CS540 Intro to AI
Uninformed Search

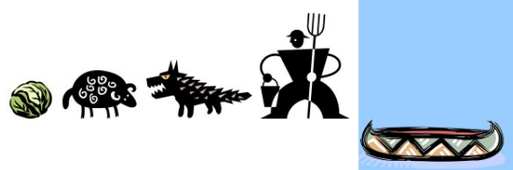
Yingyu Liang

University of Wisconsin-Madison

Slides created by Xiaojin Zhu (UW-Madison),
lightly edited by Anthony Gitter

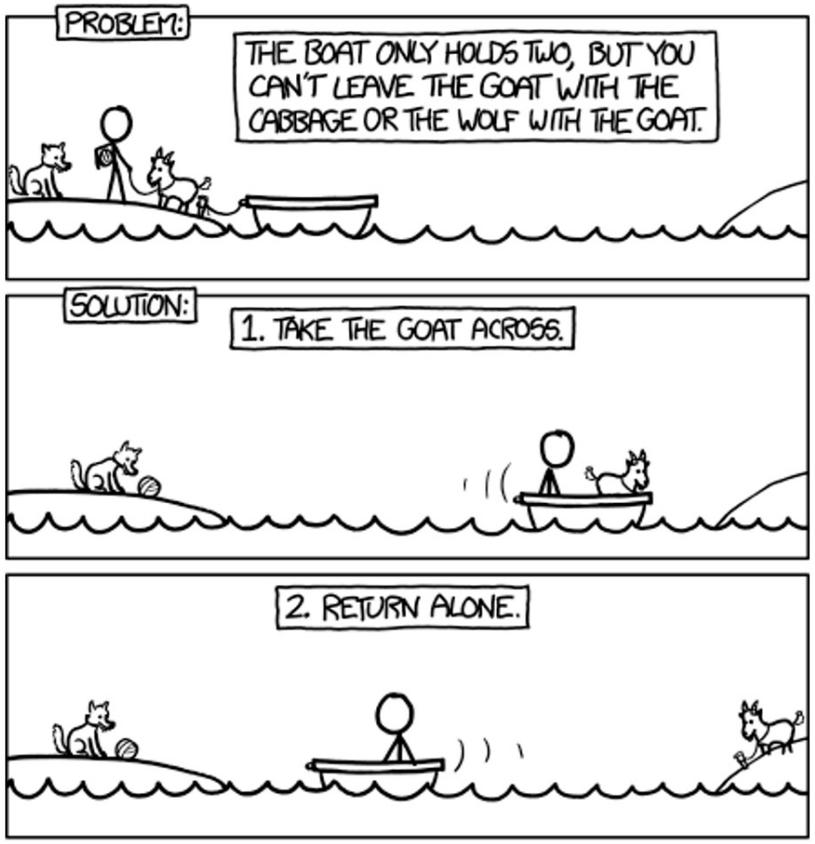
Many AI problems can be formulated as search.

Search is a general problem-solving framework. We will see several examples.



slide 3

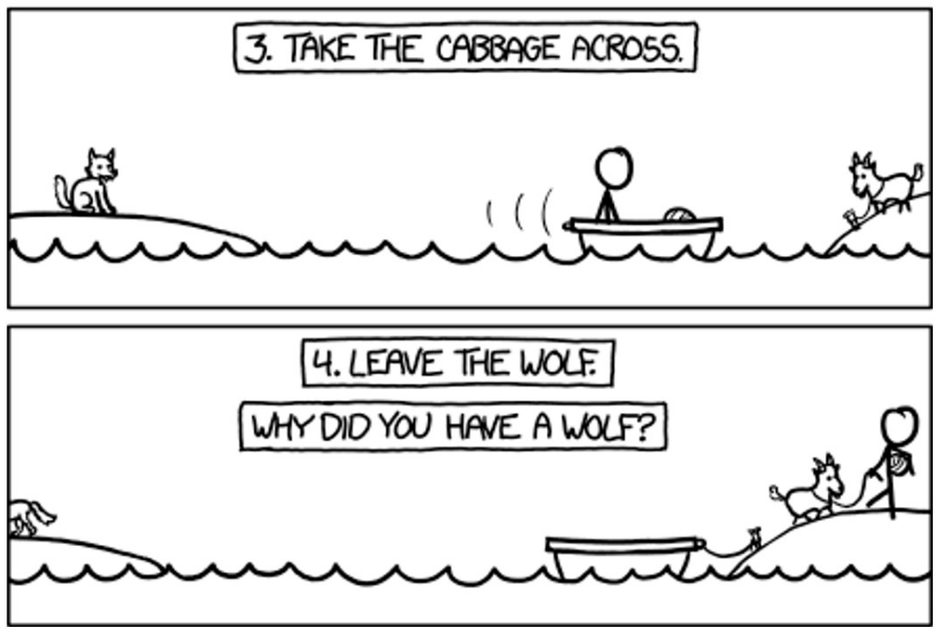
Let’s consider this scenario when a farmer is trying to cross the river with a wolf, a sheep and a cabbage. There are two conditions, the sheep cannot stay along with the cabbage and the wolf cannot stay alone with the sheep. The boat can only hold at most two at a time, and the farmer is the only one who can handle the boat.



http://xkcd.com/1134/

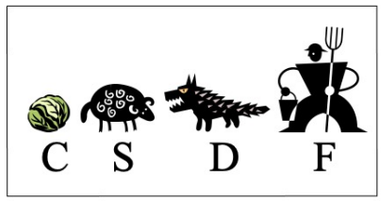
We can let the farmer take the goat across and then return.

http://xkcd.com/1134/



Of course this is a not a real solution. At least for the design of the original problem, we would like to bring the wolf across the river. So let’s be more specific about the problem by giving a formal description.

The search problem



- State space S : all valid configurations
- Initial state I={(CSDF,)} $\subseteq S$
- Goal state G={(,CSDF)} $\subseteq S$
- Successor function *succs*(s)$\subseteq S$: states reachable in one step from state s
  - *succs*((CSDF,)) = {(CD, SF)}
  - *succs*((CDF,S)) = {(CD,FS), (D,CFS), (C, DFS)}
- Cost(s,s')=1 for all steps. (weighted later)
- The search problem: find a solution path from a state in I to a state in G.
  - Optionally minimize the cost of the solution.

Let's first enumerate all the possible situations or configurations; call them the state space.

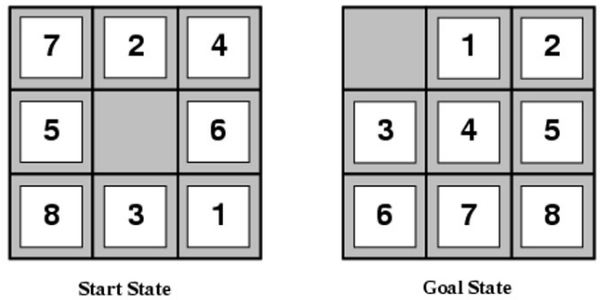
Initial state: Can have multiple initial states.
Goal state: a situation we want to achieve. (Can have multiple goal states.)

What actions can we perform? Successor function returns States reachable one step away from s, which can be 0 or more.

Cost on an action: usually=1 for all steps, but can have general weights.

Search examples

- 8-puzzle



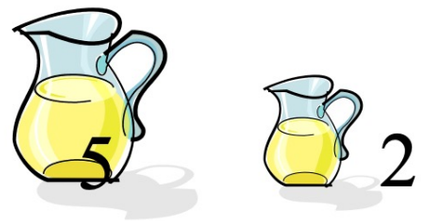
- States = 3x3 array configurations
- action = up to 4 kinds of movement
- Cost = 1 for each move

Given an initial configuration of 8 numbered tiles on a 3 x 3 board, move the tiles in such a way so as to produce a desired goal configuration of the tiles.

State = 3 x 3 array configuration of the tiles on the board. Operators: Move Blank square Left, Right, Up or Down. (Note: this is a more efficient encoding of the operators than one in which each of four possible moves for each of the 8 distinct tiles is used.) Initial State: A particular configuration of the board. Goal: A particular configuration of the board.

Search examples

- Water jugs: how to get 1?

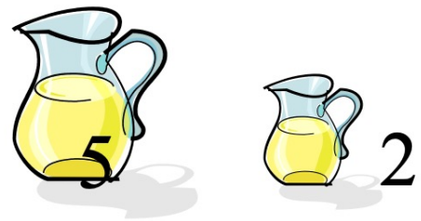


State = (x,y), where x = number of gallons of water in the 5-gallon jug and y is gallons in the 2-gallon jug
Initial State = (5,0)
Goal State = (*,1), where * means any amount

Given a 5-gallon jug and a 2-gallon jug, with the 5-gallon jug initially full of water and the 2-gallon jug empty, the goal is to fill the 2-gallon jug with exactly one gallon of water.

Search examples

- Water jugs: how to get 1?



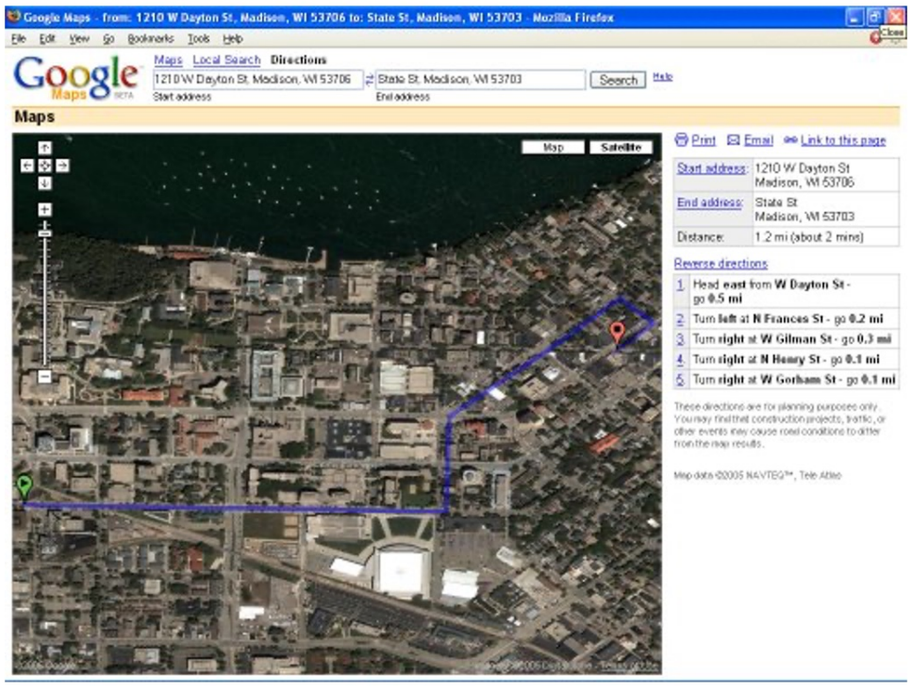
State = (x,y), where x = number of gallons of water in the 5-gallon jug and y is gallons in the 2-gallon jug
Initial State = (5,0)
Goal State = (*,1), where * means any amount
Operators
(x,y) -> (0,y) ; empty 5-gal jug
(x,y) -> (x,0) ; empty 2-gal jug
(x,2) and x<=3 -> (x+2,0) ; pour 2-gal into 5-gal
(x,0) and x>=2 -> (x-2,2) ; pour 5-gal into 2-gal
(1,0) -> (0,1) ; empty 5-gal into 2-gal

Given a 5-gallon jug and a 2-gallon jug, with the 5-gallon jug initially full of water and the 2-gallon jug empty, the goal is to fill the 2-gallon jug with exactly one gallon of water.

There could be many different operations/successors allowed. Here we list some for example.

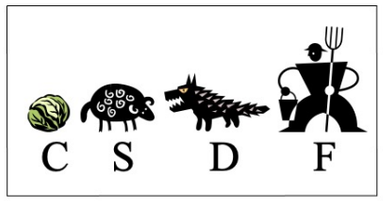
Search examples

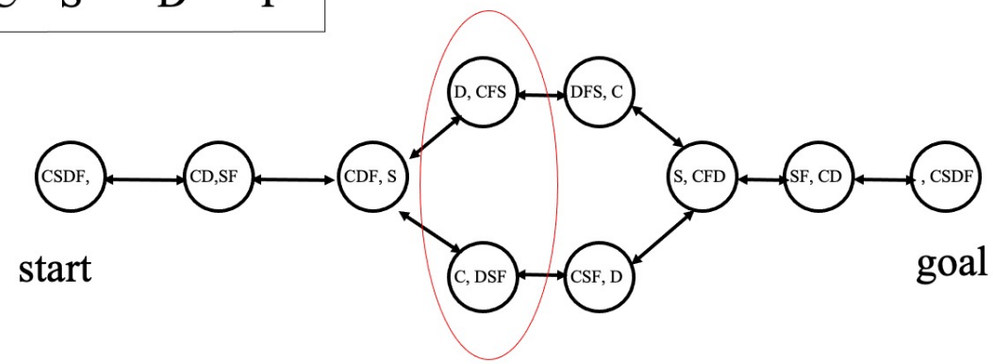
- Route finding (State? Successors? Cost weighted)



Another example is route finding. The states can be different locations (e.g., intersection of streets, places where action decisions need to be made). Notably, in this example, the cost can be weighted (e.g., time).

A directed graph in state space





- In general there will be many generated, but un-expanded states at any given time
- One has to choose which one to expand next

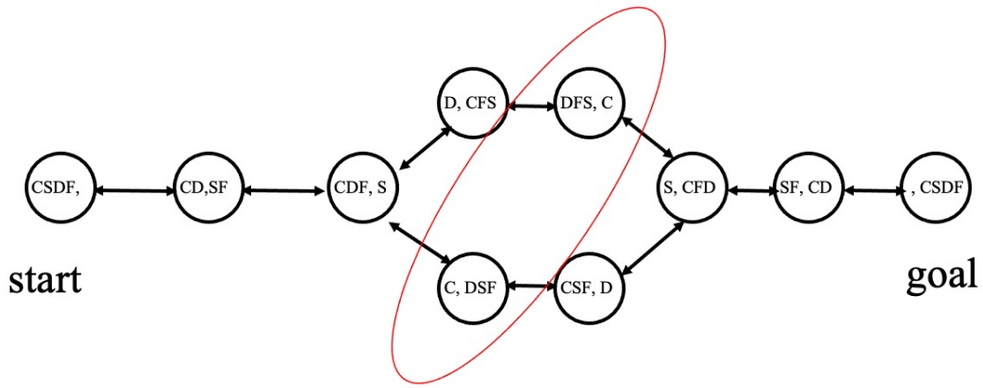
View states as nodes, view successor function as directed edges, then we can get a directed graph.

Suppose we are now in the state (CDF,S), and by the successor function, we know three successors (CD, SF), (D,CFS) and (C,DSF). We have been to (CD, SF) before, so we only keep the two states (D,CFS) and (C,DSF) under consideration.

expanded states: those states we have been to and applied successor functions
Fringe: those states generated from the successor function but not yet expanded, e.g., the two states (D,CFS) and (C,DSF) in the figure.

Different search strategies

- The generated, but not yet expanded states form the fringe (OPEN).
- The essential difference is which one to expand first.
- Deep or shallow?



Fringe (OPEN set): those states in our mind (generated from the successor function, but not yet expanded)

Continuing the example in the previous slide: if we decide to go to the state (D,CFS) and check its successor, then now the fringe is {(C,DSF), (DFS,C)}. Note that (C, DSF) is still in the fringe.

Different ways to select nodes from the fringe to expand lead to different search algorithms with different properties.

Uninformed search on trees

- Uninformed means we only know:
  - The goal test
  - The *succs*() function
- But not which non-goal states are better: that would be informed search (next topic).
- For now, we also assume *succs*() graph is a tree.
  - Won't encounter repeated states.
  - We will relax it later.
- Search strategies: BFS, UCS, DFS, IDS
- Differ by what un-expanded nodes to expand

Formally we will talk about the simplest version of search which is uniformed search. Note the difference between uninformed and informed search.

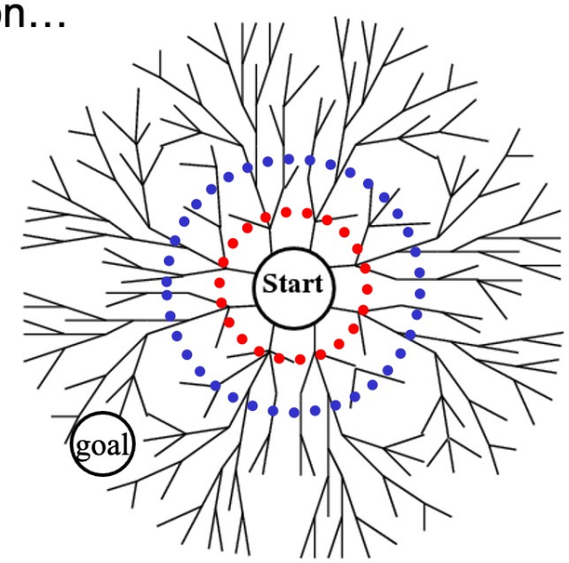
Simplification: consider the case when the search graph is a tree

Breadth-first search (BFS)

Expand the shallowest node first

- Examine states **one** step away from the initial states
- Examine states **two** steps away from the initial states
- and so on…

ripple



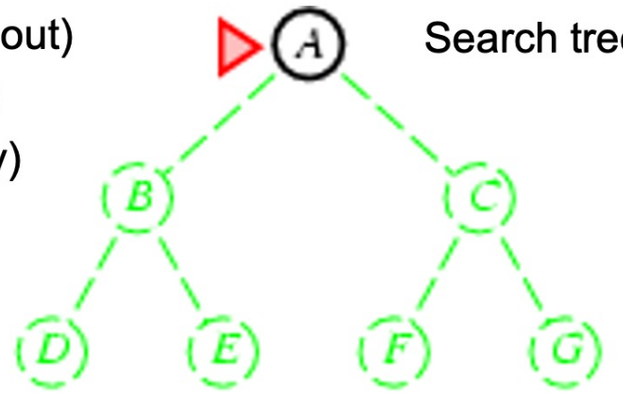
Breadth-first search (BFS): expand the shallowest node first.
High level intuition: like ripple in a pool

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile

Initial state: **A**
Goal state: **G**



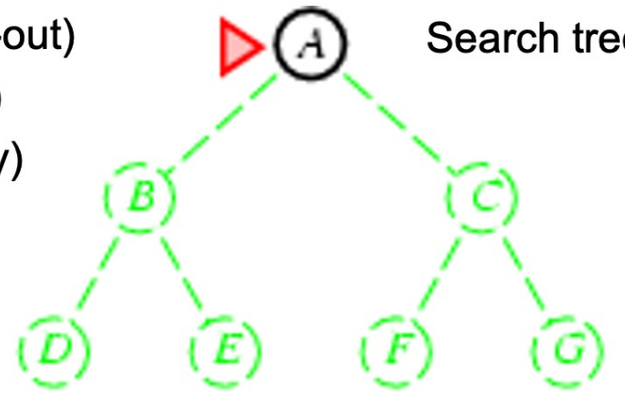
Search tree

BFS: implemented by queue, the FIFO data structure. The first-in nodes are shallower (closer in steps to the initial state) so are expanded first.

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

queue (fringe, OPEN)
→ [A] →

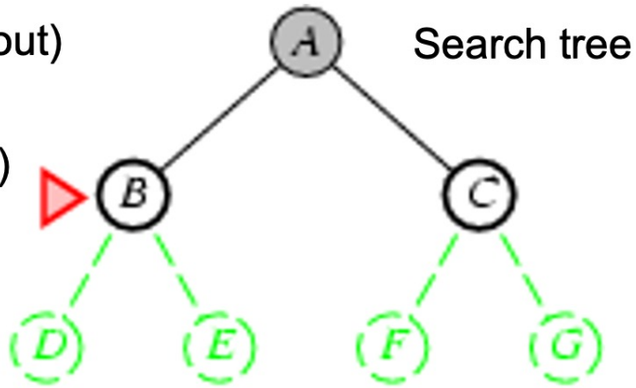
Initial state: **A**
Goal state: **G**

At the beginning, put the init state in the queue

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

queue (fringe, OPEN)
→ [CB] → A

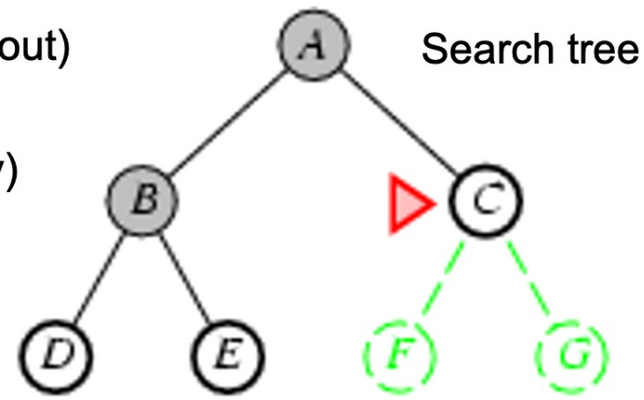
Initial state: **A**
Goal state: **G**

Here we show iteration 1:
the state checked: A
the queue at the end of the iteration: [CB]
The right hand side of the queue is the front of the queue.

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

queue (fringe, OPEN)
→ [EDC] → B

Initial state: **A**
Goal state: **G**

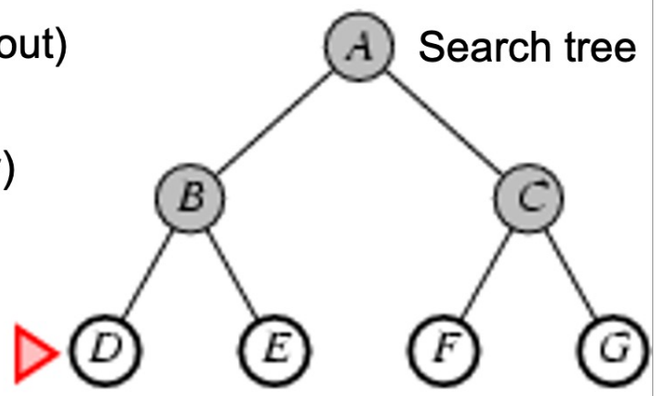
Iteration 2

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile

Initial state: **A**
Goal state: **G**



Search tree

queue (fringe, OPEN)
→[GFED] → C

If G is a goal, we've seen it, but we don't stop!

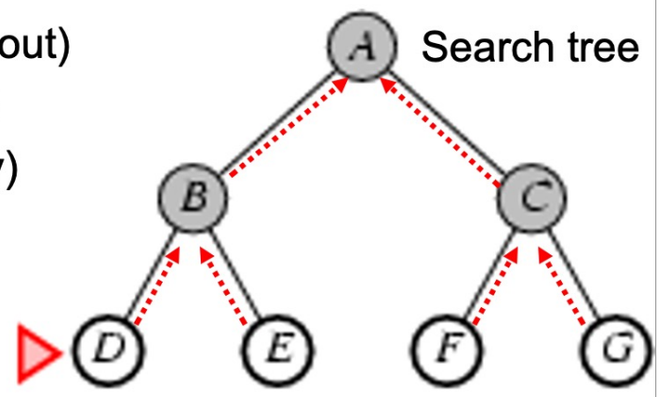
Iteration 3.

Note that we generate G and put it in the queue but haven’t checked it yet.

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

queue
→[] →G

... until much later we pop G.

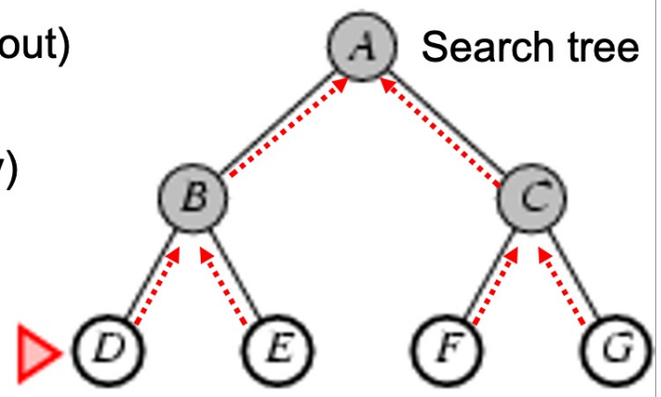
Looking foolish?
Indeed. But let's be consistent…

Iteration 4: [GFE]-> D
Iteration 5: [GF]-> E
Iteration 6: [G]-> F
Iteration 7: []-> G. Success!

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

queue
→[] →G

... until much later we pop G.

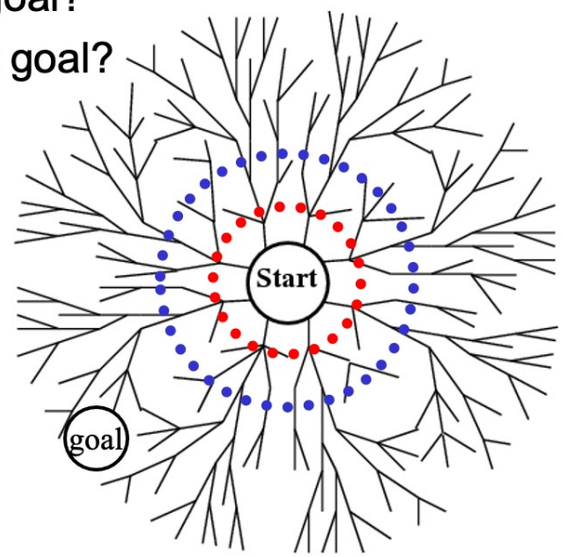
Looking foolish?
Indeed. But let's be consistent…

We need back pointers to recover the solution path.

If we would like to return the solution path, we can keep a back point for each node that we generated. Then when succeed, we can trace back the path.

Performance of BFS

- Assume:
  - the graph may be infinite.
  - Goal(s) exists and is only finite steps away.
- Will BFS find at least one goal?
- Will BFS find the least cost goal?
- Time complexity?
  - # states generated
  - Goal *d* edges away
  - Branching factor *b*
- Space complexity?
  - # states stored



A few fundamental questions about the performance.

Note that the time complexity is roughly proportional to total # states generated. To describe that we introduce parameters d and b.
The space complexity is roughly proportional to the max # states stored at any time point during the execution of the algorithm.

Performance of BFS

Four measures of search algorithms:

- Completeness (not finding all goals): yes, BFS will find a goal.
- Optimality: yes if edges cost 1 (more generally positive non-decreasing in depth), no otherwise.
- Time complexity (worst case): goal is the last node at radius d.
  - Have to generate all nodes at radius d.
  - $b + b^2 + \ldots + b^d \sim O(b^d)$
- Space complexity (bad)
  - Back pointers for all generated nodes $O(b^d)$
  - The queue / fringe (smaller, but still $O(b^d)$)

If all edge cost are identical, then optimal.

More generally, if the cost is positive and non-decreasing in depth then optimal (we don’t require to understand the general case in this course).

O(b^d): big-O notation. It means that it’s roughly (C*b^d + smaller terms) for some constant C

If all edge cost are identical, then optimal.

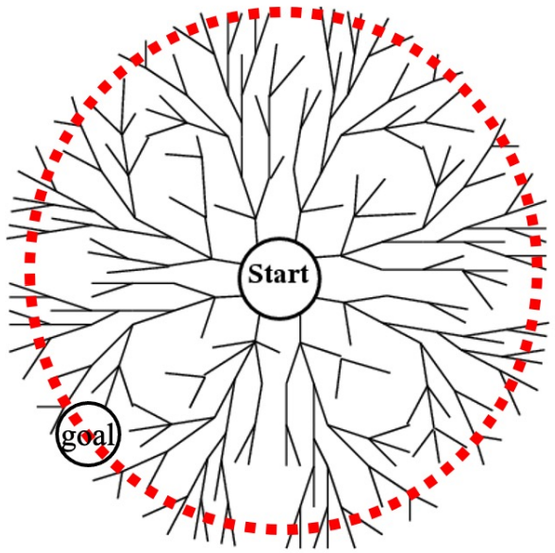
More generally, if the cost is positive and non-decreasing in depth then optimal (we don’t require to understand the general case in this course).

O(b^d): big-O notation. It means that it’s roughly (C*b^d + smaller terms) for some constant C.

In the worst case, we need to generate all nodes within radius d+1 (including the children of the nodes at radius d). Then the time is b+b^2+ ... + b^d + b^{d+1} = O(b^d) viewing b as a constant.

What's in the fringe (queue) for BFS?

- Convince yourself this is $O(b^d)$



The space complexity is roughly proportional to the max # states stored at any time point during the execution of the algorithm. In the worst case, at some point it can store all nodes the same depth as the goal state.

Performance of search algorithms on trees

b: branching factor (assume finite) d: goal depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$

1. Edge cost constant, or positive non-decreasing in depth

Q1-1: You are running BFS on a finite tree-structured state space graph that does not have a goal state. What is the behavior of BFS?

1. Visit all N nodes, then return one at random
2. Visit all N nodes, then stop and return "failure"
3. Visit all N nodes, then return the node farthest from the initial state
4. Get stuck in an infinite loop

Q1-1: You are running BFS on a finite tree-structured state space graph that does not have a goal state. What is the behavior of BFS?

1. Visit all N nodes, then return one at random
2. Visit all N nodes, then stop and return "failure" 
3. Visit all N nodes, then return the node farthest from the initial state
4. Get stuck in an infinite loop

Performance of BFS

Four measures of search algorithms:

Solution: Uniform-cost search

- Completeness (not finding all goals): find a goal.
- Optimality: yes if edges cost 1 (more generally positive non-decreasing with depth), no otherwise.
- Time complexity (worst case): goal is the last node at radius d.
  - Have to generate all nodes at radius d.
  - $b + b^2 + \ldots + b^d \sim O(b^d)$
- Space complexity (bad, see the Figure)
  - Back points for all generated nodes $O(b^d)$
  - The queue (smaller, but still $O(b^d)$)

One drawback: may not be optimal

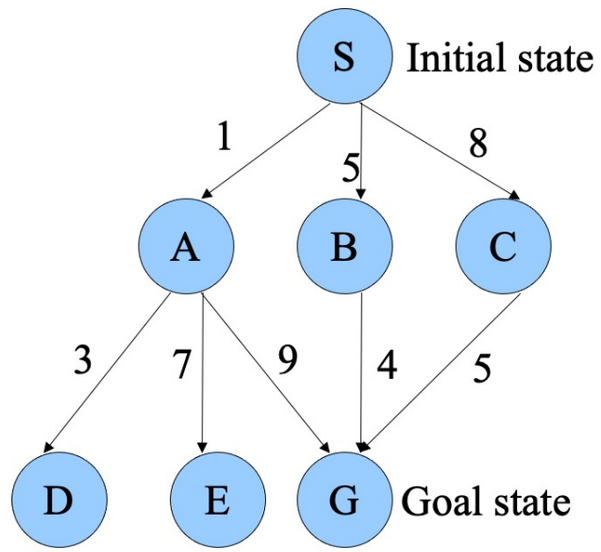
Uniform-cost search

- Find the least-cost goal
- Each node has a path cost from start (= sum of edge costs along the path).
- Expand the least cost node first.
- Use a priority queue instead of a normal queue
  - Always take out the least cost item

Each time pick the least-cost node in the fringe to expand.

Need to keep the cost of each node when generated. Use the priority queue data structure: priority queue is designed exactly for the purpose of picking the least-cost item.

Example



(All edges are directed, pointing downwards)

Example for UCS.

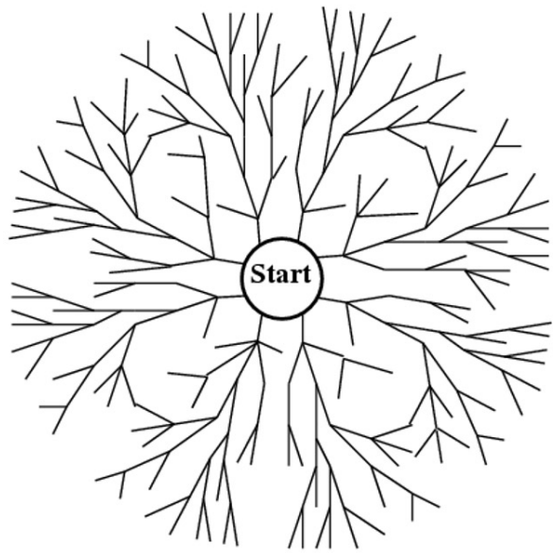
Iteration, node expanded, fringe at the end of the iteration (nodes not sorted; nodes are in the format (id, cost))
1: (S,0), [(A,1), (B,5), (C,8)]
2: (A,1), [(B,5), (C,8), (D,4), (E,8), (G,10)]
3: (D,4), [(B,5), (C,8), (E,8), (G,10)]
4: (B,5), [(C,8), (E,8), (G,9)]
At iteration 4, we will get a new copy of G: (G,9). We can keep both copies: the old (G,10) and the new (G,9). We can also keep only the least-cost copy: (G,9). Here we do the latter for simplicity. Note that if we keep a back pointer, we also need to update the back point accordingly.

5: (C,8), [(E,8), (G,9)]
At iteration 5, we need to break ties between (C,8) and (E,8). Here we use the dictionary order: C before E. Also, note that from C we generate a new copy of G: (G, 13). However, its cost is larger than the old copy (G,9), so we only keep (G,9).

6: (E,8), [(G,9)]
7: (G,9), []: Success!

Uniform-cost search (UCS)

- Complete and optimal (if edge costs $\geq \varepsilon > 0$)
- Time and space: can be much worse than BFS
  - Let C^* be the cost of the least-cost goal
  - $O(b^{C^*/\varepsilon})$



Recall that the time complexity is proportional to total #nodes generated before success. It can be all the nodes with smaller or equal cost as the goal state. These nodes are at most C*/\epsilon steps away from the initial state, so in the worst case there can be roughly b^{C*/\epsilon} so many of them.

Similar reasoning for the space complexity.

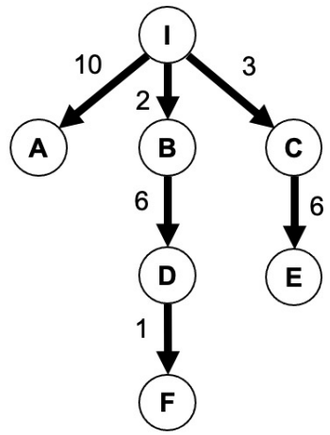
Performance of search algorithms on trees

b: branching factor (assume finite) d: goal depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$

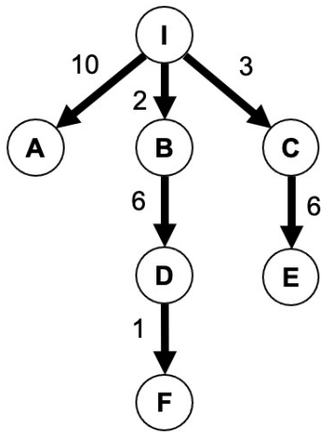
1. edge cost constant, or positive non-decreasing in depth
2. edge costs $\geq \varepsilon > 0$. C* is the best goal path cost.

Q1-2: You are running UCS in the state space graph below. You just called the successor function on node D. What is the cost of node F?



1. 2
2. 7
3. 8
4. 9

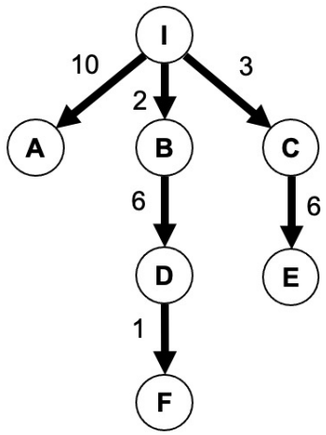
Q1-2: You are running UCS in the state space graph below. You just called the successor function on node D. What is the cost of node F?



1. 2
2. 7
3. 8
4. 9 ➡

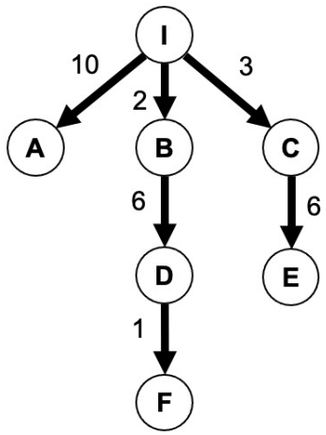
The cost is simply the sum of the edge costs along the path from the initial state to D and then to F.

Q1-3: You are running UCS in the state space graph below. You just expanded (visited) node C. What node will the search expand next?



1. A
2. D
3. E
4. F

Q1-3: You are running UCS in the state space graph below. You just expanded (visited) node C. What node will the search expand next?



1. A
2. D
3. E
4. F

UCS has the property that it will always check a smaller cost node before larger cost node.

I and B have smaller cost than C, so they must have been expanded before C. D has smaller cost than E F A, so D must be expanded before E F A.