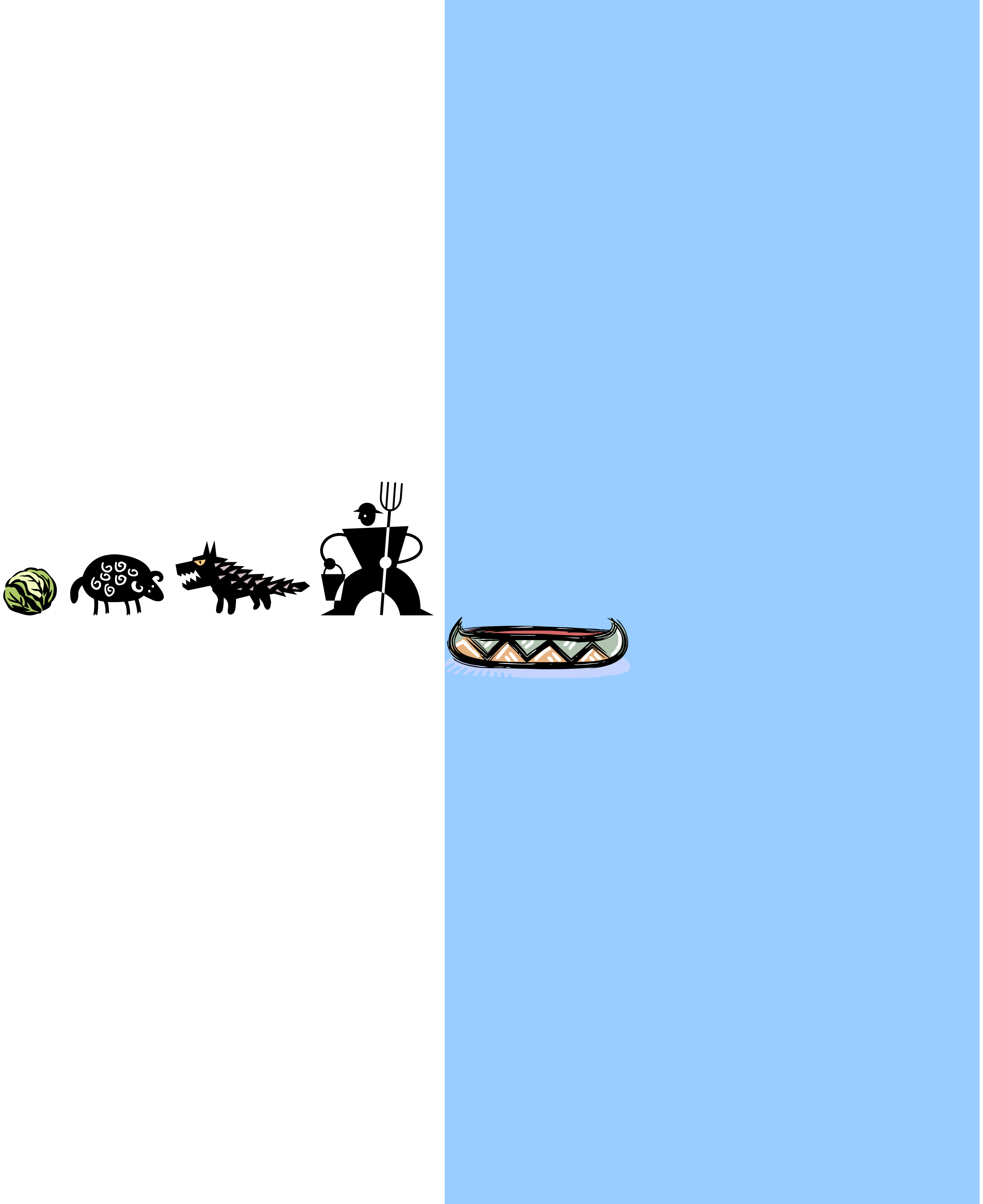
CS540 Intro to AI
Uninformed Search

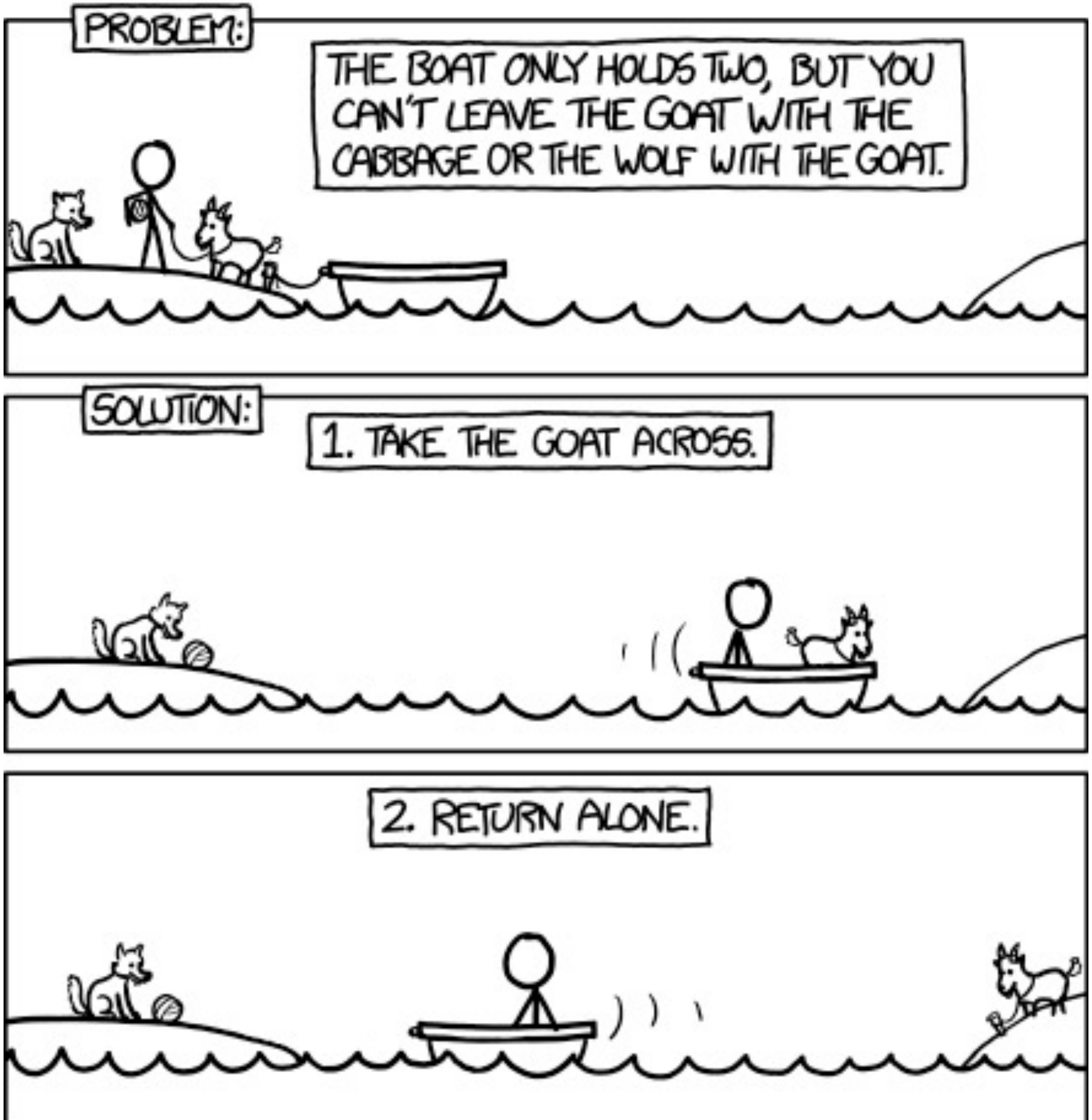
Yingyu Liang

University of Wisconsin-Madison

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

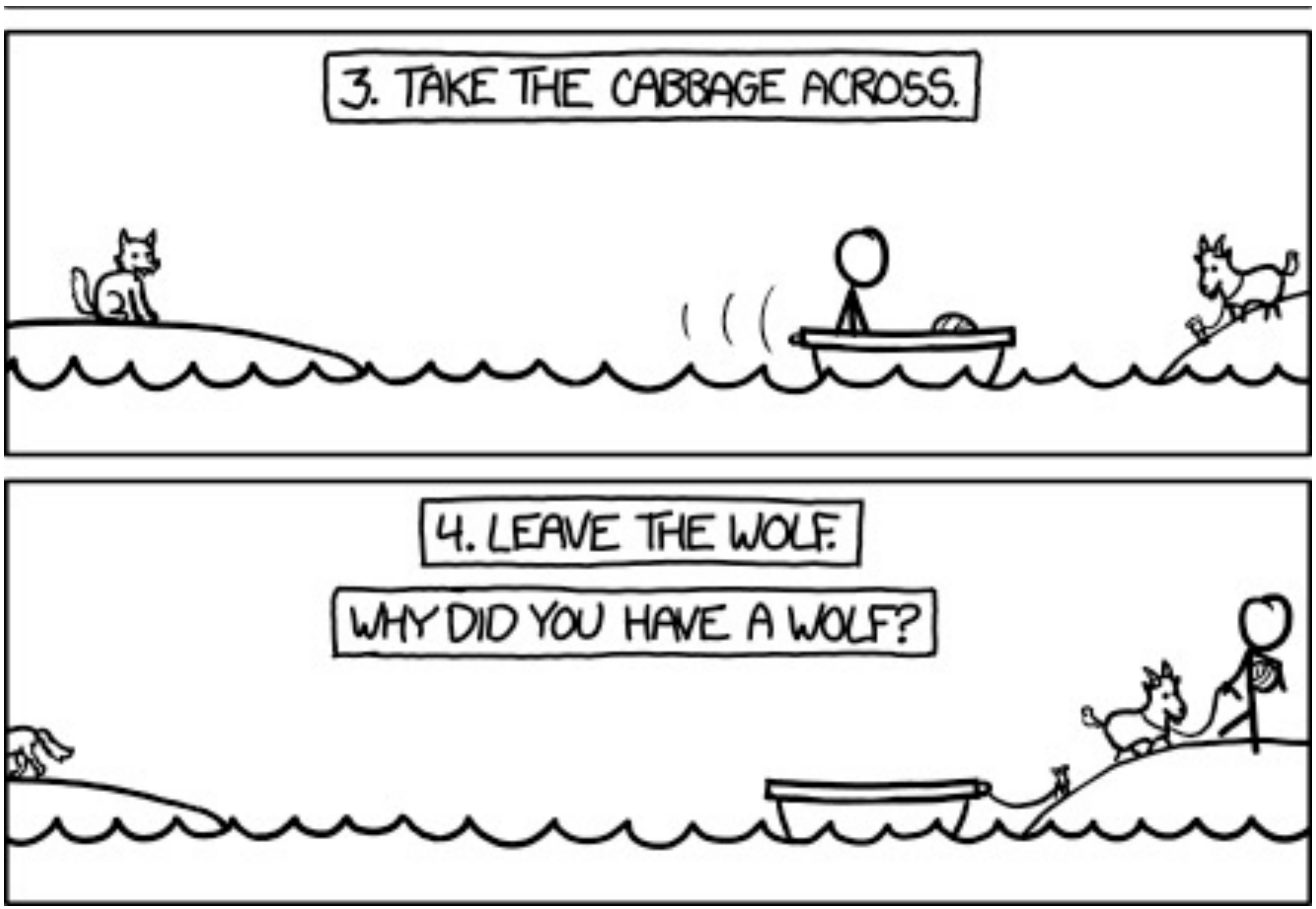
Many AI problems can be formulated as search.



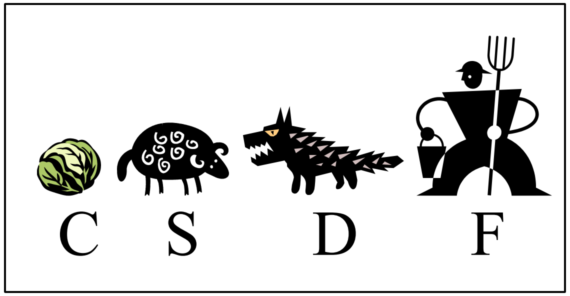


http://xkcd.com/1134/

http://xkcd.com/1134/



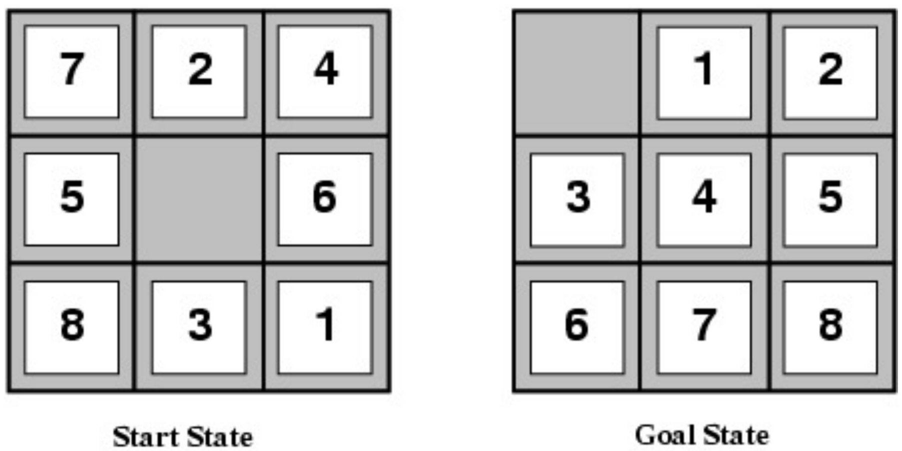
The search problem



- State space $\boldsymbol{S}$: all valid configurations
- Initial state $\boldsymbol{I}$={(CSDF,)} $\subseteq \boldsymbol{S}$
- Goal state $\boldsymbol{G}$={(,CSDF)} $\subseteq \boldsymbol{S}$
- Successor function ***succs***(s)$\subseteq \boldsymbol{S}$: 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 $\boldsymbol{I}$ to a state in $\boldsymbol{G}$.
  - Optionally minimize the cost of the solution.

Search examples

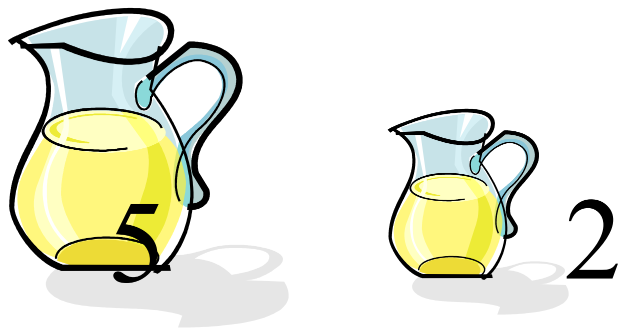
- 8-puzzle



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

Search examples

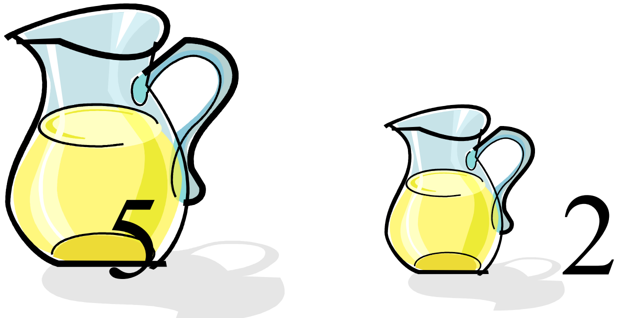
- Water jugs: how to get 1?



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

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

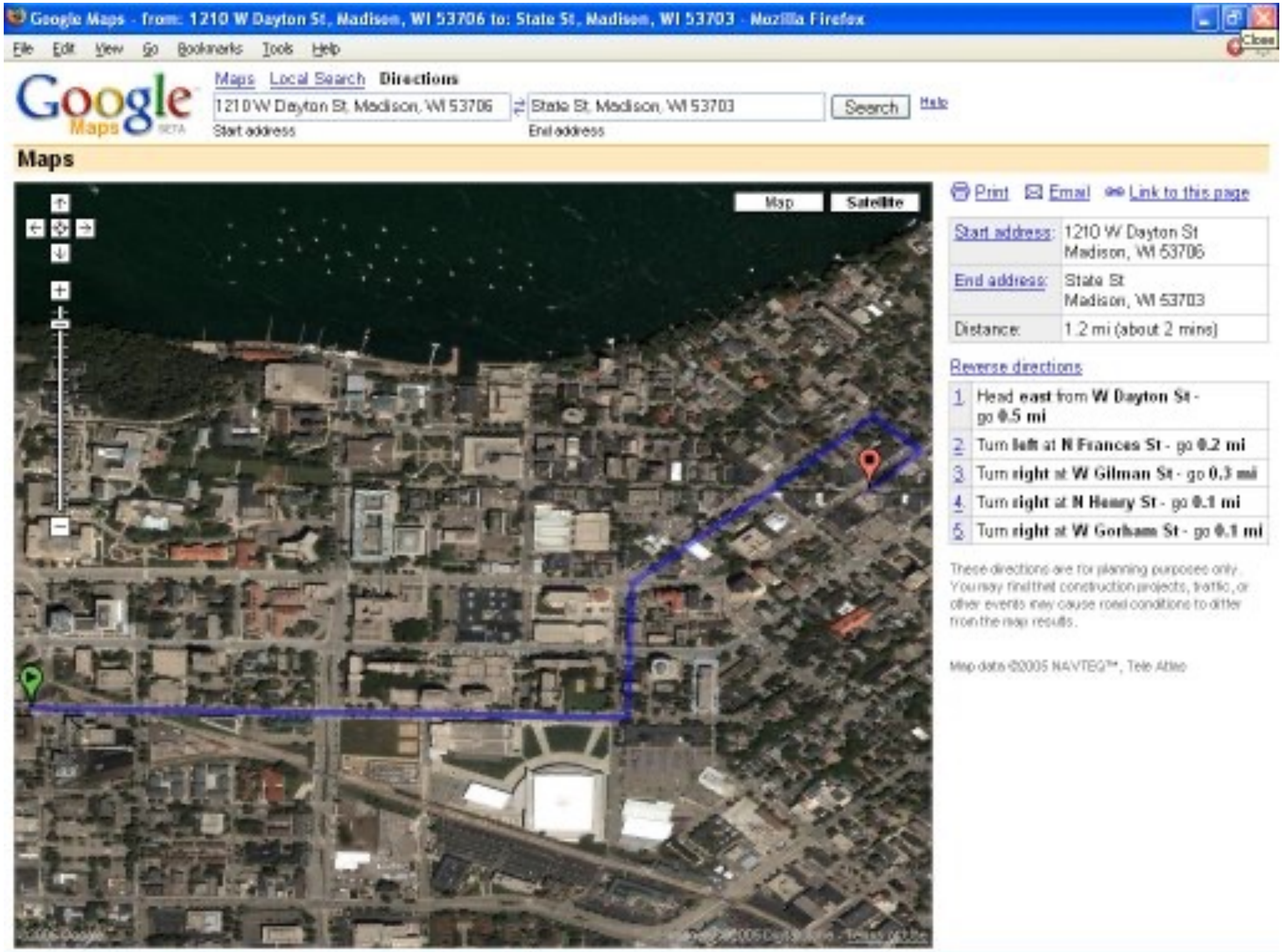
(x,0) and x>=2 -> (x-2,2) ; pour 5-gal into 2-gal

(1,0) -> (0,1) ; empty 5-gal into 2-gal

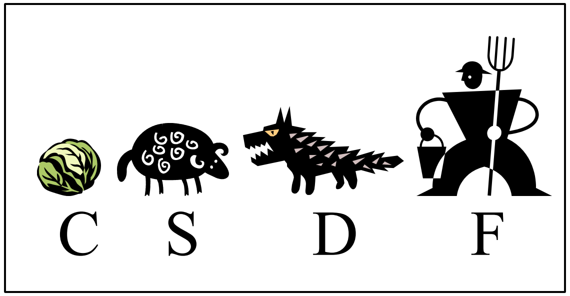
Search examples

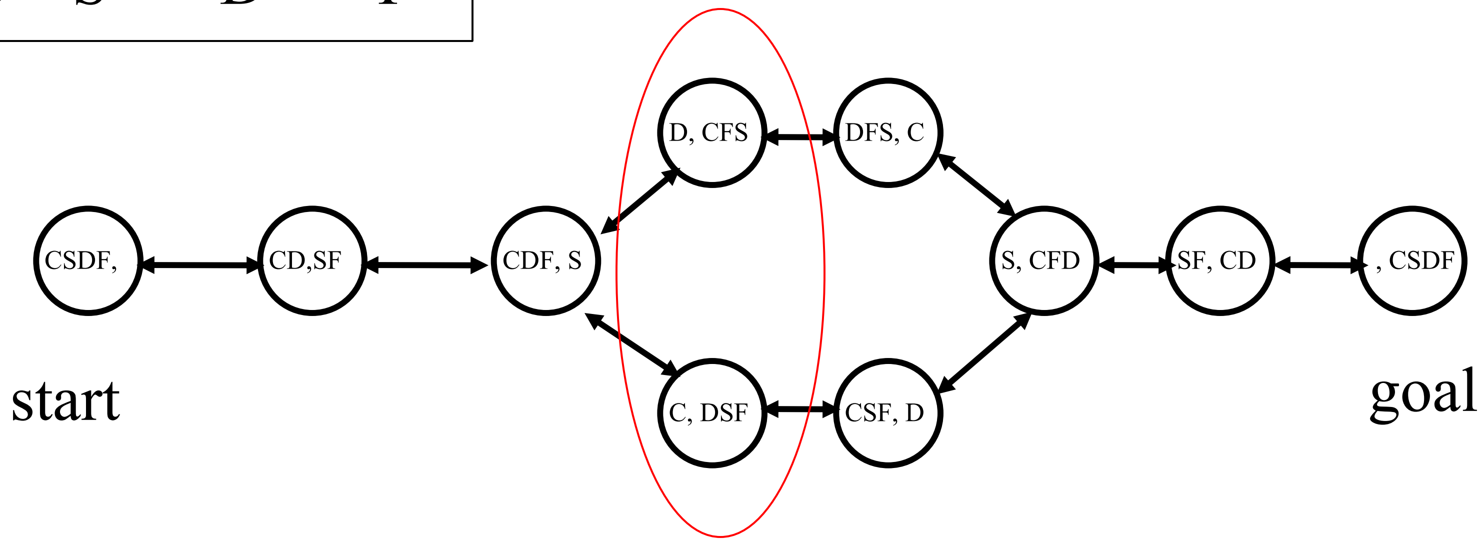
Search examples

- Route finding (State? Successors? Cost weighted)



A directed graph in state space

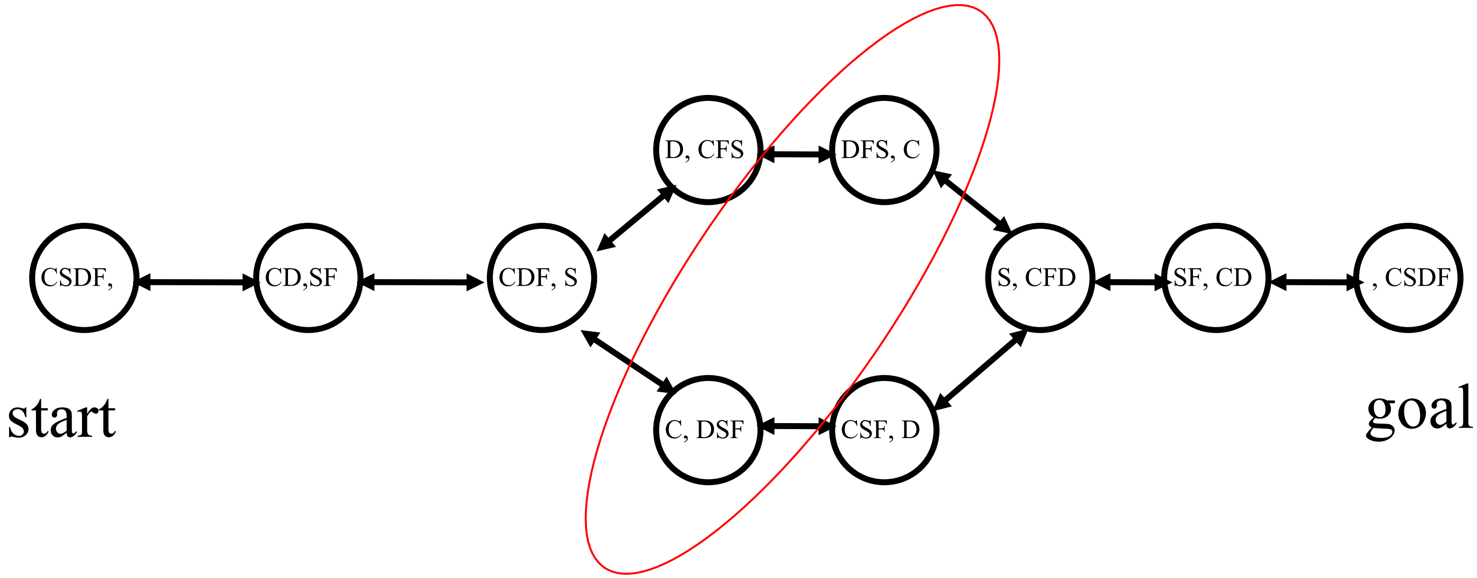




- In general there will be many generated, but un-expanded states at any given time
- One has to choose which one to expand next

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?



Uninformed search on trees

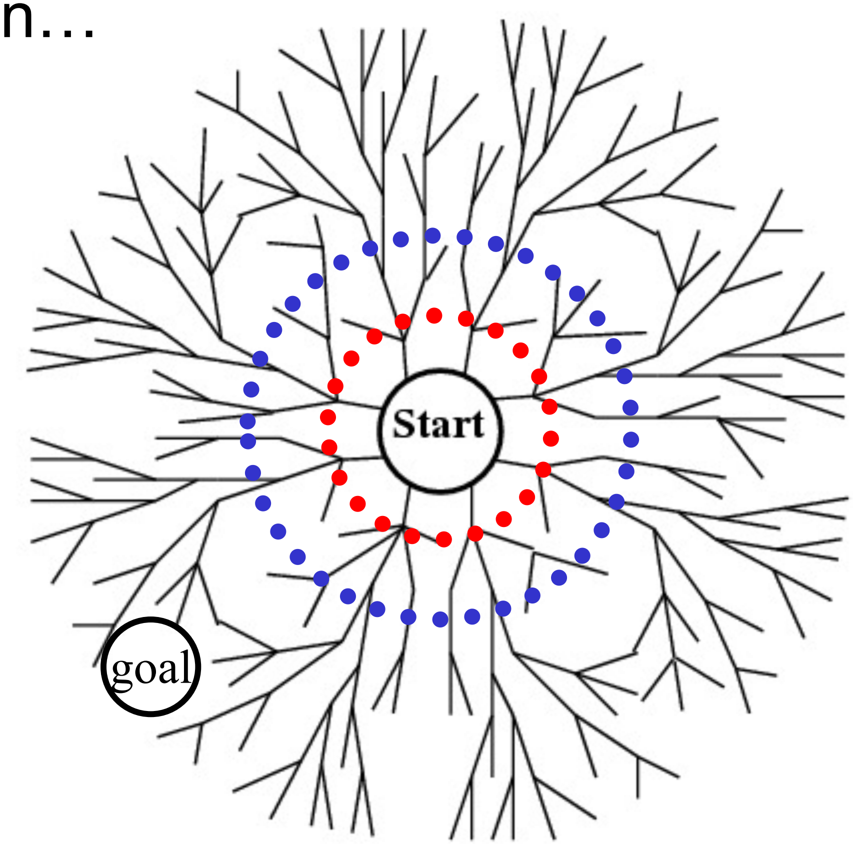
- Uninformed means we only know:
  - The goal test
  - The ***succs***() function
- But not which non-goal states are better: that would be informed search (next topic).
- For now, we also assume ***succs***() graph is a tree.
  - Won't encounter repeated states.
  - We will relax it later.
- Search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

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…

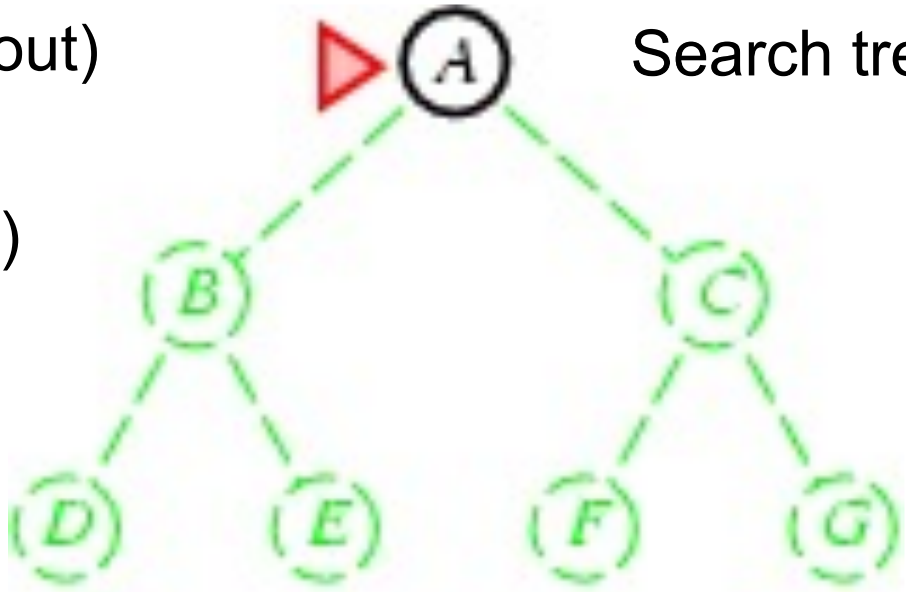
ripple



Breadth-first search (BFS)

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

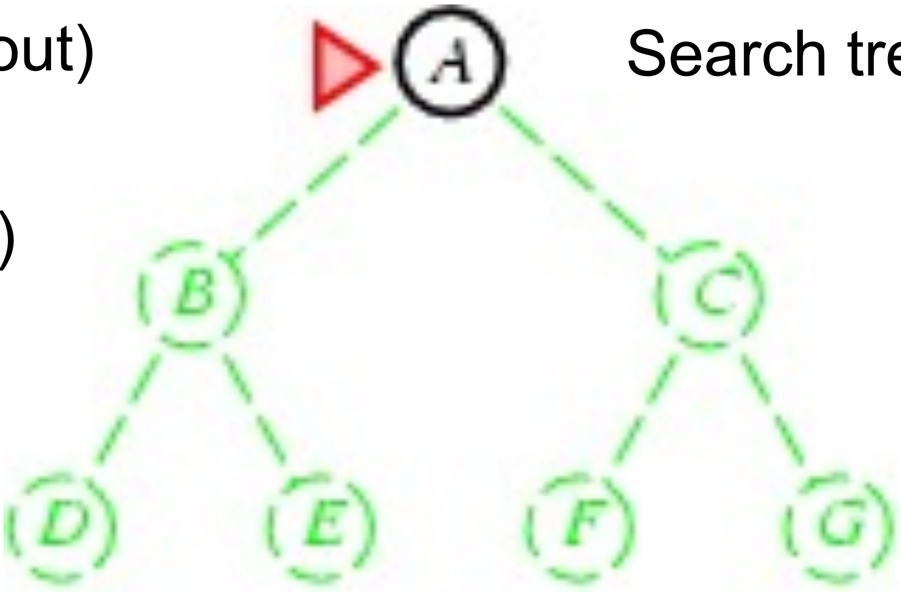
Initial state: **A**

Goal state: **G**

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] →

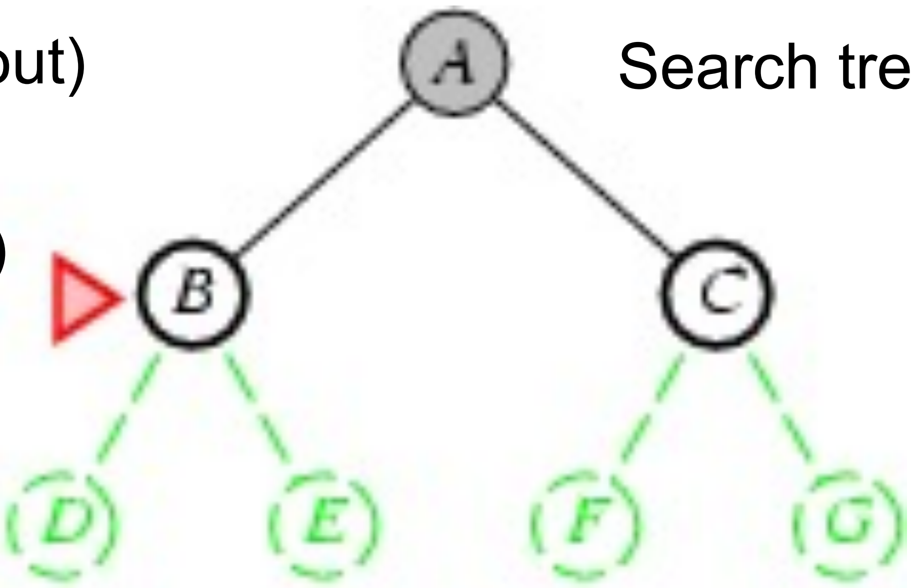
Initial state: **A**
Goal state: **G**

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



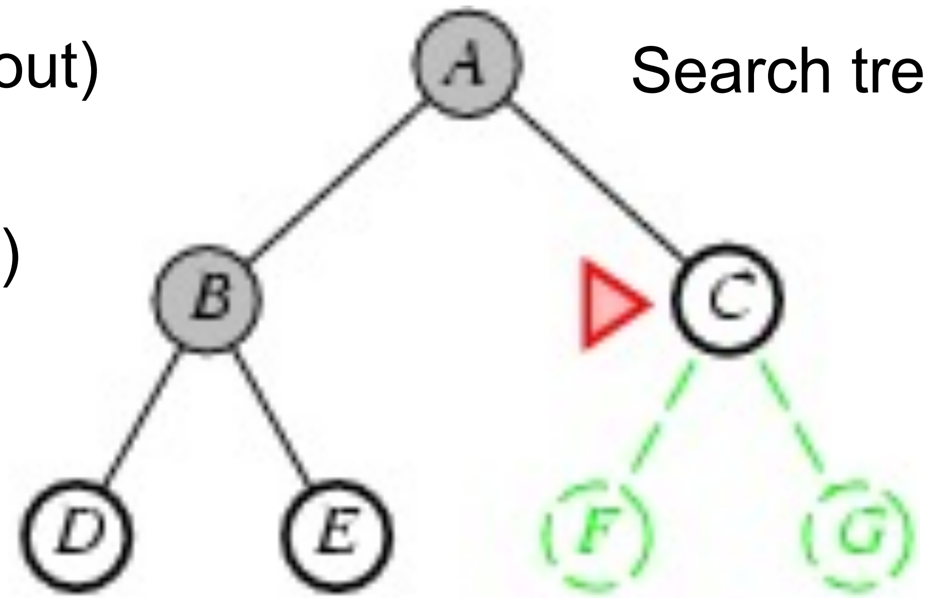
queue (fringe, OPEN)
→ [CB] → A

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

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

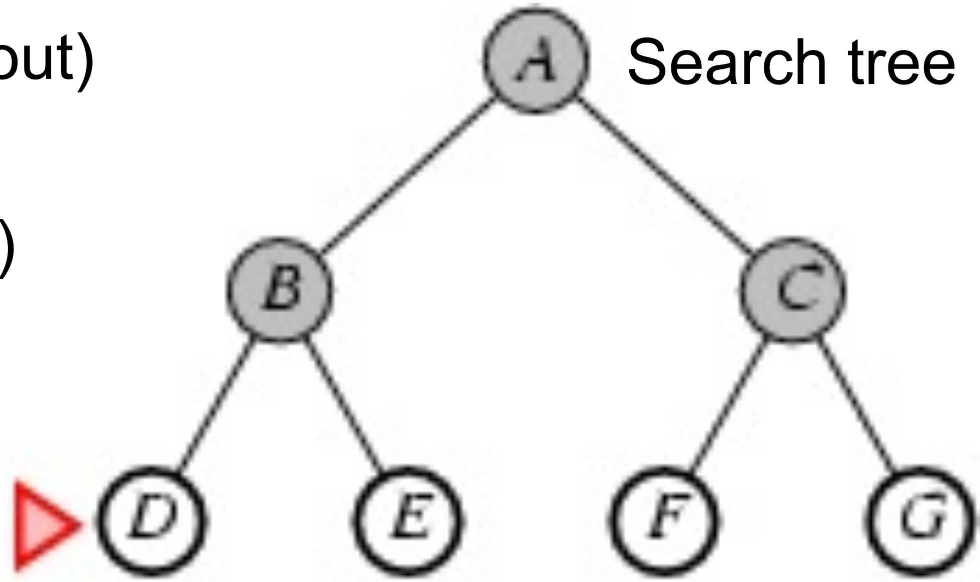
queue (fringe, OPEN)
→ [EDC] → B

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

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



queue (fringe, OPEN)
→[GFED] → C

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

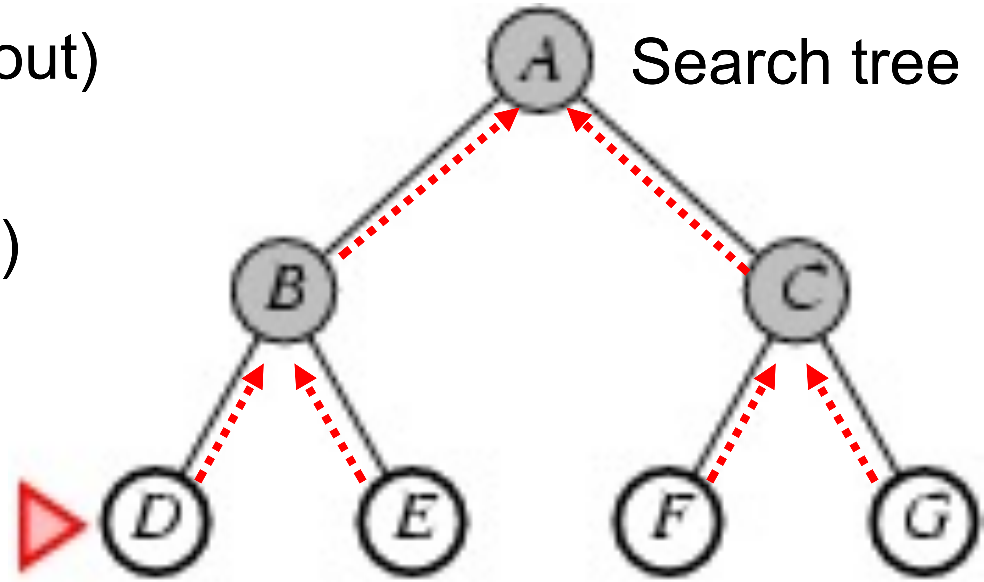
If G is a goal, we've seen it, but we don't stop!

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

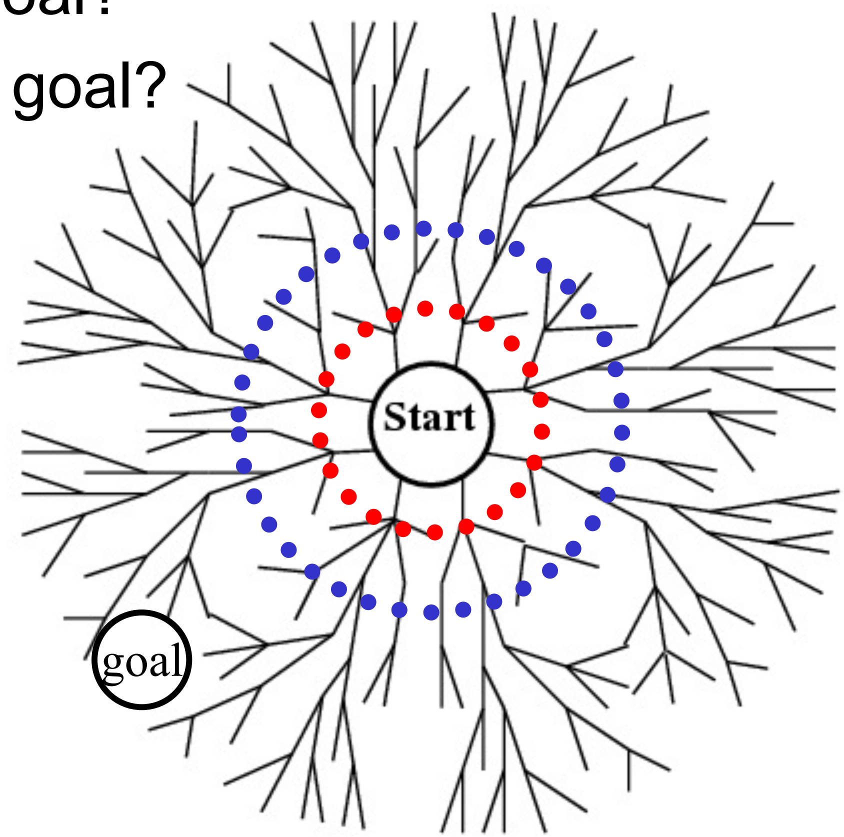
... until much later we pop G.

We need back pointers to recover the solution path.

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

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



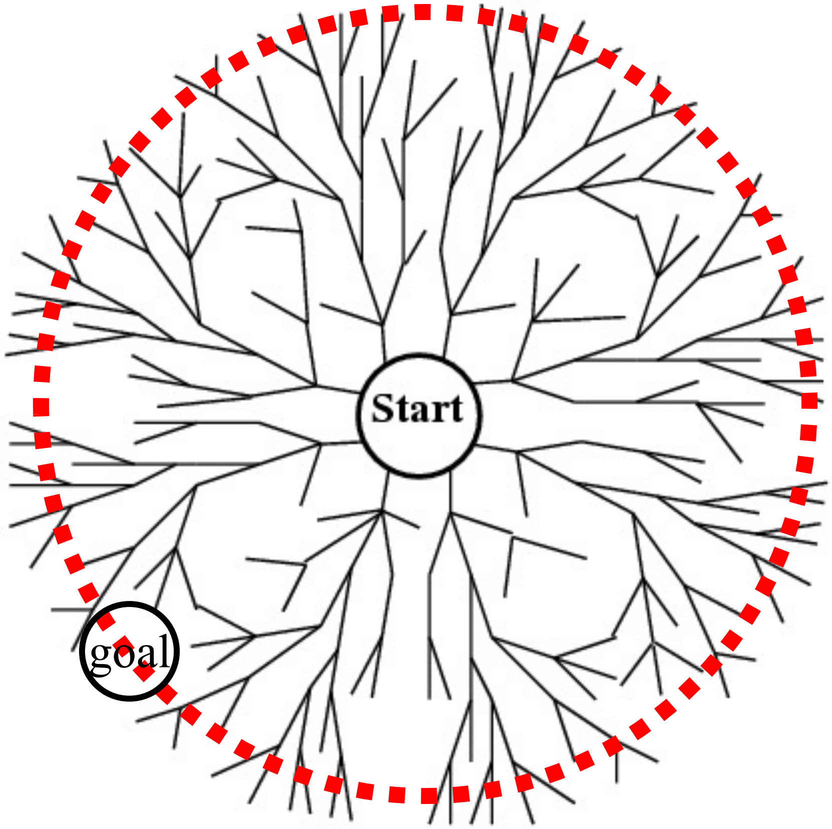
Performance of BFS

Four measures of search algorithms:

- Completeness (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 $O(b^d)$)

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

- Convince yourself this is ***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

Four measures of search algorithms:

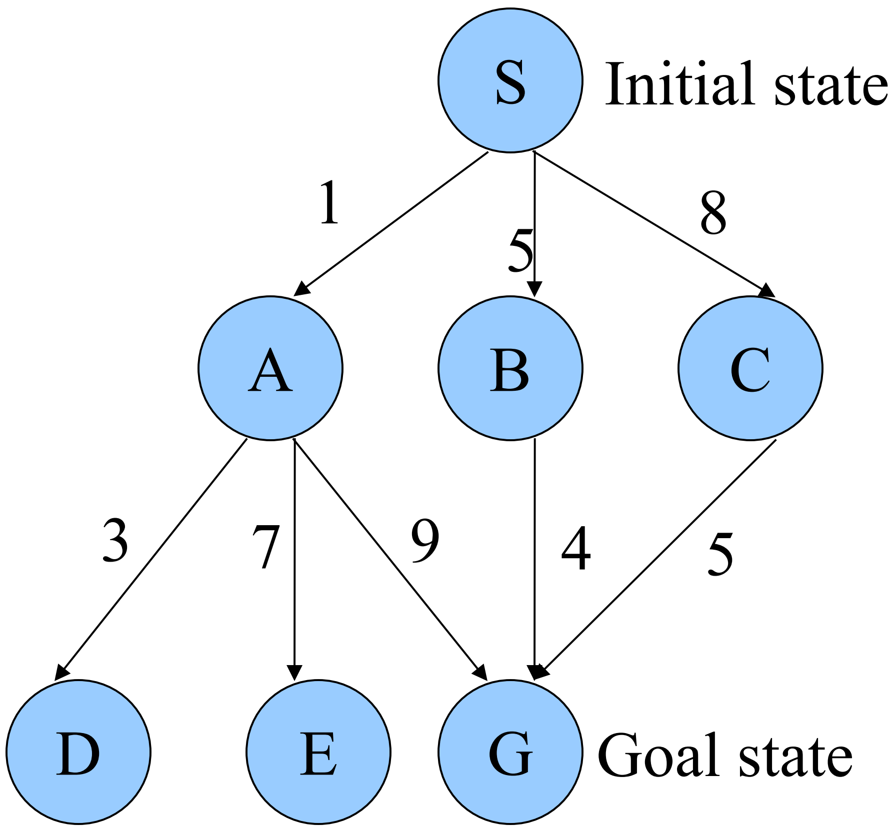
Solution: Uniform-cost search

- 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, Figure 3.11)
  - 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

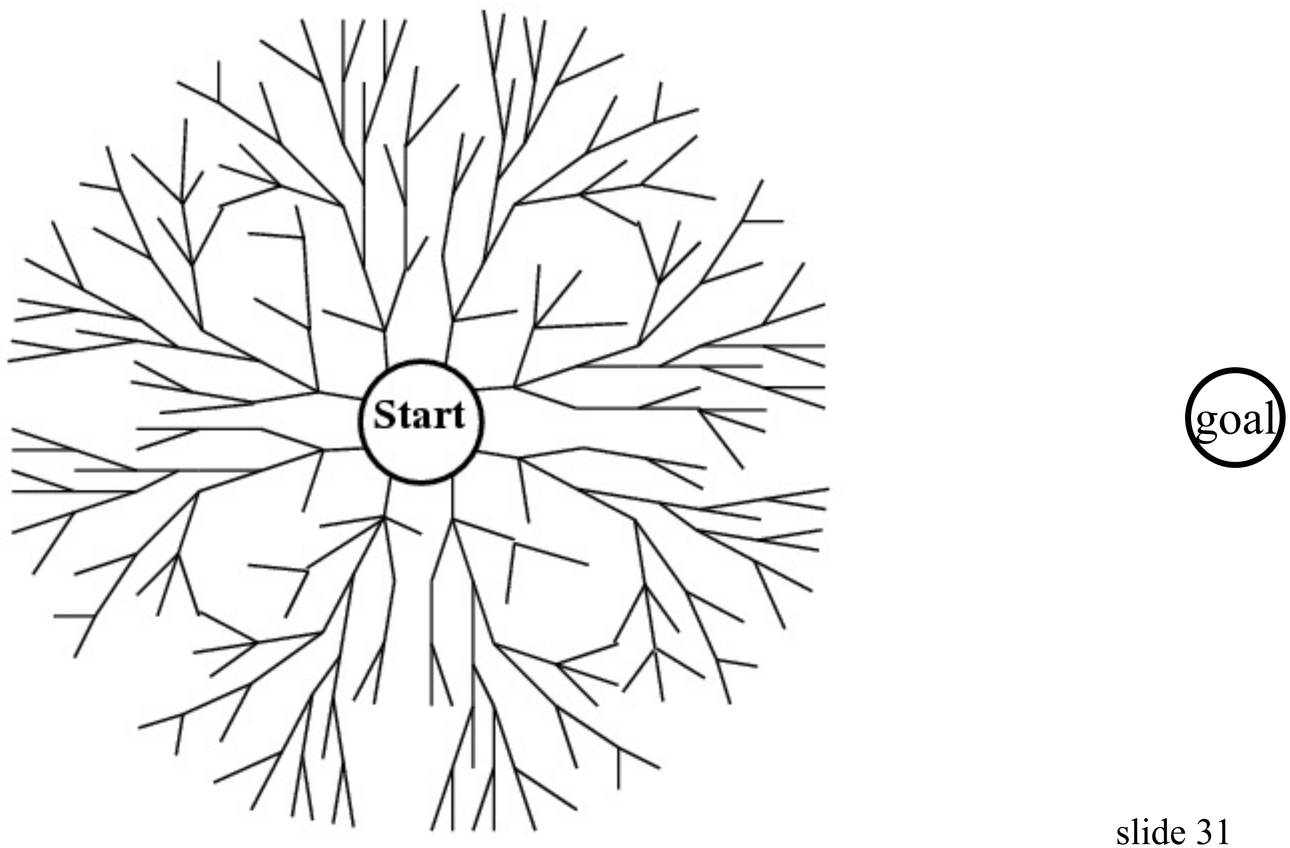
Example



(All edges are directed, pointing downwards)

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
  - $O(b^{C^*/\varepsilon})$



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.

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
```

Recall the bad space complexity of BFS

Solution: Uniform-cost search

Four measures of search algorithms:

- Completeness (not finding all goals): find a goal.
- Optimality: yes if edges cost 1 (more gene ally positive non-decreasing with depth), no otherwise.
- Time comple): goal is the last node at radius d.
  - Have to g es at radius d.
  - $b + b^2 + \ldots + b^d \sim O(b^d)$

Solution: Depth-first search

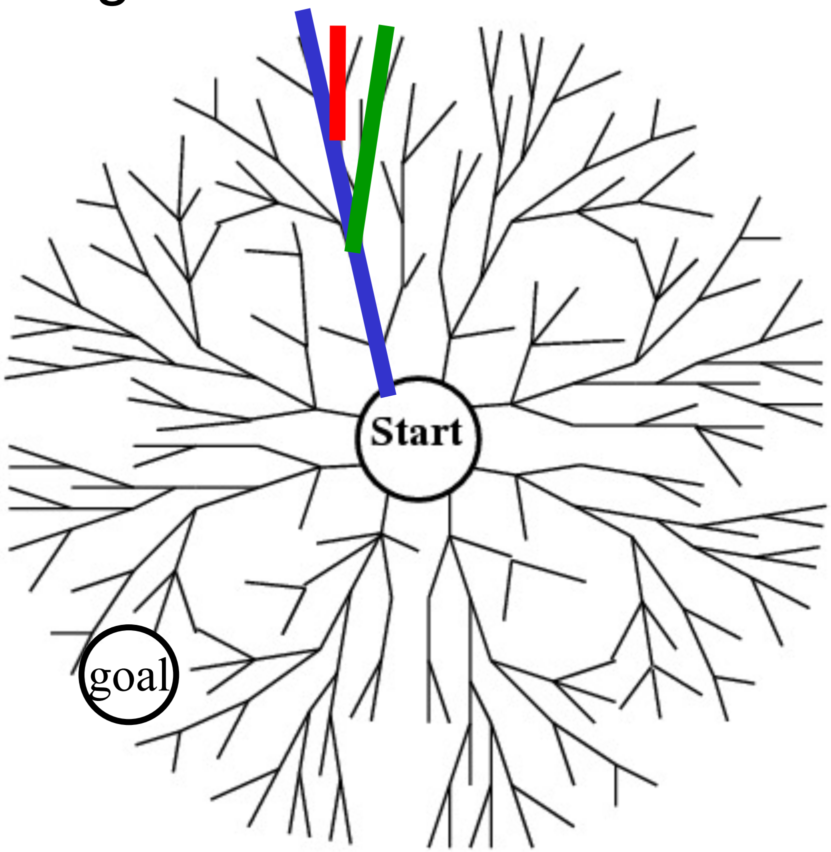
- Space complexity (bad, Figure 3.11)
  - Back points for all generated nodes $O(b^d)$
  - The queue (smaller, but still $O(b^d)$)

Depth-first search

Expand the deepest node first

1. Select a direction, go deep to the end
2. Slightly change the end
3. Slightly change the end some more…

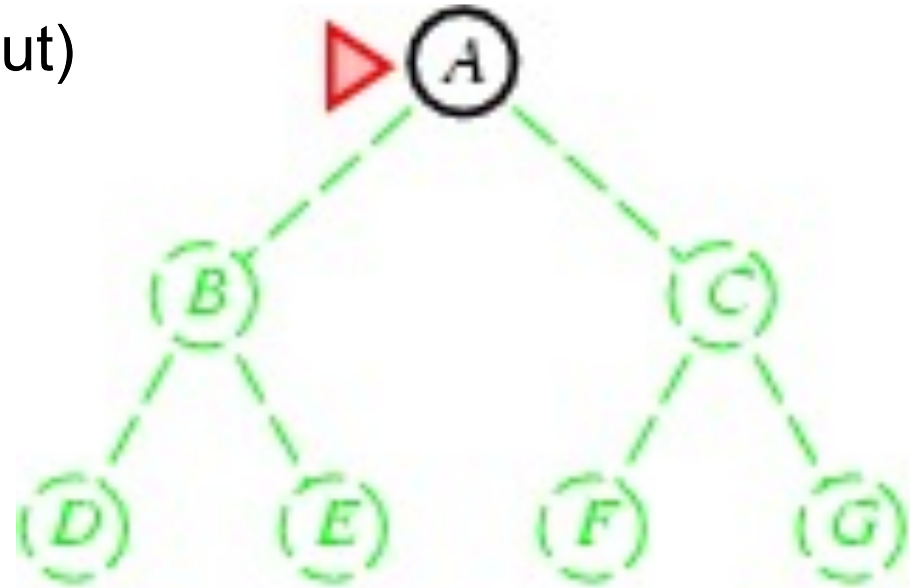
fan



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



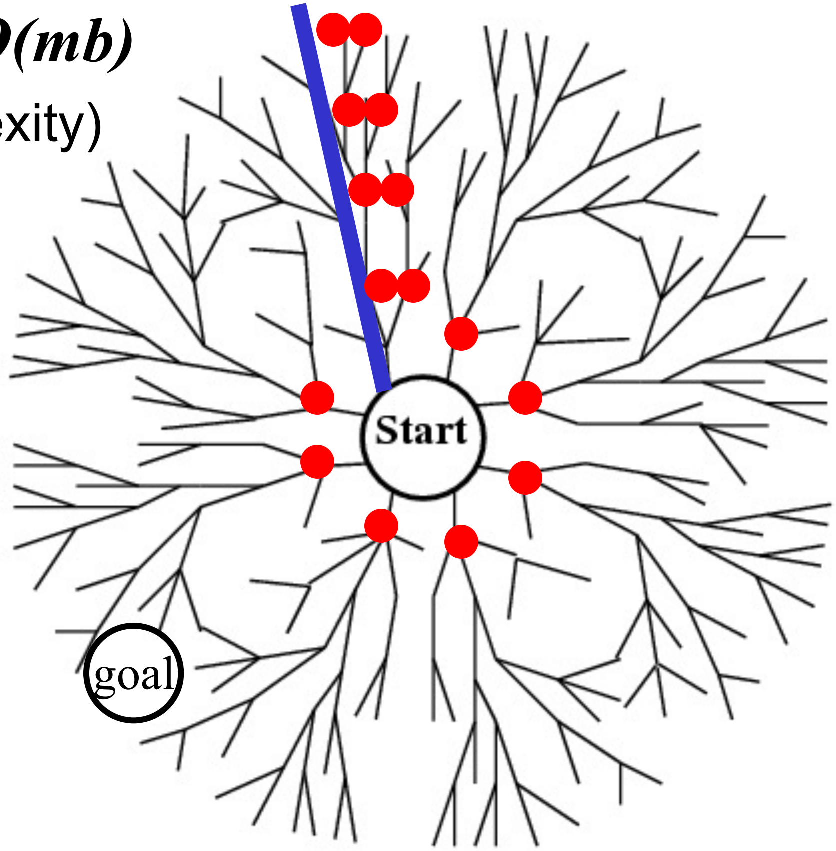
stack (fringe)

[] ⇔

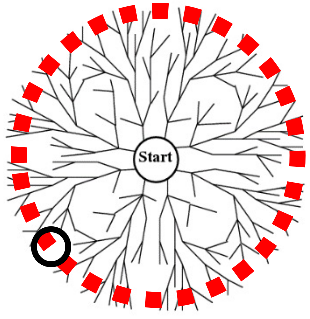
What's in the fringe for DFS?

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

(Space complexity)



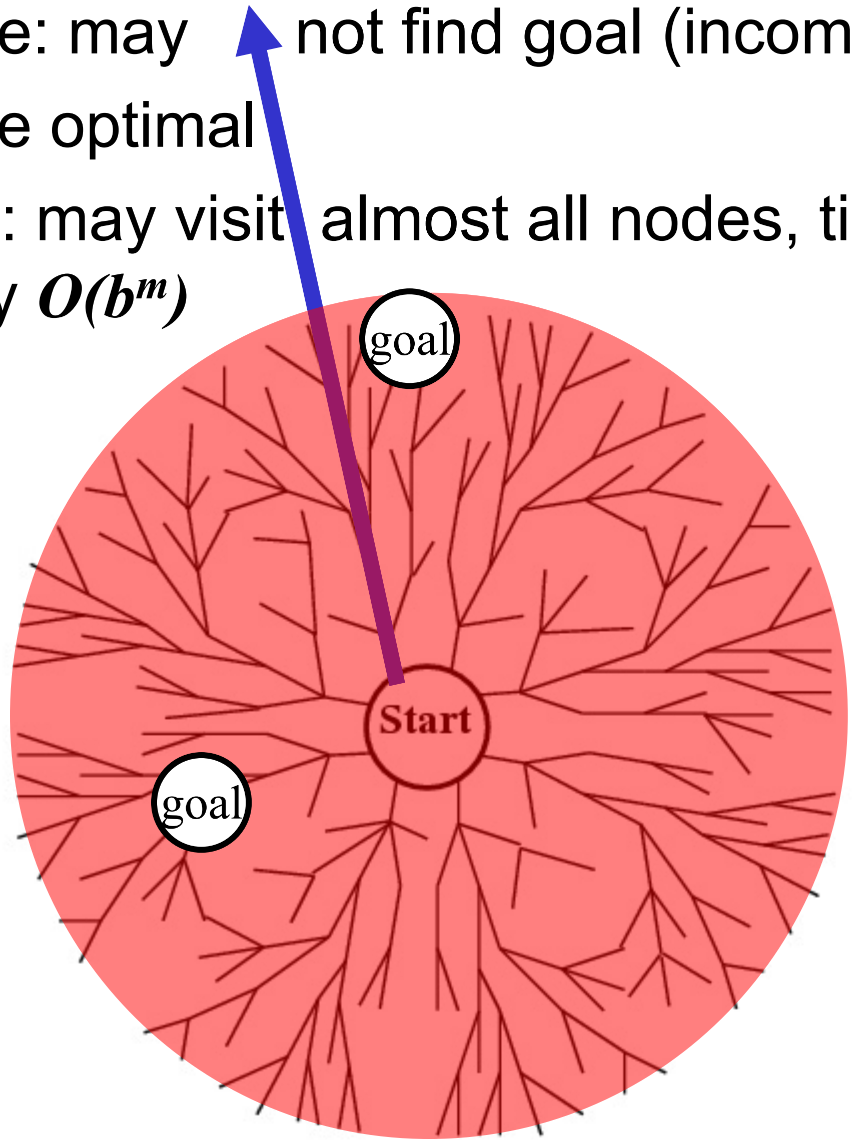
c.f. BFS $O(b^d)$



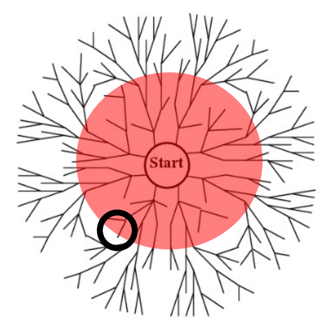
- "backtracking search" even less space
  - generate siblings (if applicable)

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

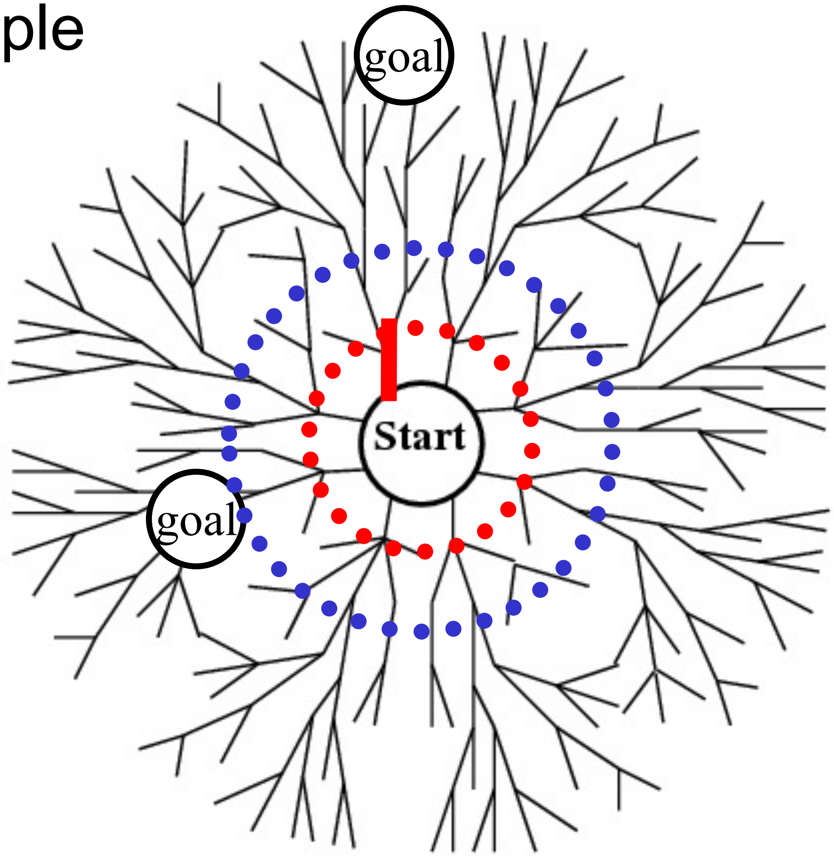
	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$

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

How about this?

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

fan within ripple



Iterative deepening

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

Performance of search algorithms on trees

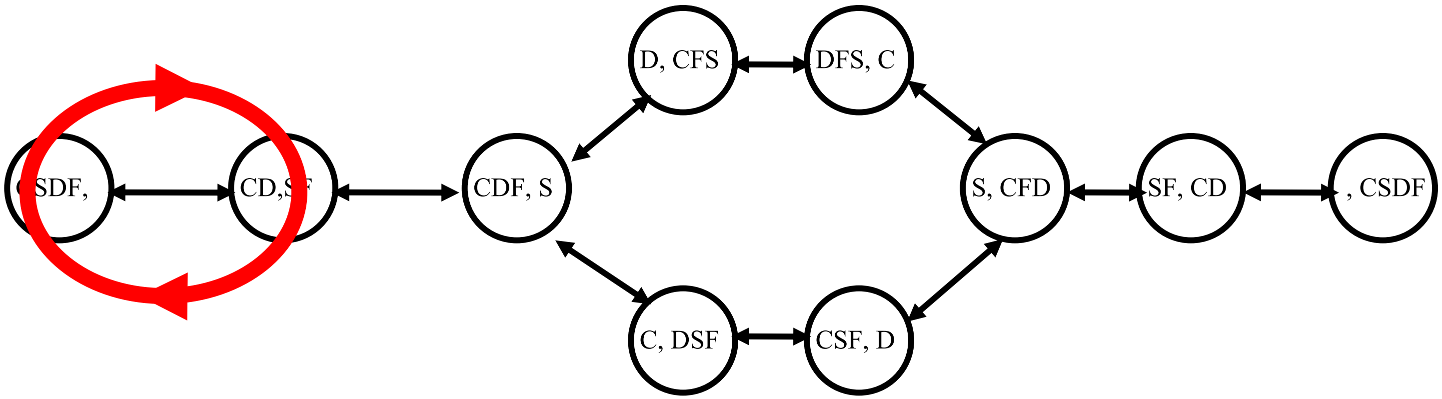
b: branching factor (assume finite) d: goal depth m: graph depth

	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.

If state space graph is not a tree

- The problem: repeated states

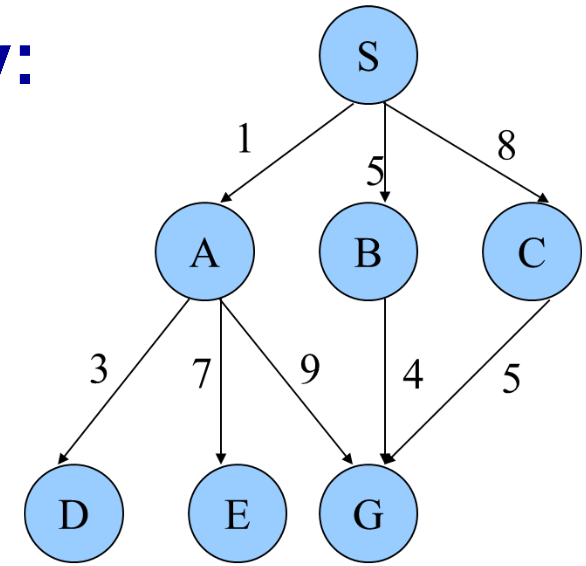


- Ignore the danger of repeated states: wasteful (BFS) or impossible (DFS). Can you see why?
- How to prevent it?

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.

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 (This is the only uninformed search that worries about costs.)

- Depth-First Search: S A D E G

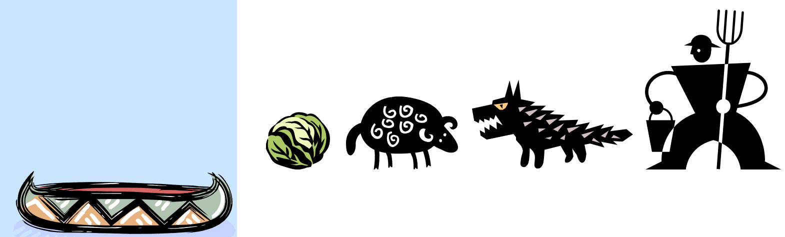
  Solution found: S A G

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

  Solution found: S A G

What you should know

- Problem solving as search: state, successors, goal test
- Uninformed search
  - Breadth-first search
    - Uniform-cost search
  - Depth-first search
  - Iterative deepening



- Can you unify them using the same algorithm, with different priority functions?
- Performance measures
  - Completeness, optimality, time complexity, space complexity