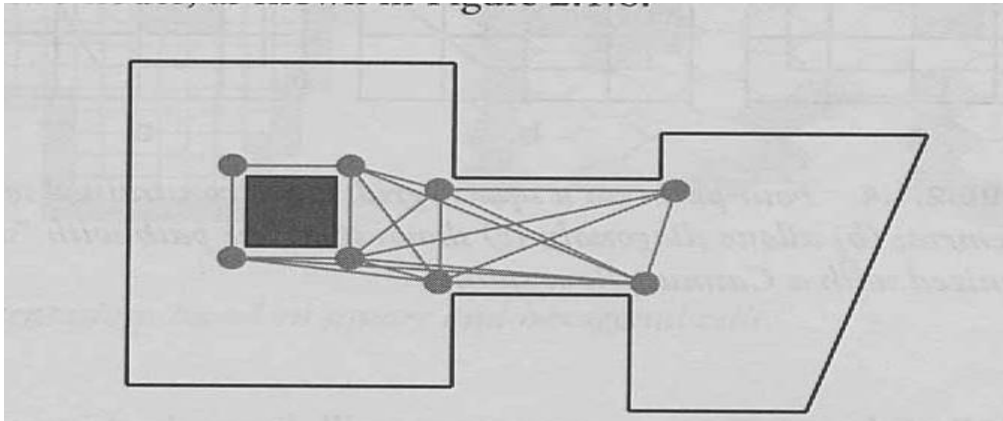
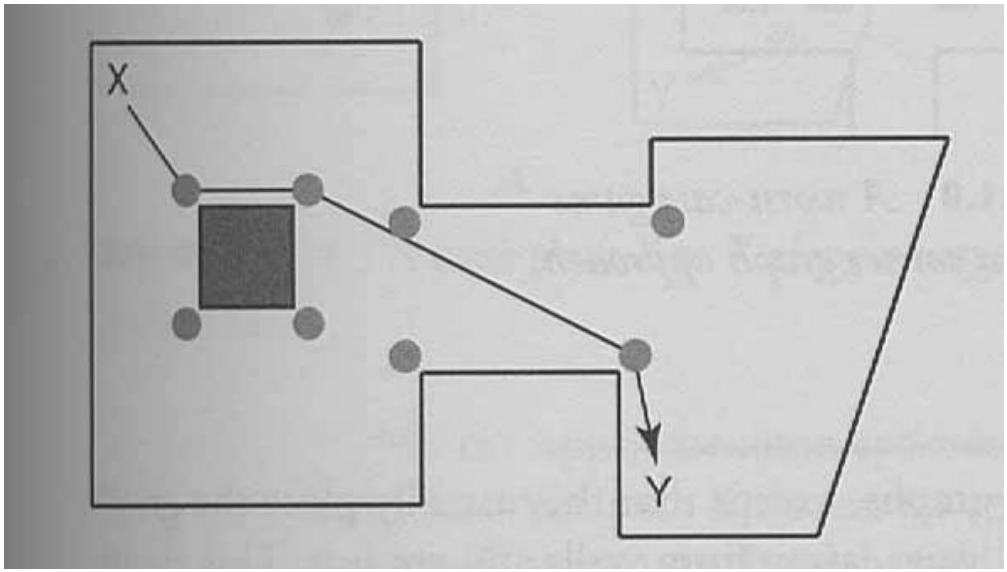


FIGURE 2.1.6 *A corner graph.*

Corner Graphs

- Sub-optimal paths
- AI agents appear to be “on rails”



Corner Graphs and String-pulling

- Can get close to the optimal path in some cases with String-pulling
- Requires expensive line-of-sight-testing

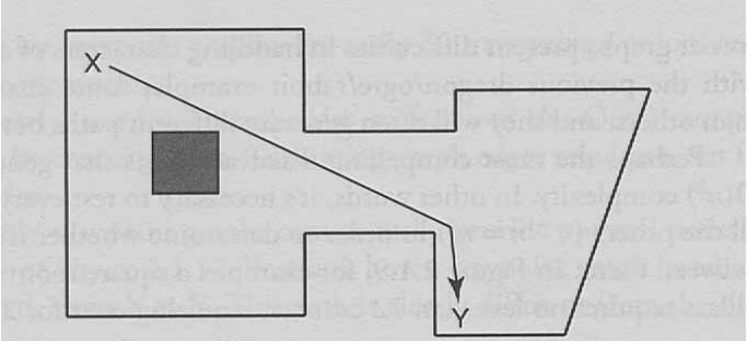
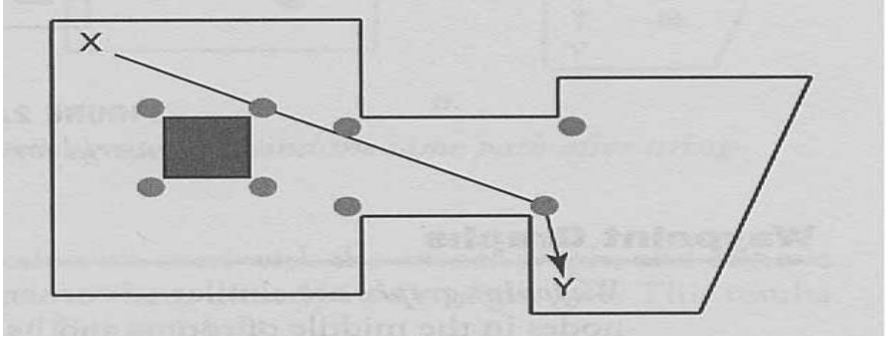


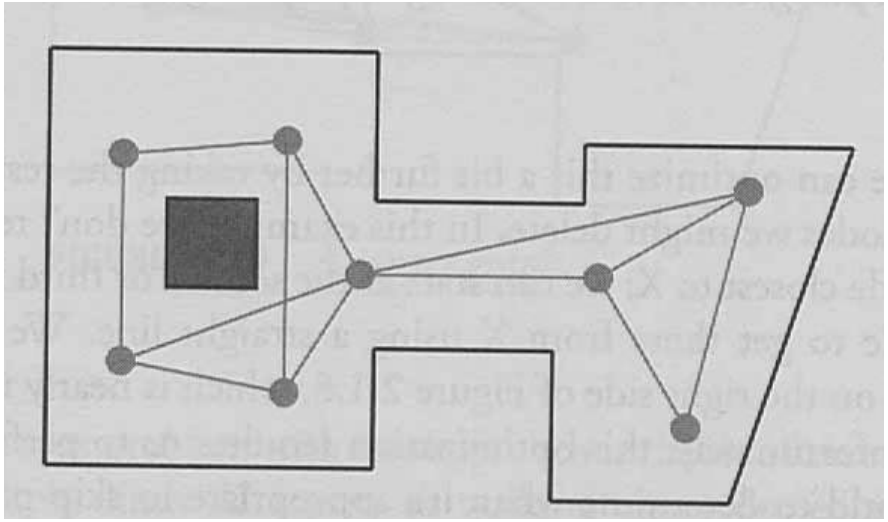
FIGURE 2.1.7 *The optimal path from X to Y.*



String-pulling

Waypoint Graphs

- Work well with 3D games and tight spaces
- Similar to Corner Graphs
  - Except nodes are further from walls and obstacles
  - Avoids wall-hugging issues



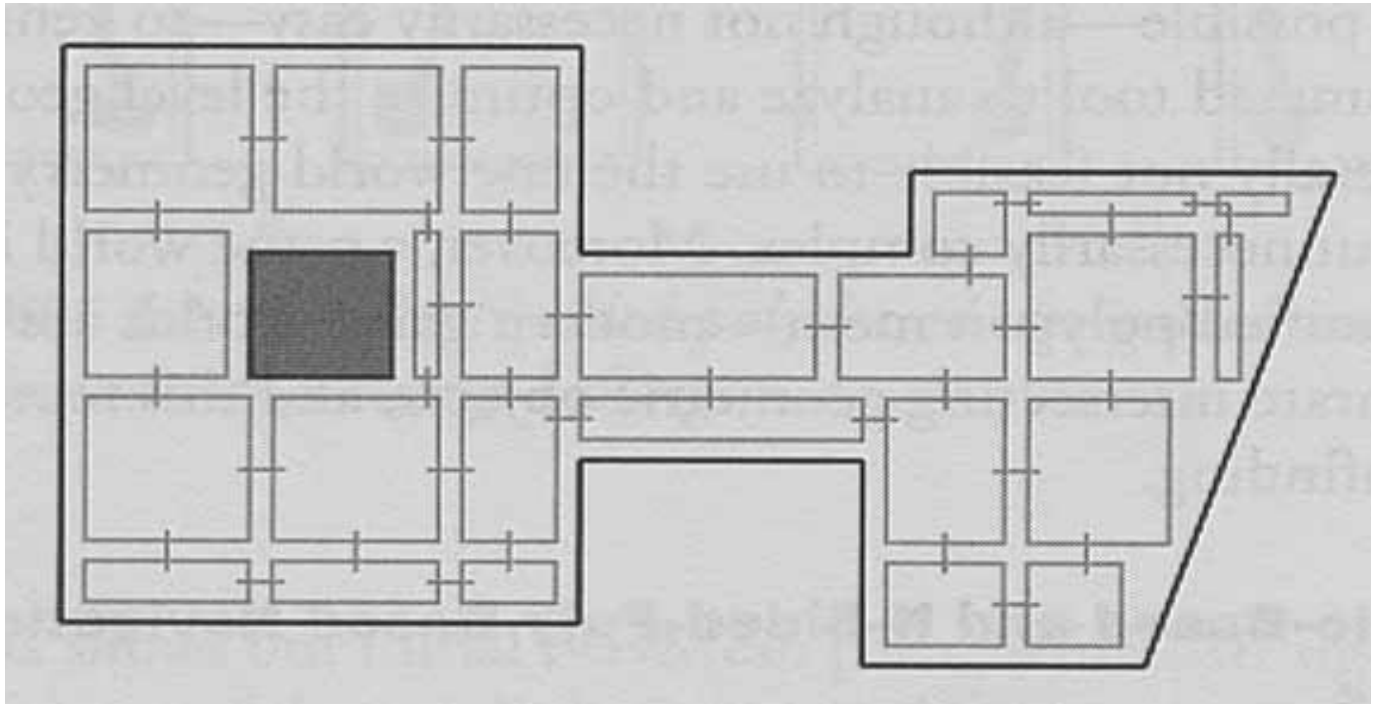
Circle-based Waypoint Graphs

- Same as waypoint graphs
  - Except add a radius around each node indicating open space
- Adds a little more information to each node
- Edges exist only between nodes whose circles overlap
- Some games use a hybrid of Circle-based waypoint graphs and regular waypoint graphs, using Circle-based for outdoor open terrain and regular for indoor environments

Space-Filling Volumes

- Similar to Circle based approach, but use rectangles instead of circles
- Work better than circle based in angular environments, but might not be able to completely fill all game worlds

Space-filled Volumes



Navigation Meshes

- Handles indoor and outdoor environments equally well
- Cover walkable surfaces with convex polygons
- Requires storage of large number of polygons, especially in large worlds or geometrically complex areas
- Example games: used in Thief 3 and Deus Ex 2

NavMesh

- Polygons must be convex to guarantee that an agent can walk from any point within a polygon to any other point within that same polygon
- Is possible to generate NavMesh with automated tool

2 types of NavMeshs

- Triangle based
  - All polygons must be triangles
  - When done correctly, will not hug walls too tightly
- N-Sided-Poly-based
  - Can have any number of sides, but must remain convex
  - Can usually represent a search space more simply than triangle based (smaller memory footprint)
  - Can lead to paths that hug walls too tightly

2 types of NavMeshs

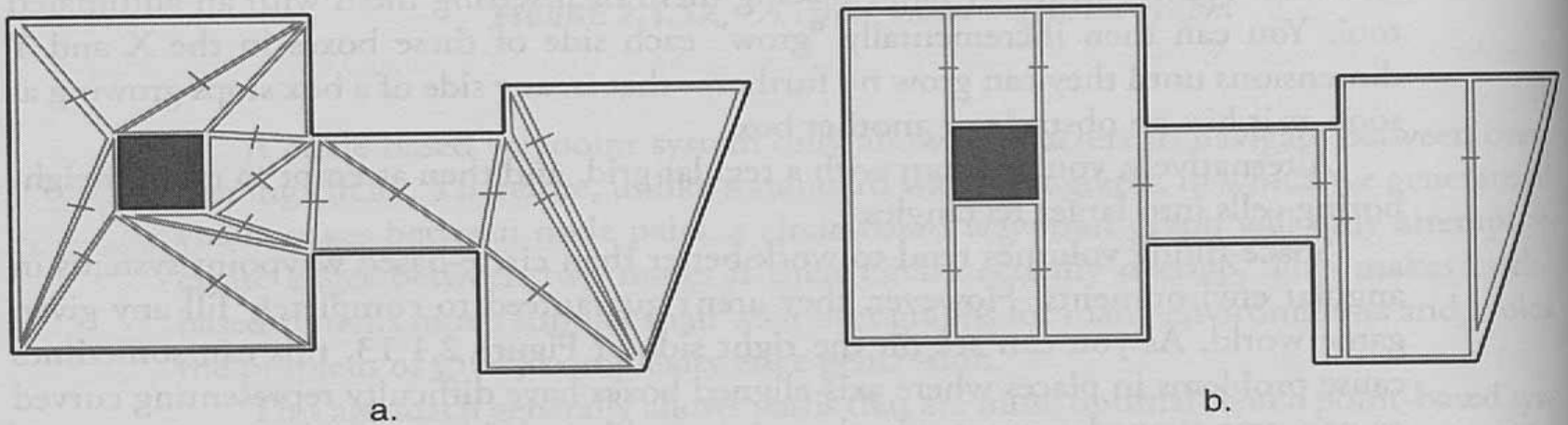


FIGURE 2.1.14 *Triangle-based (a) and N-sided-poly-based (b) navigation meshes.*

N-Sided-Poly-Based

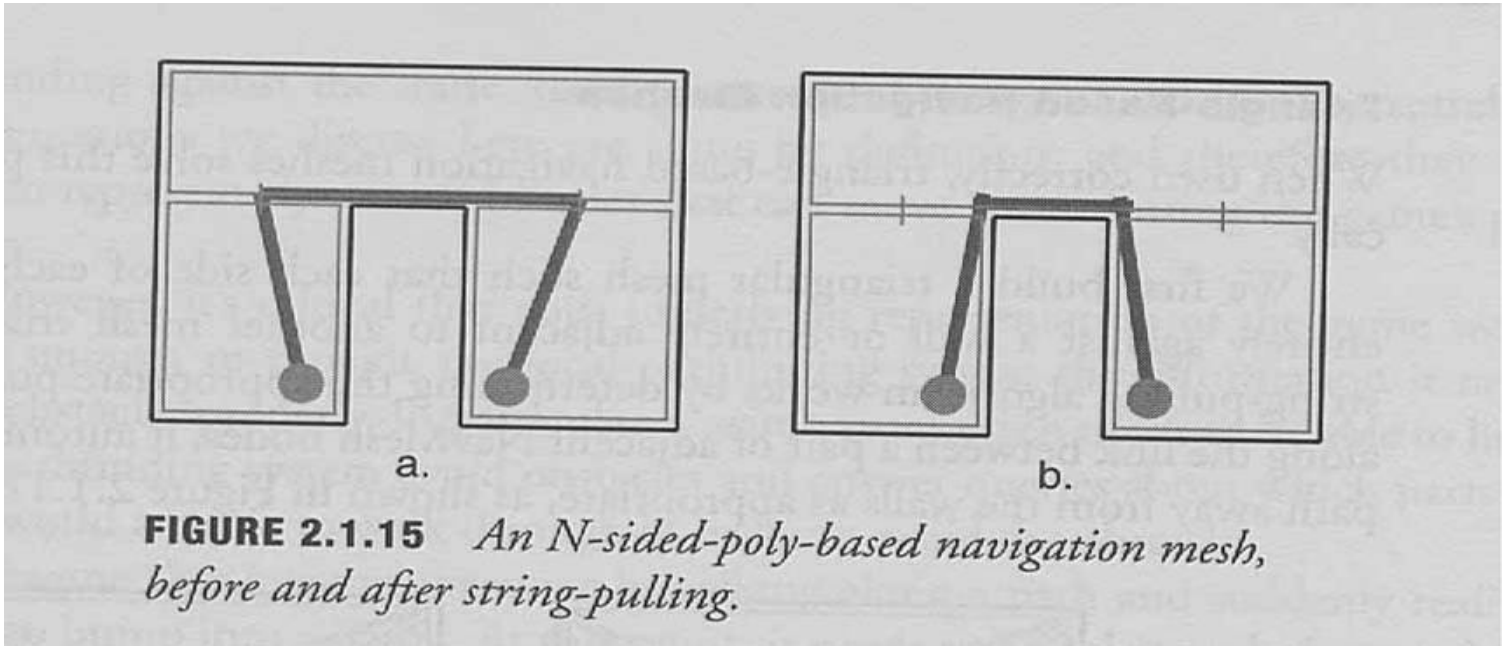


FIGURE 2.1.15 *An N-sided-poly-based navigation mesh, before and after string-pulling.*

Can address this problem with post-processing

Interacting with local path following

- Search Space is static, so it can't really deal with dynamic objects
- Should design it to give some information to pathfinding algorithm that will help
- "Can I go this way instead?" – search space should be able to answer this

Interacting with local path following

- Search Space is static, so it can't really deal with dynamic objects
- Should design it to give some information to pathfinding algorithm that will help
- "Can I go this way instead?" – search space should be able to answer this

Interacting with local path following

- Pathfinding algorithm must be able to deal with dynamic objects (things player can move)
- Can use simple object avoidance systems, but can break down in worlds with lots of dynamic objects

Interacting with local path following

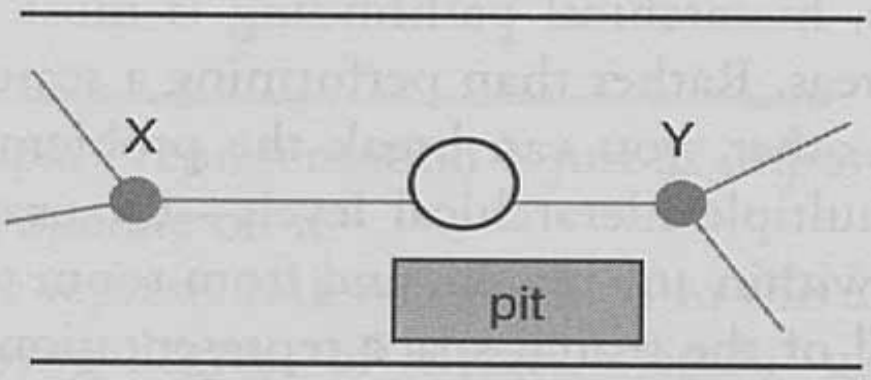


FIGURE 2.1.18 *A dynamic obstacle (the outlined circle) along an edge in a waypoint graph.*

Don’t want to do this

Interacting with local path following

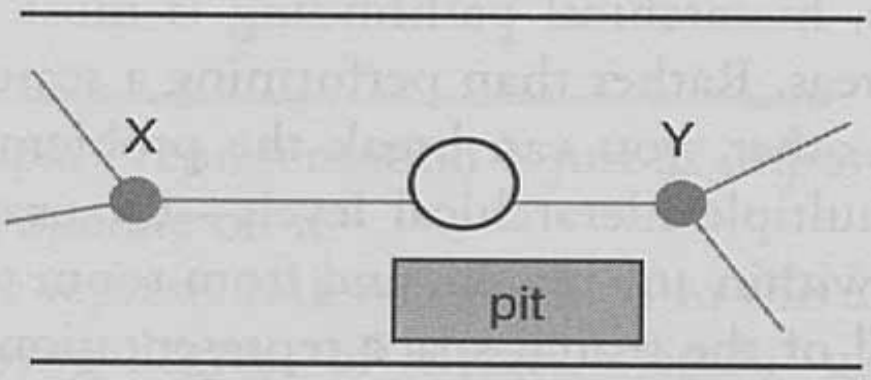


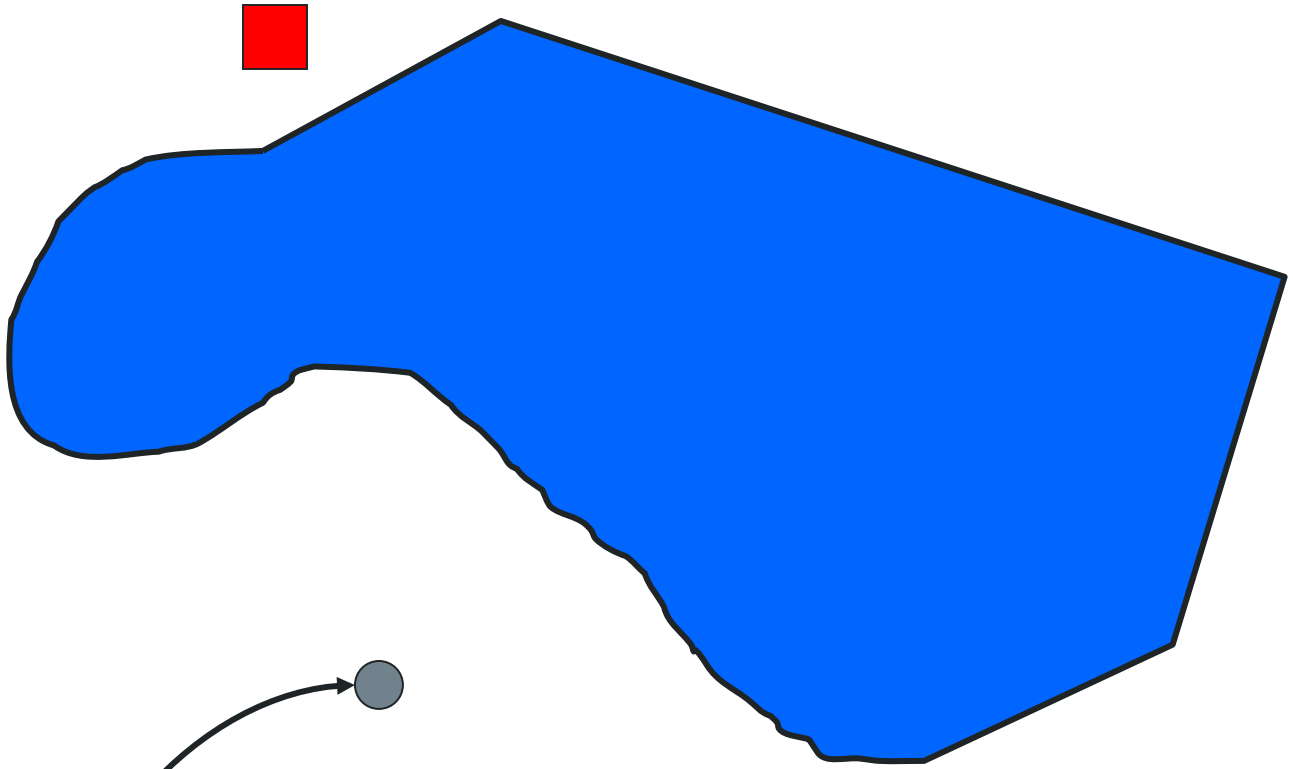
FIGURE 2.1.18 *A dynamic obstacle (the outlined circle) along an edge in a waypoint graph.*

Should do this

Generating timely paths

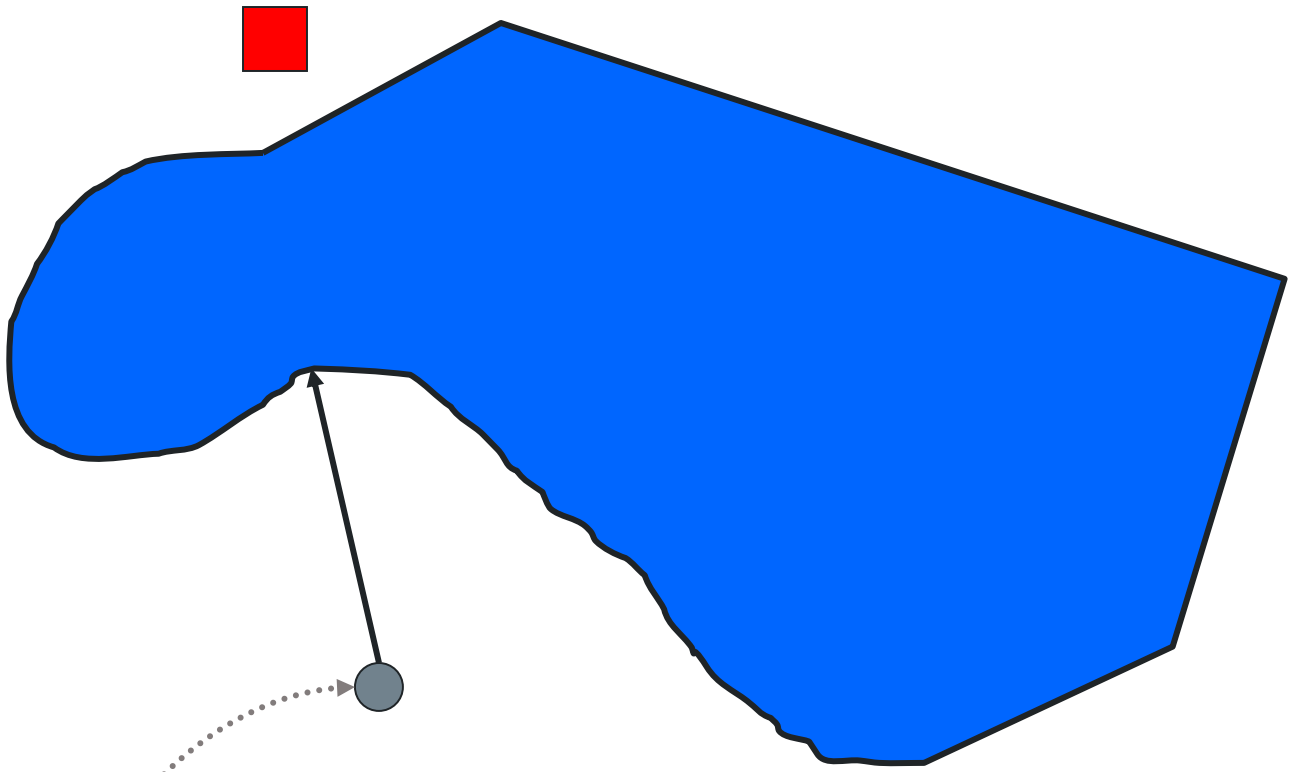
- Even with a good search space, when many AI agents are pathfinding, pathfinding all at once won't work
  - Pathfinding will chew up all the CPU resulting in game freezes
  - Even if the whole game doesn't freeze, agents will just sit there waiting for other agents to finish pathfinding
- The solution: timeslicing

The ideal path



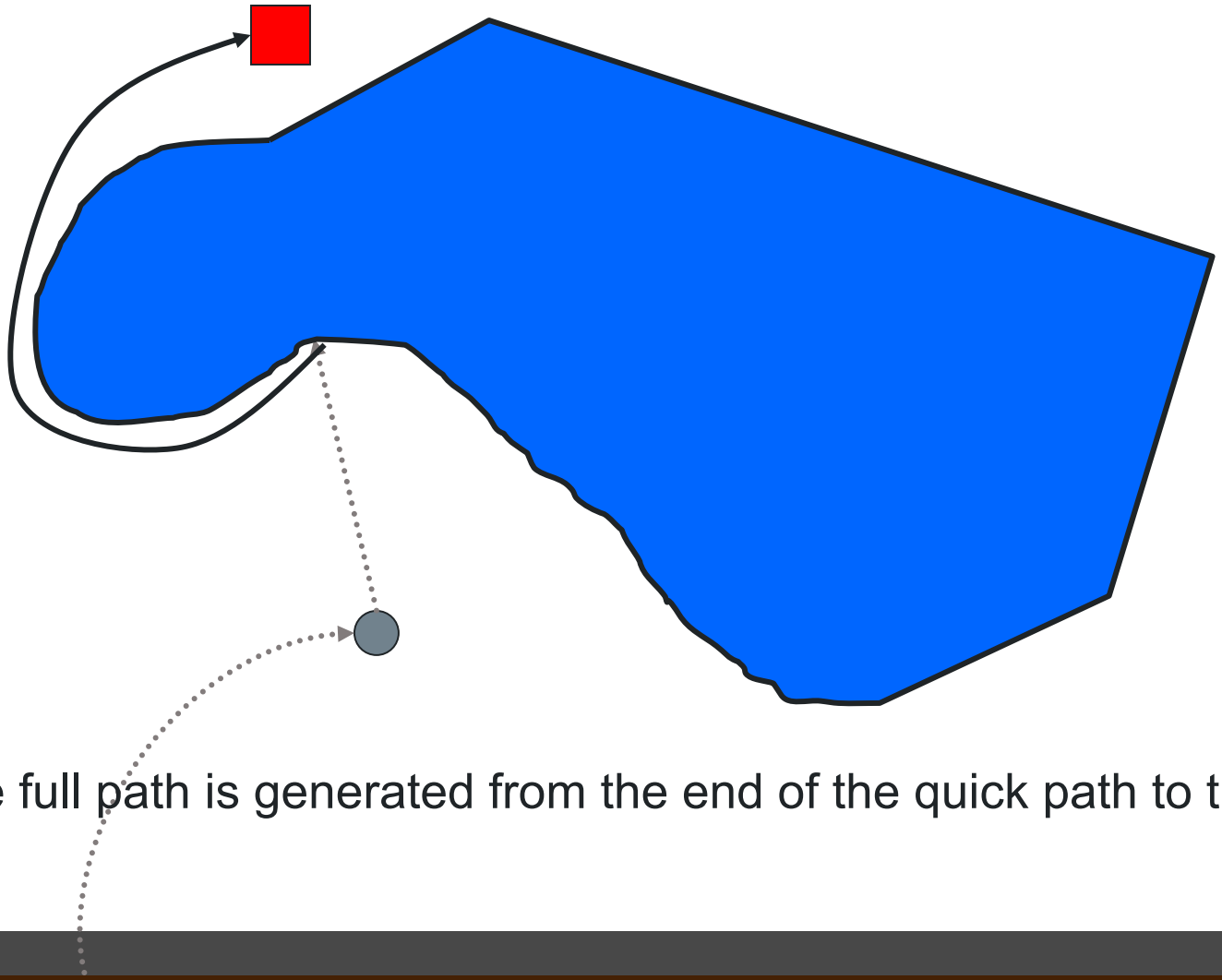
The idea path that would be generated by all-at-once pathfinding

The quick path



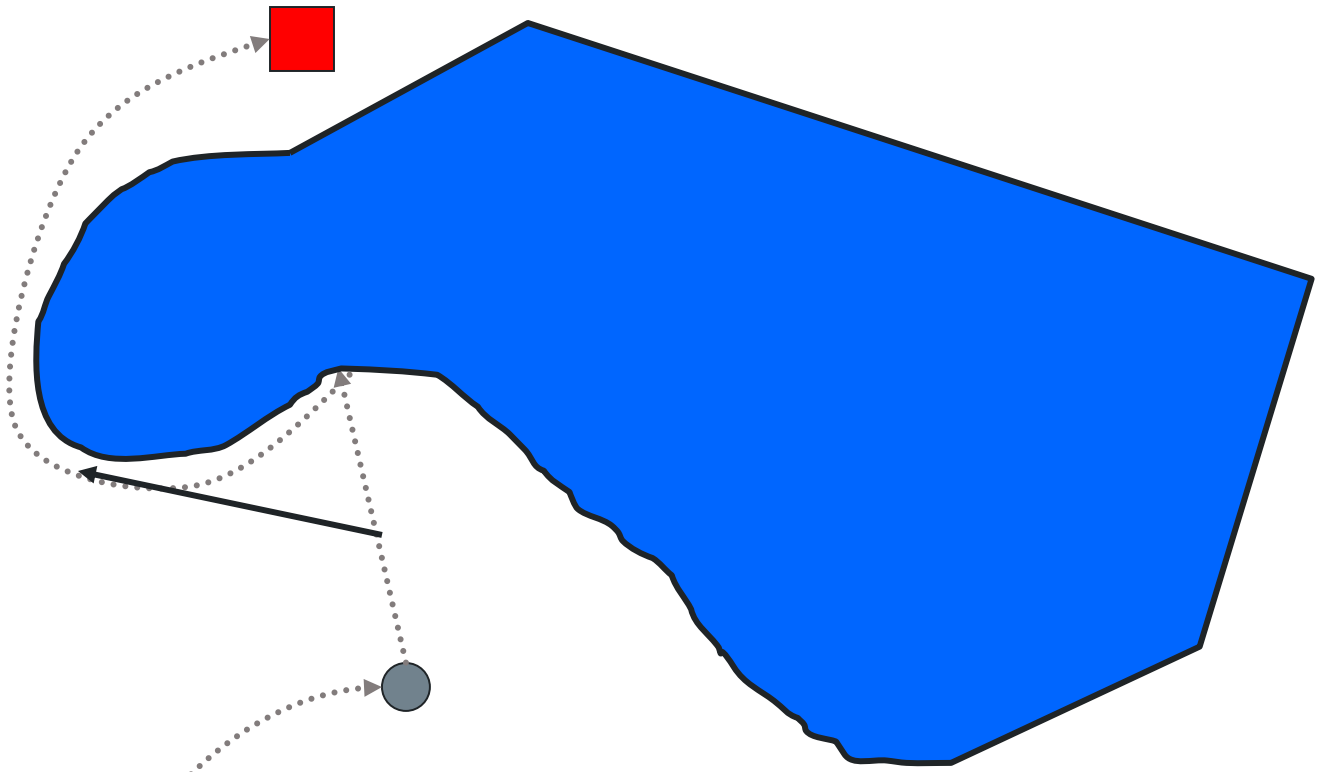
The quick path is generated by limited depth A*

The full path



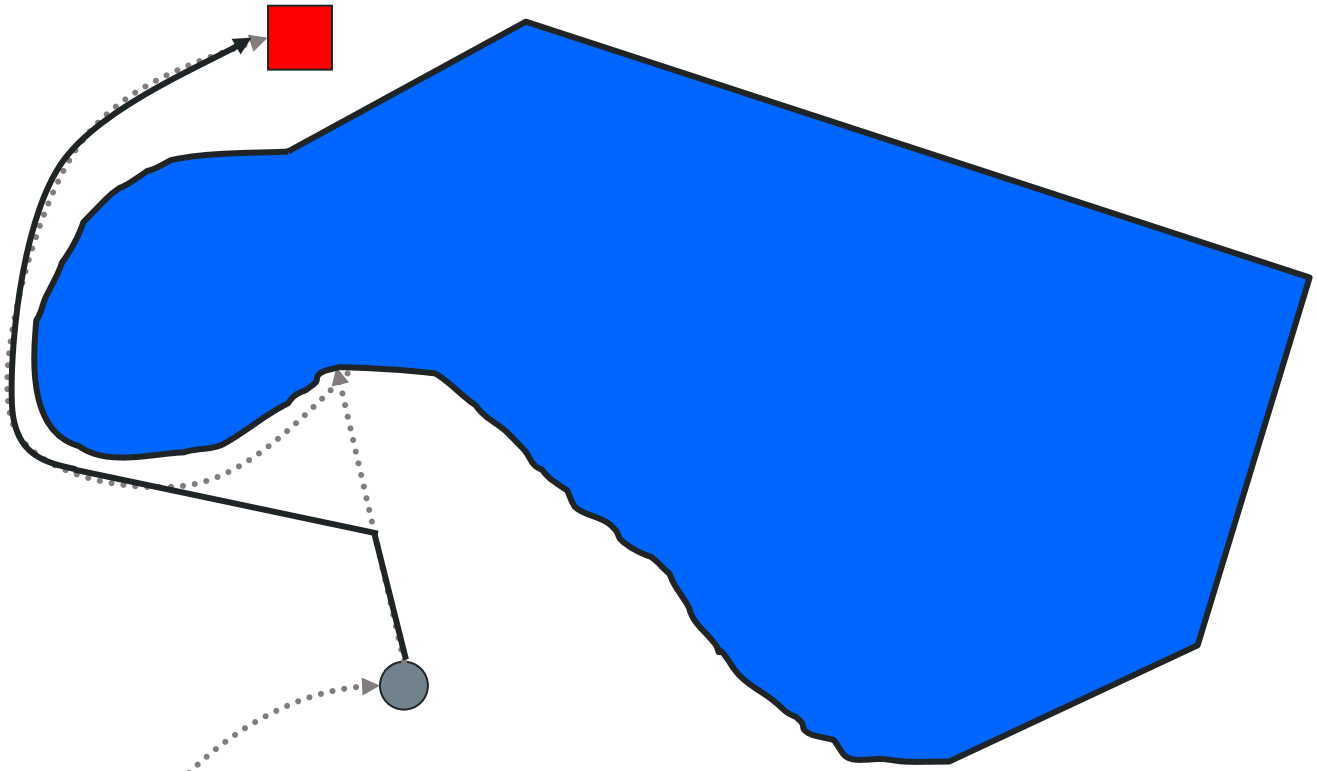
The full path is generated from the end of the quick path to the destination

The splice path



The splice path connects from the point on the quick path when the full path became available to the full path

The final path



The final path consists of an initial segment of the quick path, the splice path, and a final segment of the full path

Managing pathfinding priority

- Some ways of allocating CPU resources among the pathfinders
  - Equal: each active pathfinder gets equal time (equal number of revolutions per tick)
  - Progressive: start each pathfinder with a low number of revolutions per tick – increase this (with cap) the longer the pathfinder stays active (long jobs get more resources over time)
  - Biased – give some pathfinders, like those working on quickpaths, higher priority

- If you run out of quickpath before you've generated the full path, you'd better crank up the priority of the full path pathfinder (otherwise your agent looks stupid just sitting there at the end of the quick path)