

CS540 Intro to AI
Uninformed Search

University of Wisconsin-Madison

Fall 2026

Sections 1 and 2

Announcements

Homework 7 due Friday Nov 13 at 11:59PM.

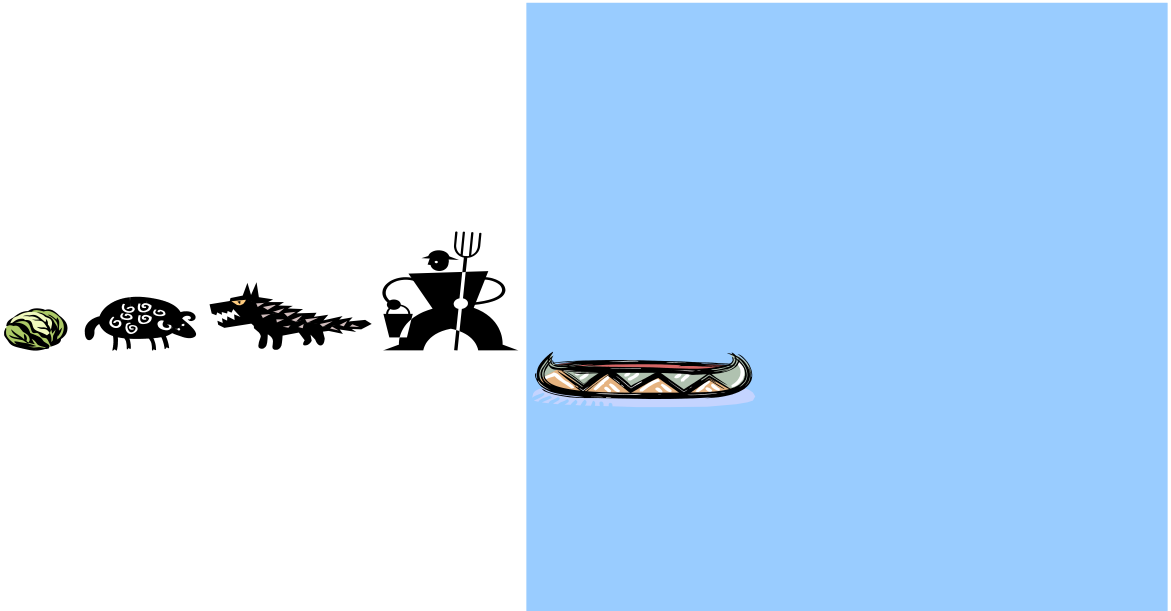
Change of schedule: no review of NN/DL.

Class roadmap and schedule:

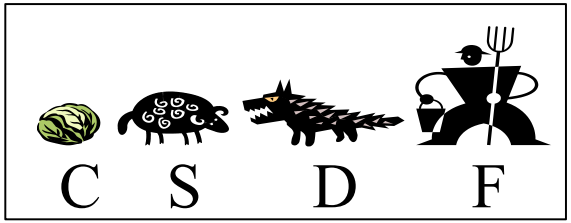
Search I
Search II

Many AI problems can be formulated as search.

You, a farmer, have a dog, a sheep, and a cabbage and are trying to cross the river. You cannot leave the sheep alone with the cabbage and the you cannot leave the dog alone with the sheep. The boat can only hold at most two at a time and you have to be on the boat to ride the boat. How do you cross the river?



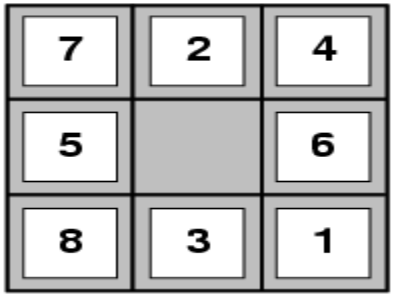
The search problem



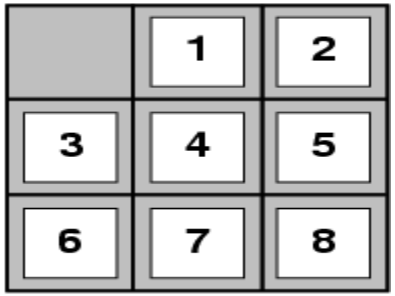
- State space S : all valid configurations
- Initial state I={(CSDF,)} $\subseteq S$
- Goal state G={(,CSDF)} $\subseteq S$
- Successor function $succs$(s)$\subseteq S$: states reachable in one step from state s
  - $succs$((CSDF,)) = {(CD, SF)}
  - $succs$((CDF,S)) = {(CD,FS), (D,CFS), (C, DFS)}
- Cost(s,s')=1 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.

Search examples

- 8-puzzle



Start State



Goal State

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

Search examples

How do you measure 1 gallon water from 5 and 2 gallon jars?



5



2

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

Search examples

How do you measure 1 gallon water from 5 and 2 gallon jars?



5 2

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

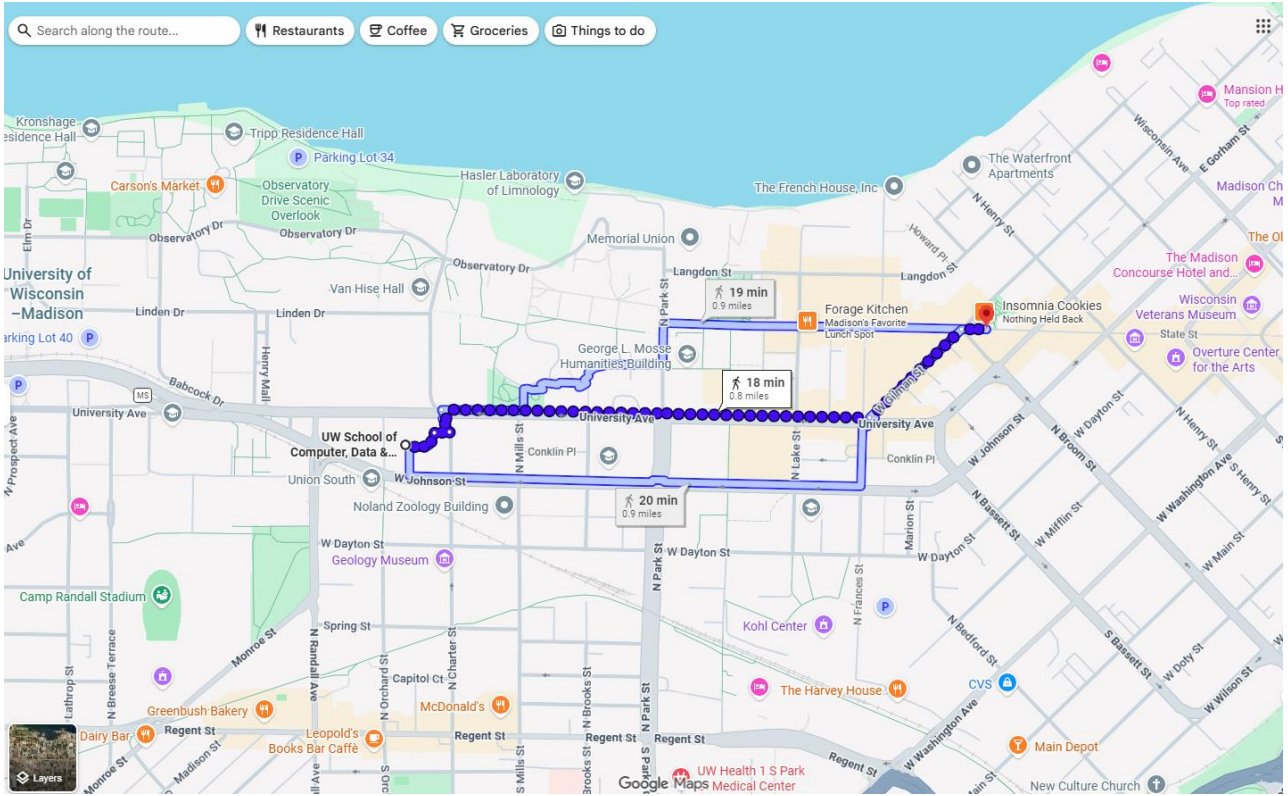
(x,2) and x<=3 -> (x+2,0) ; pour 2-gal into 5-gal

(x,0) and x>=2 -> (x-2,2) ; pour 5-gal into 2-gal

(x,0) -> (0,x) ; empty 5-gal into 2-gal

Search examples

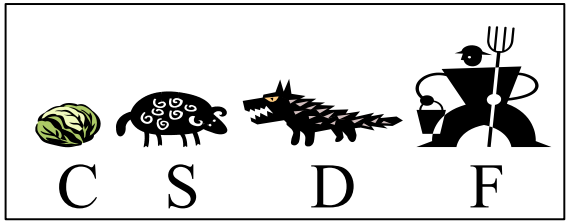
- Route finding (State? Successors? Cost weighted)

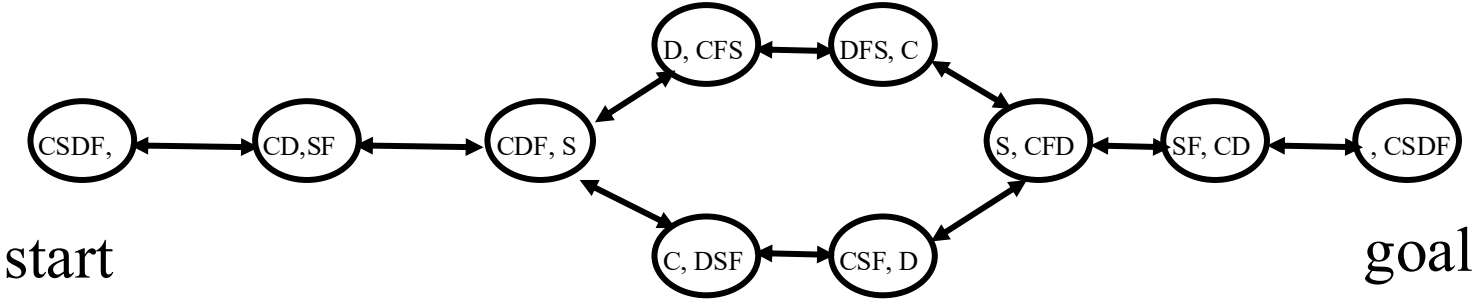


Which of these examples are similar to the above? Are the costs from one state to another weighted?

- Solving a Rubik's cube
- Finding a driving/cycling/walking route from Morgridge Hall to the Capitol (in the shortest possible time).
- Playing a game of chess
- Instructing a robot to open a door

A directed graph in state space





- In general there will be many generated, but un-**expanded** (i.e. not yet visited) states at any given time
- One has to choose which states to explore next
- Different search strategies differ on how to do so.

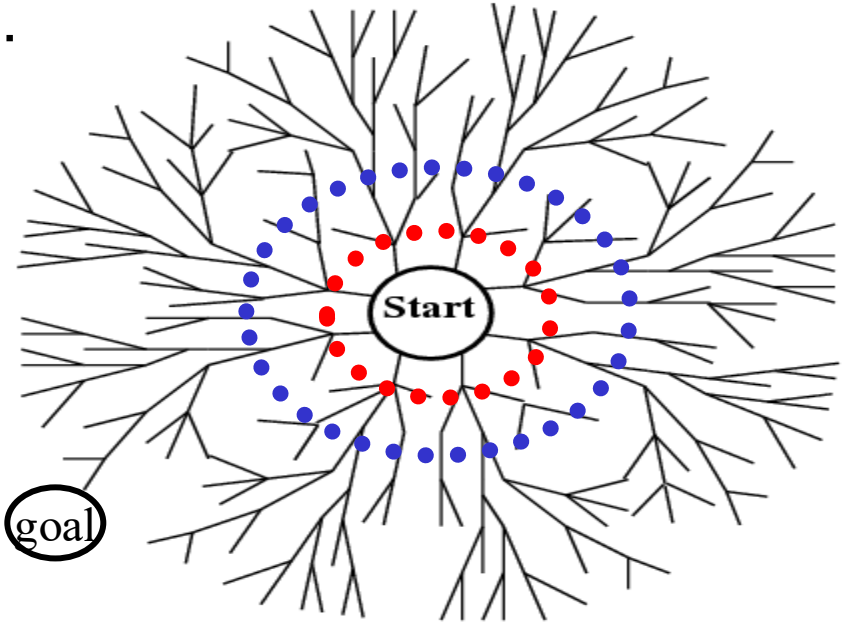
Uninformed search on trees

- Uninformed means we only know:
  - The goal (or a test for a goal)
  - The ***succs***() function
- But no idea/heuristic on which non-goal states are better: that would be informed search (next lecture).
- 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

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…



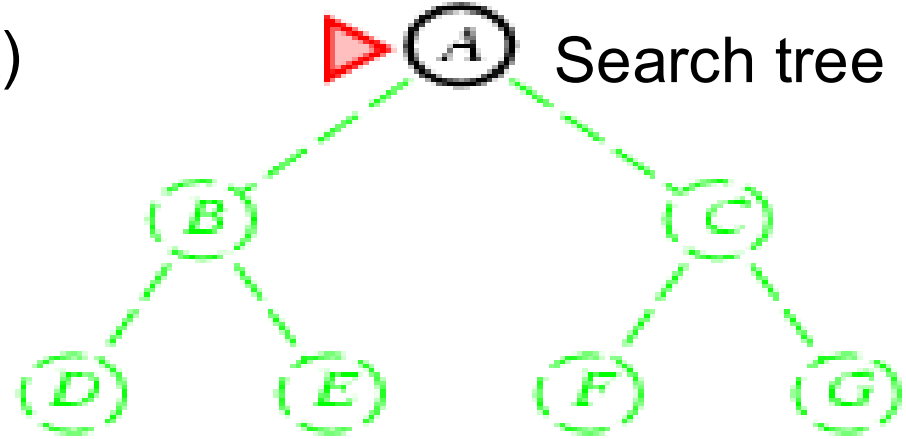
Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial state)
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

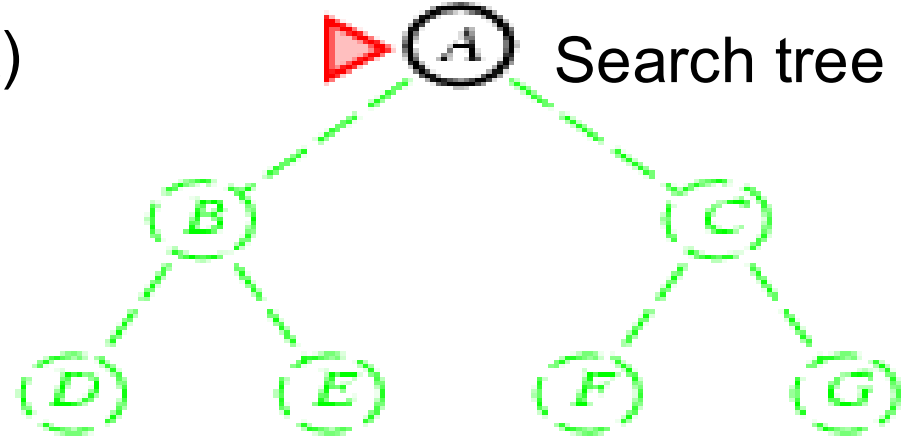
Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial state)
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)

→ [A] →

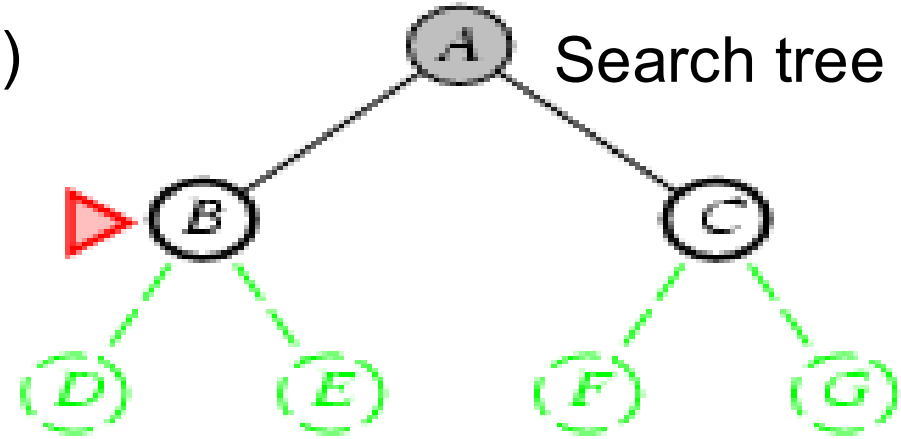
Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial state)
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)

→ [CB] → A

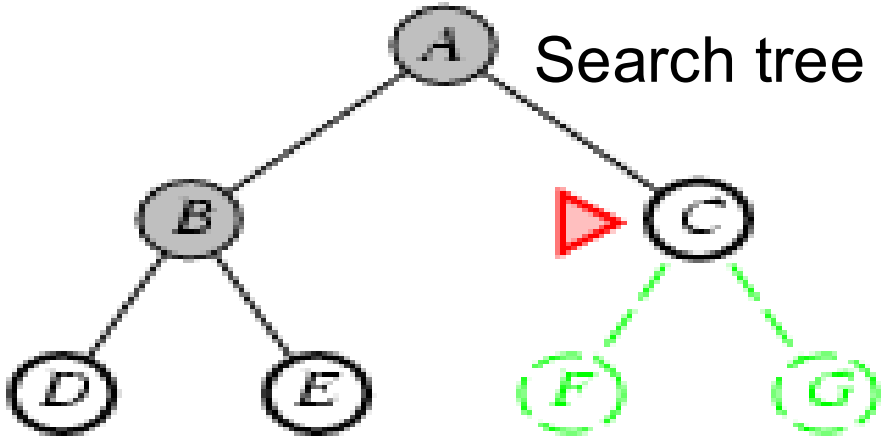
Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial state)
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)

→ [EDC] → B

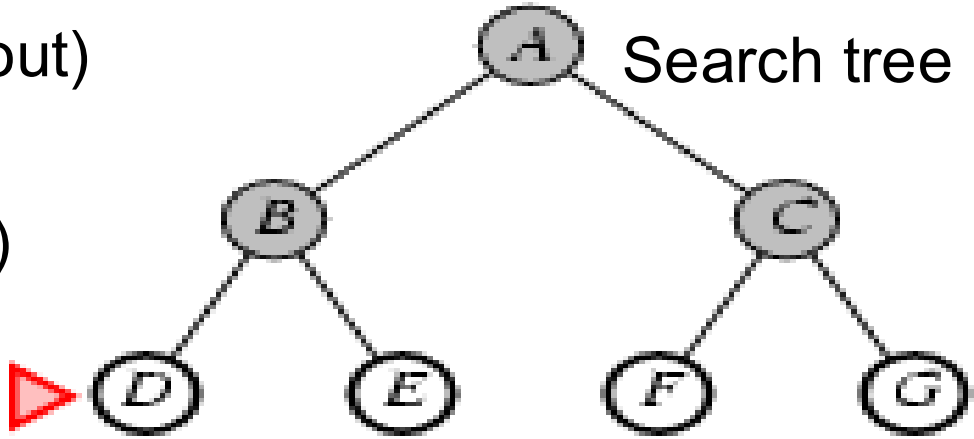
Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial state)
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)

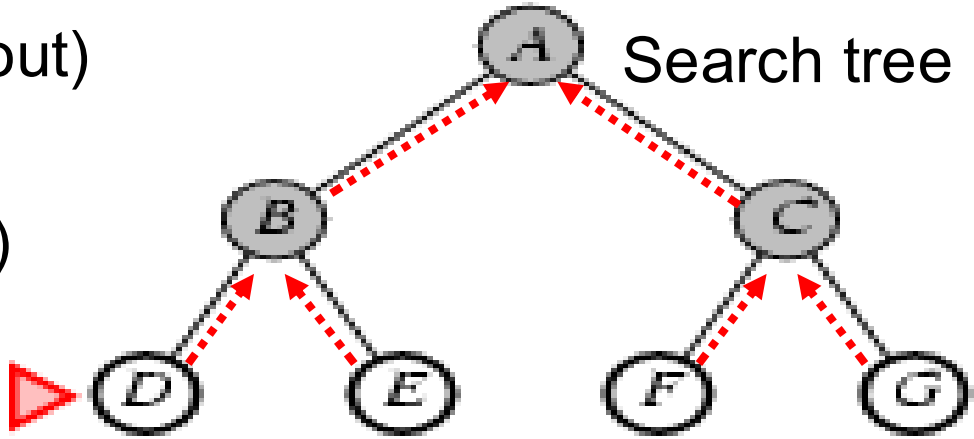
→[GFED] → C

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial state)
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**



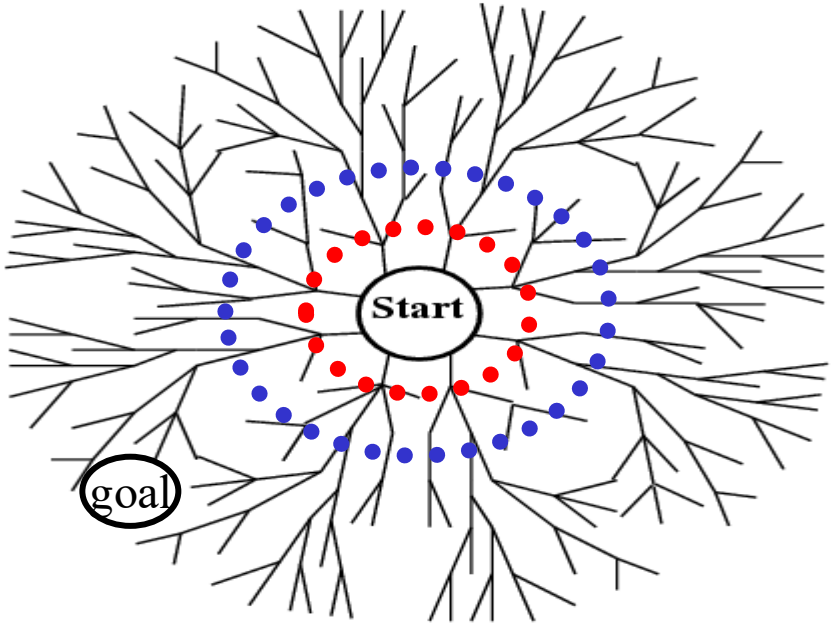
Search tree

queue
→[] →G

We need back pointers to recover the solution 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



Performance of BFS

Four measures of search algorithms:

- Completeness (find at least one goal): 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 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 $O(b^d)$)

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

Performance of BFS

Solution: Uniform-cost search

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 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)
  - Back points for all generated nodes $O(b^d)$
  - The queue (smaller, but still $O(b^d)$)

Uniform-cost search

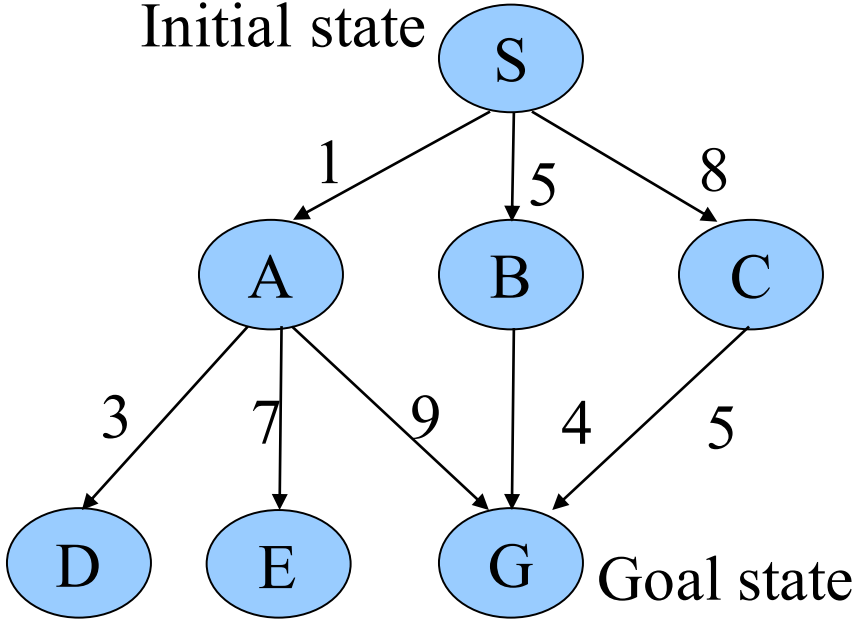
- Find the least-cost goal
- Each node has a path cost from start (= sum of edge costs along the path).
- Expand the least cost node first.
- Use a priority queue instead of a normal queue
  - Always take out the least cost item

Breadth-first search (BFS)

Use a priority-queue

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

Example

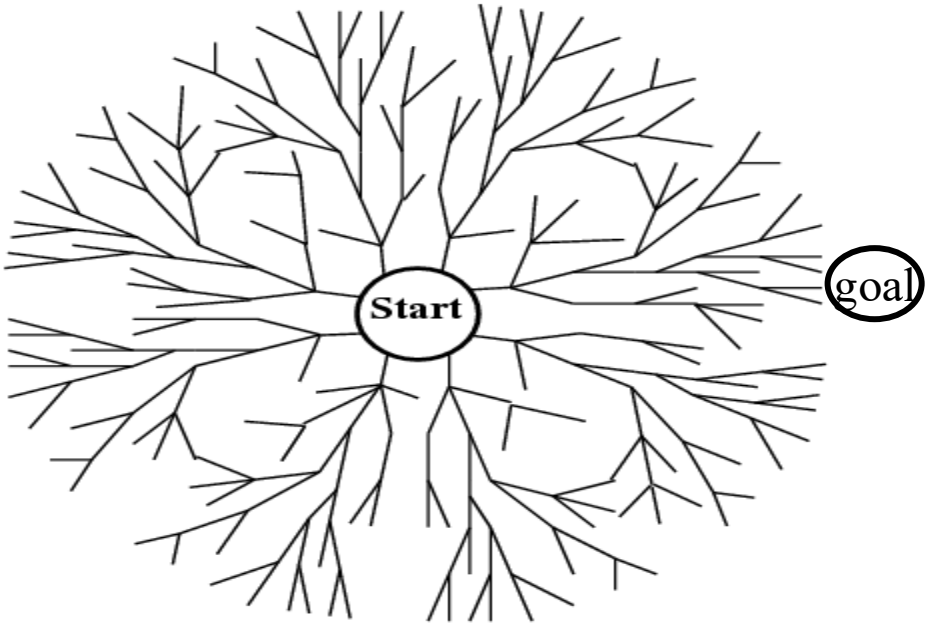


1: (S,0), [(A,1), (B,5), (C,8)]
2: (A,1), [(B,5), (C,8), (D,4), (E,8), (G,10)]
3: (D,4), [(B,5), (C,8), (E,8), (G,10)]
4: (B,5), [(C,8), (E,8), (G,9)]
5: (C,8), [(E,8), (G,9)]
6: (E,8), [(G,9)]
7: (G,9), []: Success!

(All edges are directed, pointing downwards)

Uniform-cost search (UCS)

- Complete and optimal (if edge costs $\geq \varepsilon > 0$)
- Time and space: $O(b^{C^*/\varepsilon})$
  - C^* is the cost of the least-cost goal



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.

Recall the bad space complexity of BFS

Four measures of search algorithms:

- Completeness (not fin… goal.
- Optimality: yes if edge… (…generally positive non-decreasing with depth), no other…se.
- Time complexity (worst case): goal is 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 points for all generated nodes $O(b^d)$
  - The queue (smaller, but still $O(b^d)$)

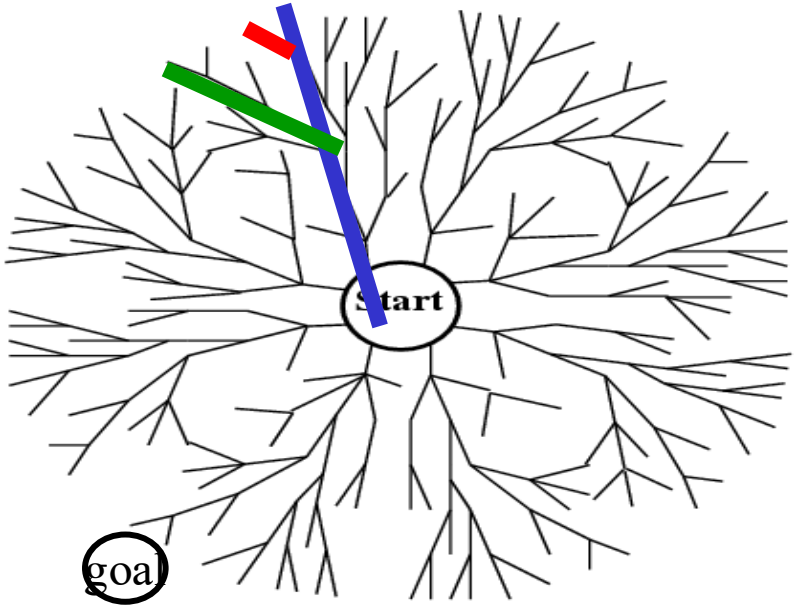
Solution: Depth-first search

Solution: Uniform-cost search

Depth-first search

Expand the deepest node first

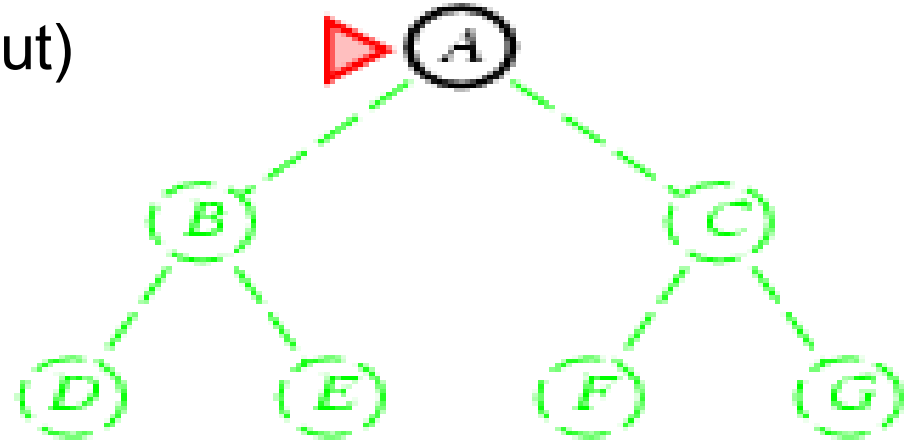
1. Select a direction, go deep to the end
2. Come back and go deep along a different path
3. Come back further, and go deep again



Depth-first search (DFS)

Use a stack (First-in Last-out)

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



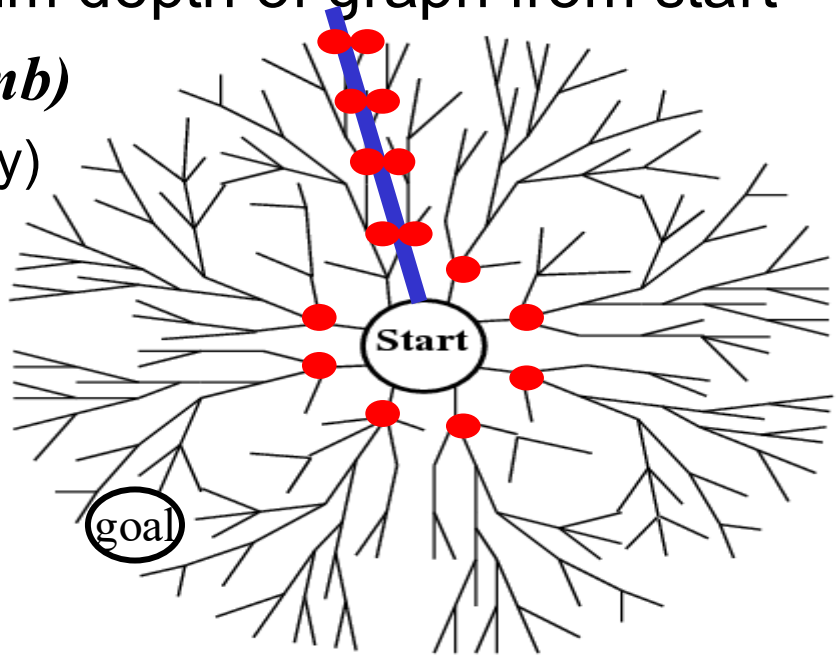
Fringe (Stack)

1. A, [B, C]
2. B, [D, E, C]
3. D, [E, C]
4. E, [C]
5. C, [F, G]
6. F, [G]
7. G

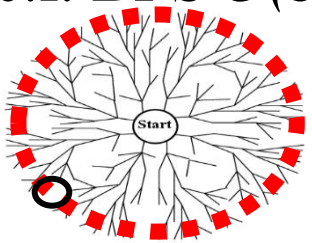
What's in the fringe for DFS?

- m = maximum depth of graph from start
- ***m(b-1) ~ O(mb)***

(Space complexity)

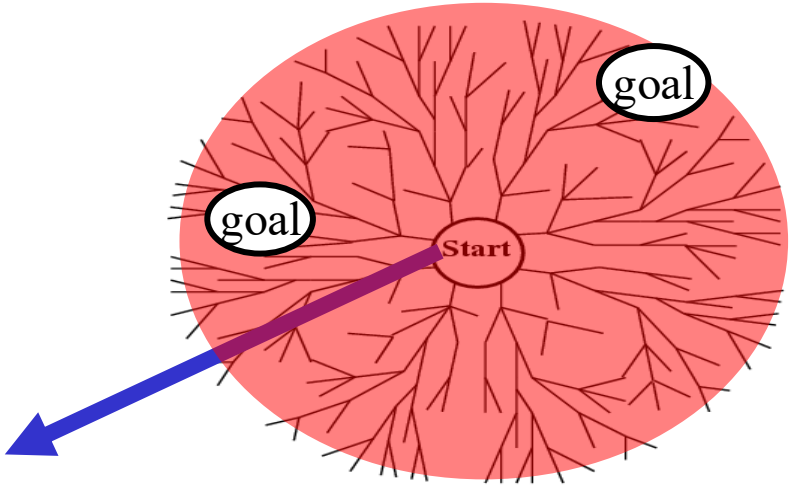


c.f. BFS $O(b^d)$

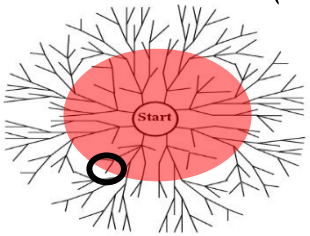


What's wrong with DFS?

- Infinite tree: may not find goal (incomplete)
- May not be optimal
- Finite tree: may visit almost all nodes, time complexity $O(b^m)$



c.f. BFS $O(b^d)$



Performance of search algorithms on trees

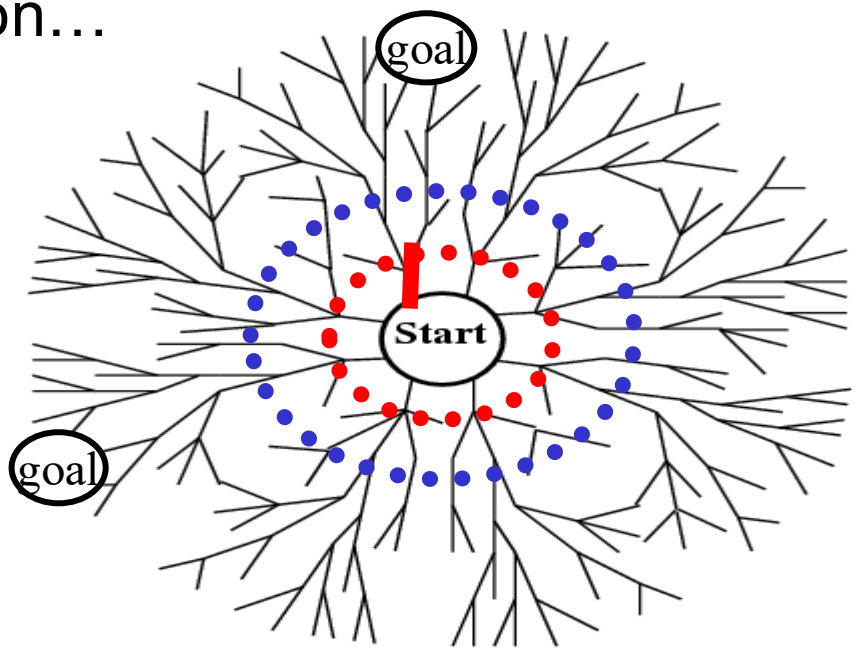
b: branching factor (assume finite) d: goal depth m: graph depth

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

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

Can you get the best of both worlds?

1. DFS, but stop if path length > 1.
2. If goal not found, repeat DFS, stop if path length > 2.
3. And so on…



Iterative deepening

- Search proceeds like BFS, but fringe is like DFS
  - Complete, optimal like BFS
  - Small space complexity like DFS
  - Time complexity like BFS
- Generally, preferred uninformed search method

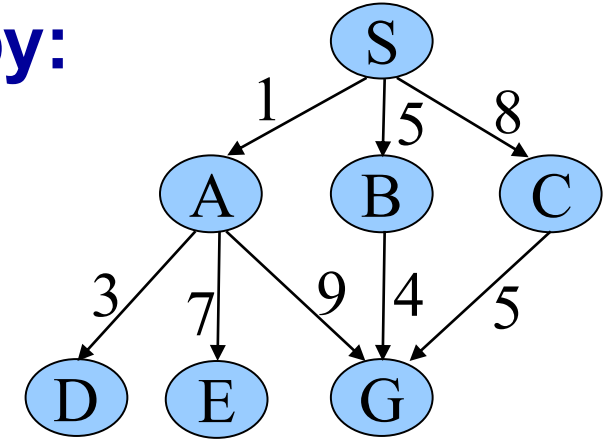
Performance of search algorithms on trees

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

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

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

Nodes expanded by:



- **Breadth-First Search:** S A B C D E G

  Solution found: S A G

- **Uniform-Cost Search:** S A D B C E G

  Solution found: S B G (only one that accounts for costs)

- **Depth-First Search:** S A D E G

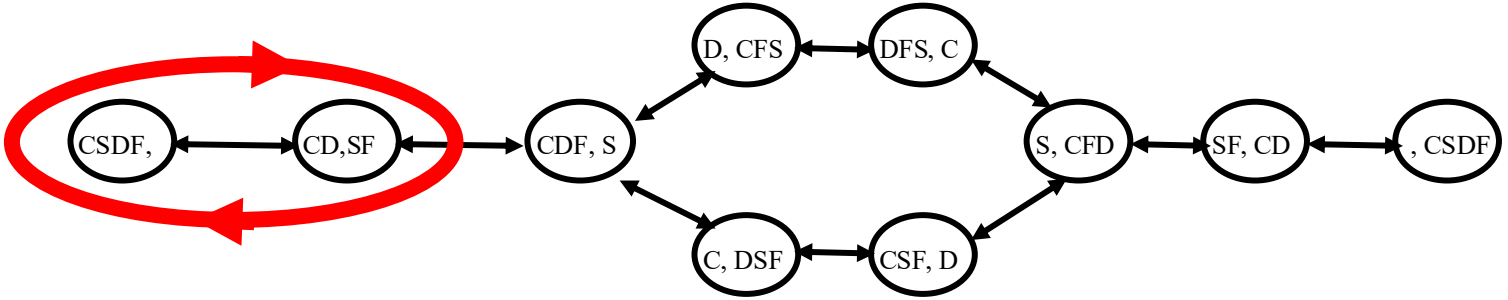
  Solution found: S A G

- **Iterative-Deepening Search:** S A B C S A D E G

  Solution found: S A G

If state space graph is not a tree

- The problem: repeated states



- Ignore the danger of repeated states: wasteful (BFS) or impossible (DFS). Can you see why?
- How to prevent it?

If state space graph is not a tree

- We have to remember already-expanded states (CLOSED).
- When we take out a state from the fringe (OPEN), check whether it is in CLOSED (already expanded).
  - If yes, throw it away.
  - If no, expand it (add successors to OPEN), and move it to CLOSED.

General State-Space Search Algorithm

```
function general-search(problem, QUEUEING-FUNCTION)
  ;; problem describes the start state, operators, goal test, and operator costs
  ;; queueing-function is a comparator function that ranks two states
  ;; general-search returns either a goal node or "failure"

  nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))

  loop
     if EMPTY(nodes) then return "failure"
     node = REMOVE-FRONT(nodes)
     if problem.GOAL-TEST(node.STATE) succeeds then return node
     nodes = QUEUEING-FUNCTION(nodes, EXPAND(node, problem.OPERATORS))
     ;; succ(s)=EXPAND(s, OPERATORS)
     ;; Note: The goal test is NOT done when nodes are generated
     ;; Note: This algorithm does not detect loops
  end
```