Lecture 2: Problem as Search in a Graph

CSSE 5600/6600: Artificial Intelligence

Instructor: Bo Liu

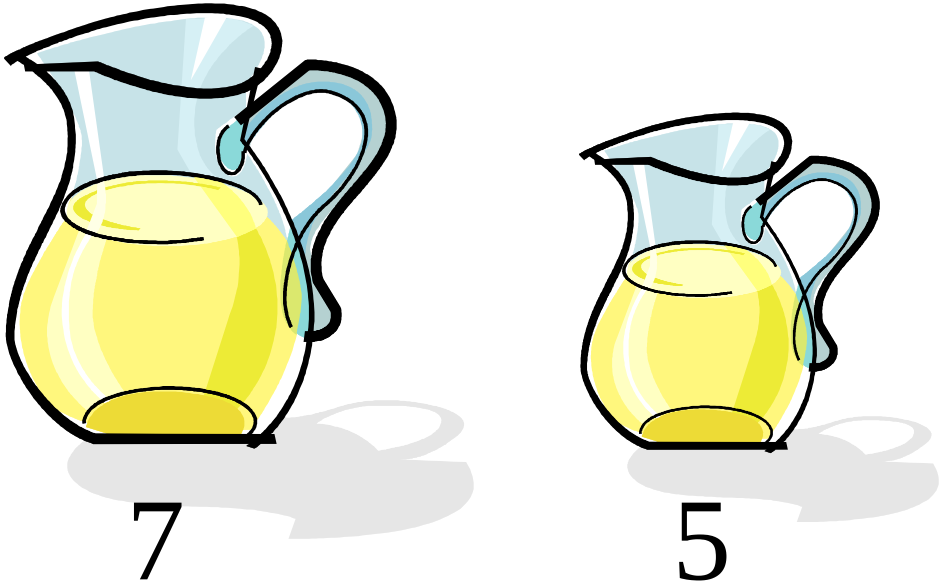
Thanks for Jerry Zhu's slides

Content

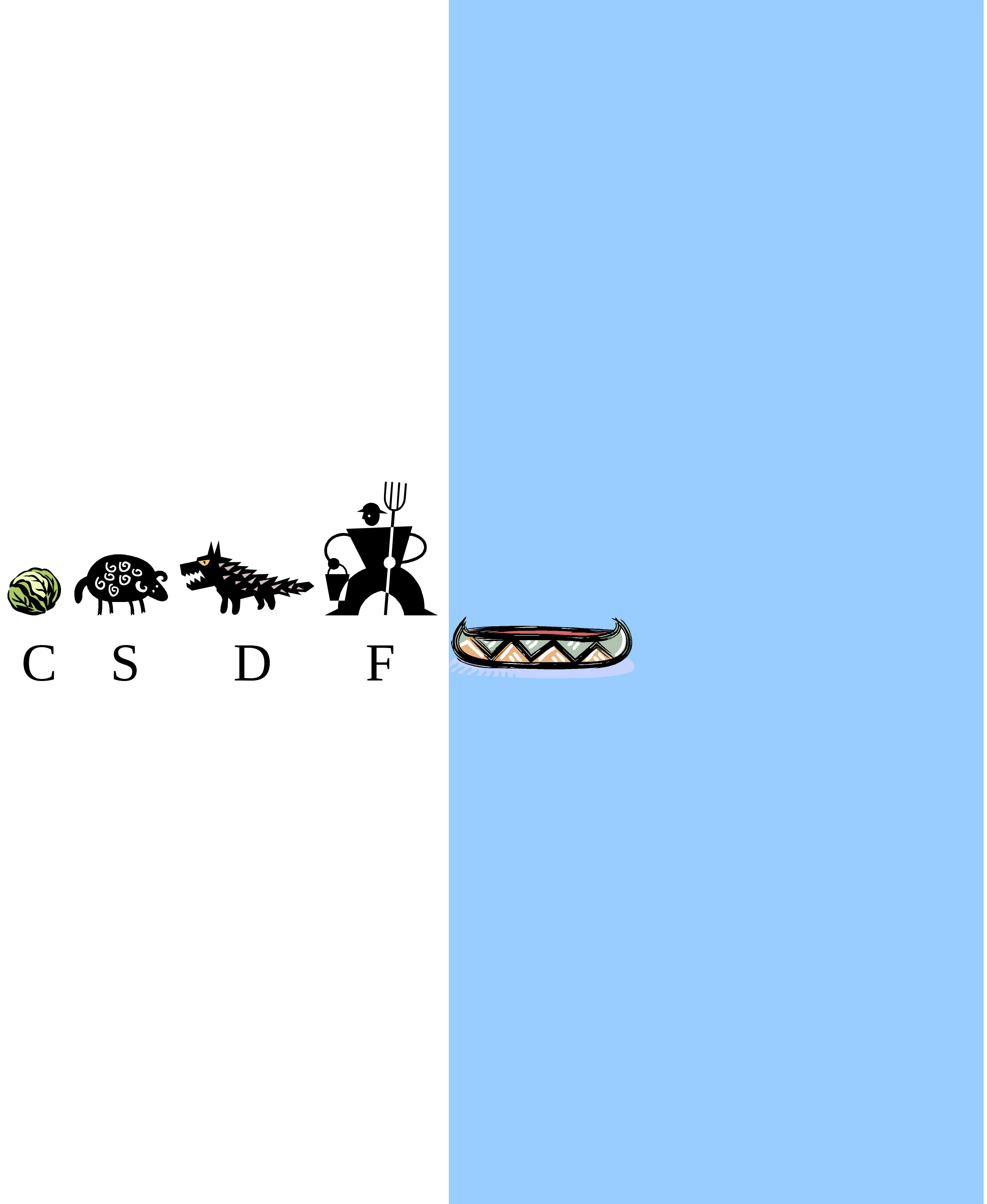
- ❖Why is search the key problem-solving technique is AI?
- ❖Problem types
- ❖Problem formulation
- ❖Understanding and comparing several “blind” search algorithms.

Search examples

- Water jugs: how to get 1?



- Two operations: fill up (from tap or other jug), empty (to ground or other jug)



Rules:

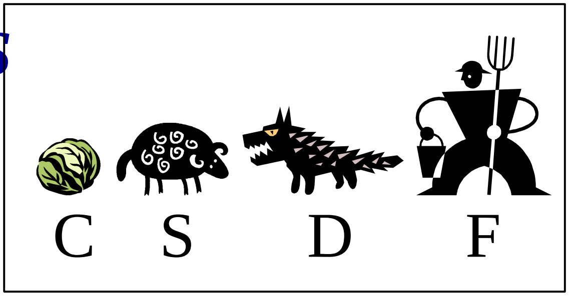
(1) boat can take at most 2 objects one time

(2) F kills D, D kills S, S kills C, and C kills F, if no others around.

Goal: Everybody go to the other side of this river

The search problem

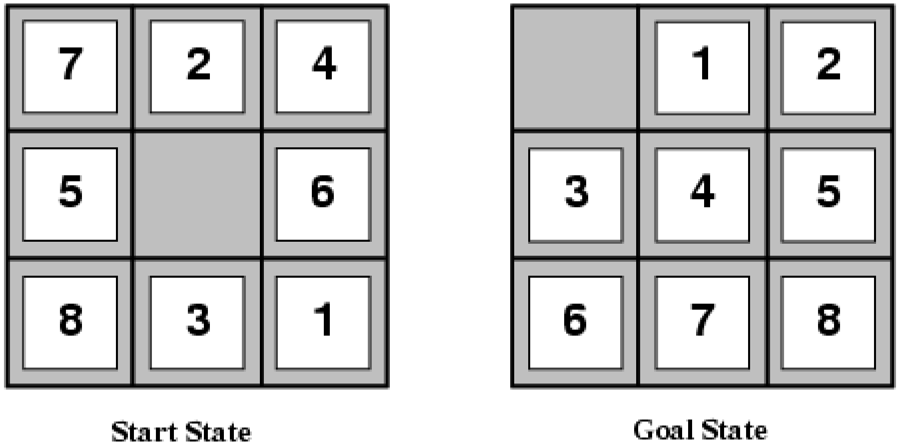
- State space $\boldsymbol{S}$: all valid configurations
- Initial states (nodes) $\boldsymbol{I}$={(CSDF,)} $\subseteq \boldsymbol{S}$
  - Where’s the boat?
- Goal states $\boldsymbol{G}$={(,CSDF)} $\subseteq \boldsymbol{S}$



- Successor function ***succs***(s)$\subseteq \boldsymbol{S}$: states reachable in one step (one arc) from s
  - ***succs***((CSDF,)) = {(CD, SF)}
  - ***succs***((CDF,S)) = {(CD,FS), (D,CFS), (C, DFS)}
- Cost(s,s’)=1 for all arcs. (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

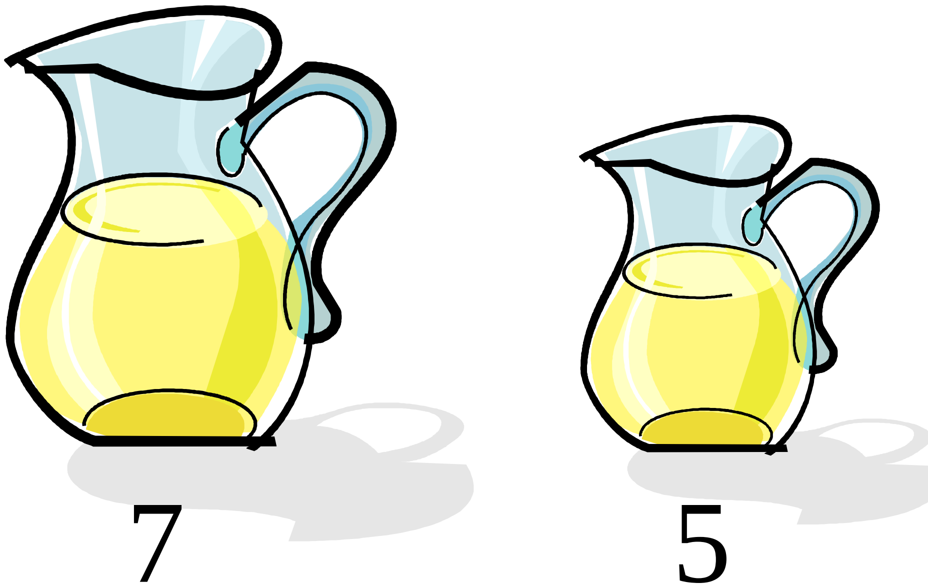
- 8-puzzle



- States = configurations
- successor function = up to 4 kinds of movement
- Cost = 1 for each move

Search examples

- Water jugs: how to get 1?



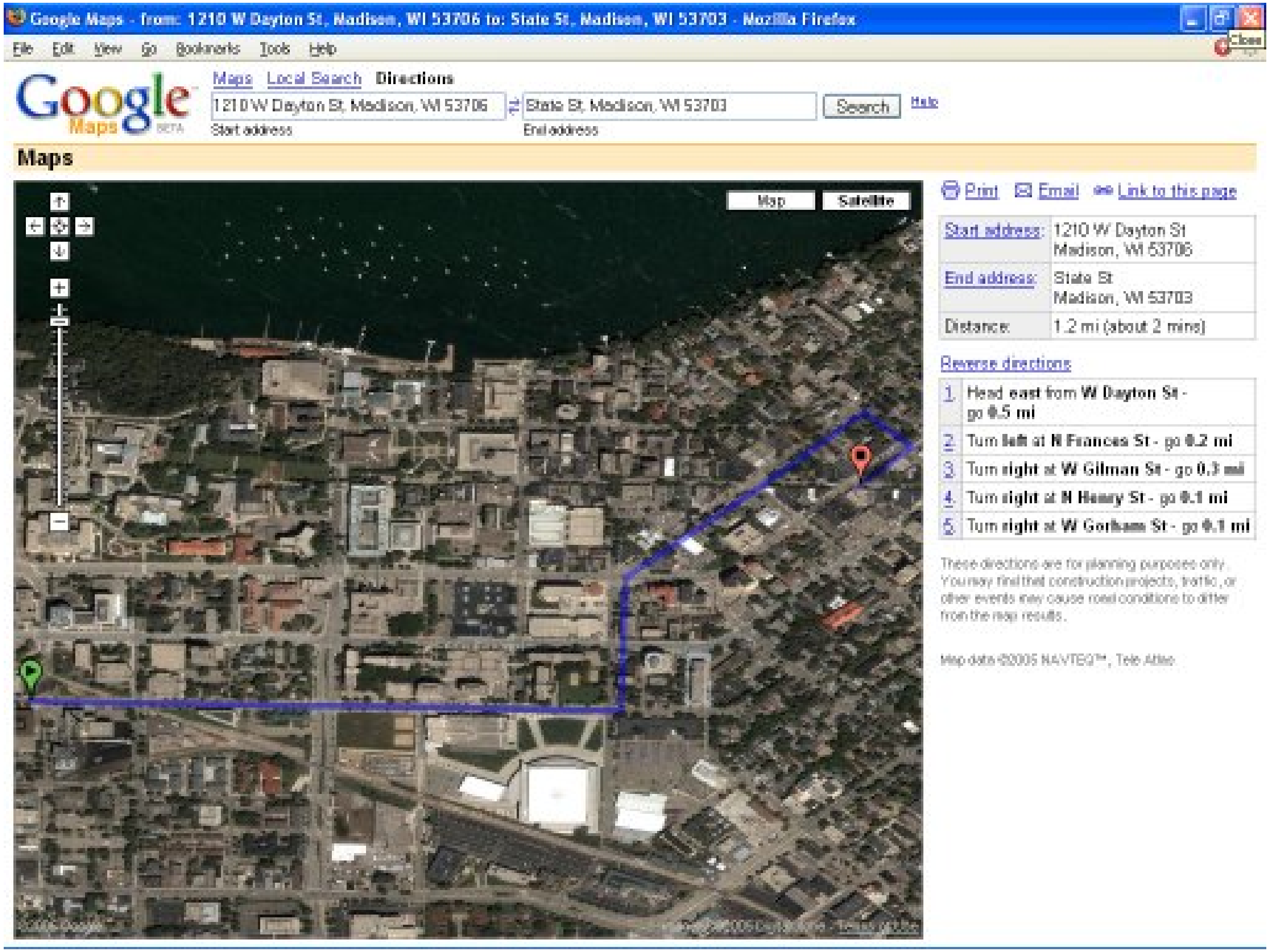
- Goal? (How many goal states? (0,0), (0,1),...,(7,5))
- Successor functions: fill up (from tap or other jug), empty (to ground or other jug) – fill big jug, fill small jug, empty big jug, empty small jug, fill big jug from small jug, fill small jug from small jug

One solution

- (7, 0) → (2, 5) → (2, 0) → (2, 0) → (7, 2) → (4, 5) → (4, 0) → (0, 4) → (7, 4) → (6, 5) → (6, 0) → (1,5) → (1, 0)
- Other solutions?

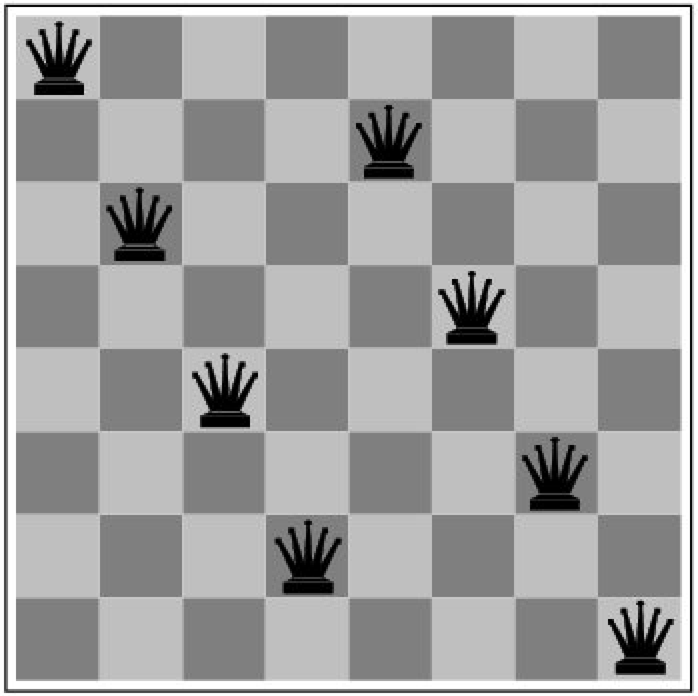
Search examples

- Route finding (state? Successors? Cost weighted)



8-queens

- How to define states?



- states? -any arrangement of n<=8 queens

  -*or* arrangements of n<=8 queens in leftmost n columns, 1 per column, such that no queen attacks any other.

- initial state? no queens on the board

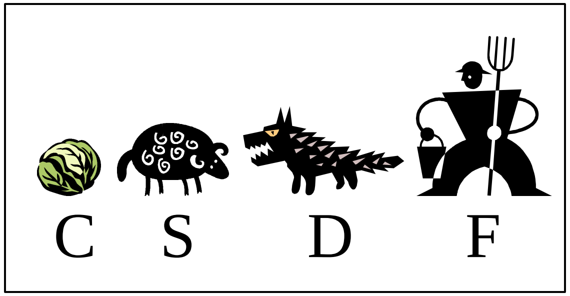
- Successor functions? -add queen to any empty square

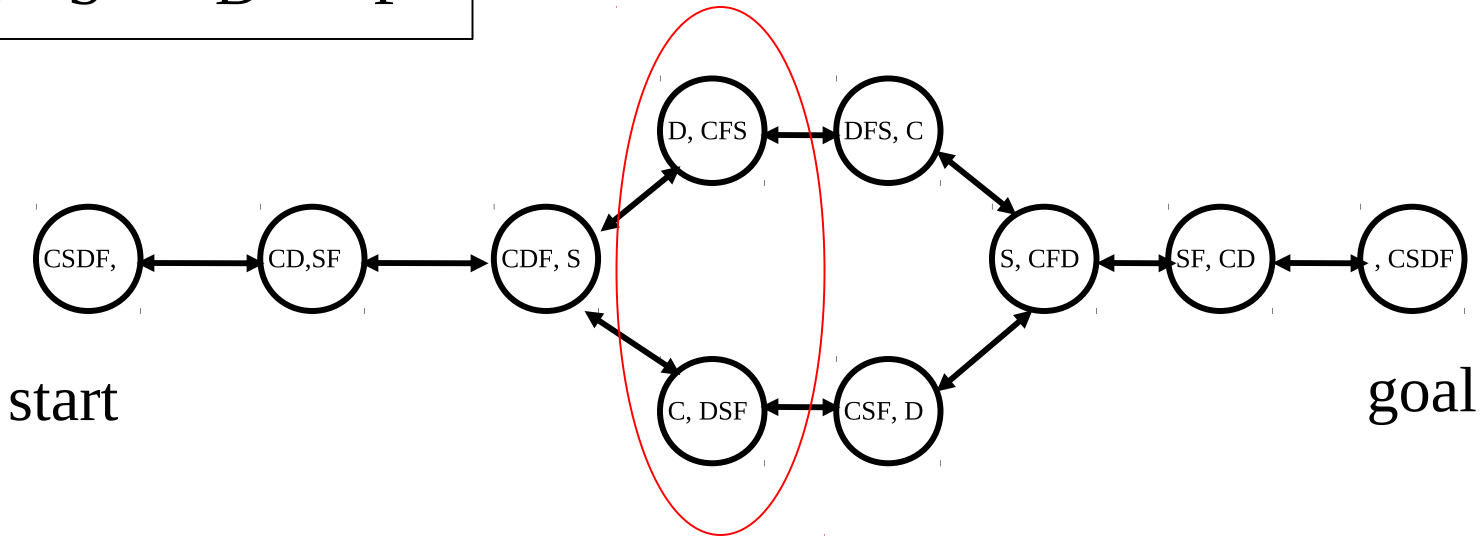
  -*or* add queen to leftmost empty square such that it is not attacked by other queens.

- goal test? 8 queens on the board, none attacked.

- path cost? 1 per move

A directed graph in state space

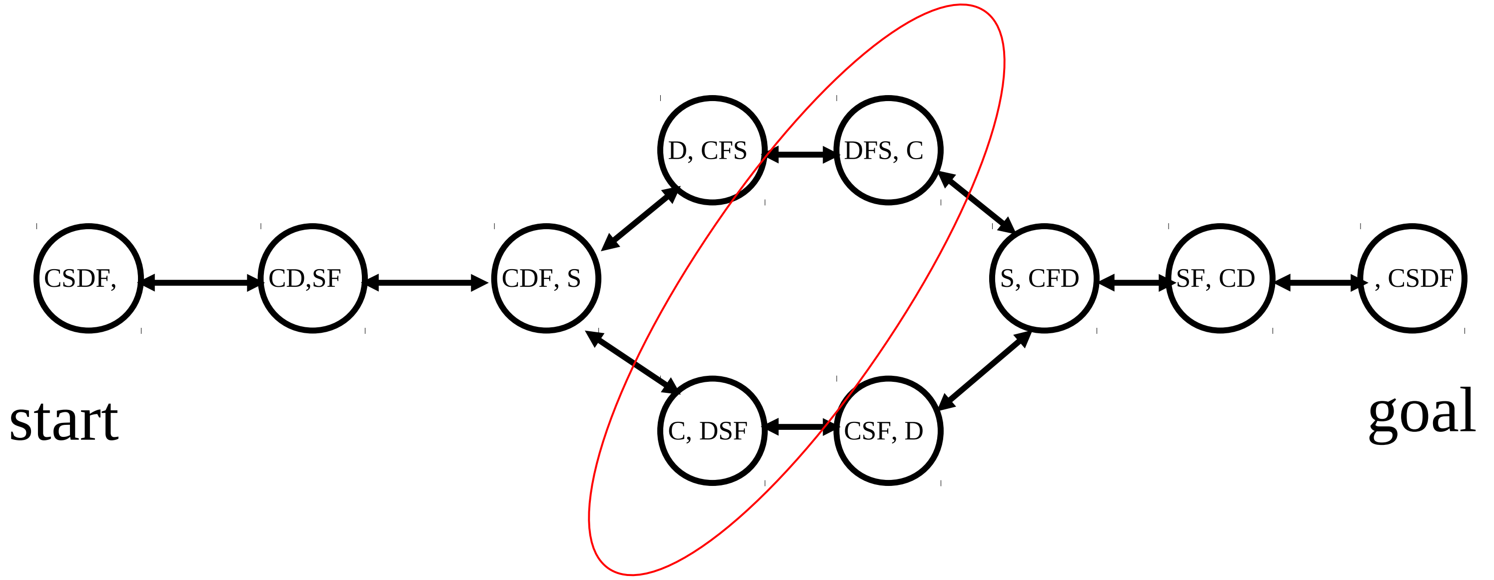




- In general there will be many generated, but un-expanded states at any given time
- One has to choose which one to expand next

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

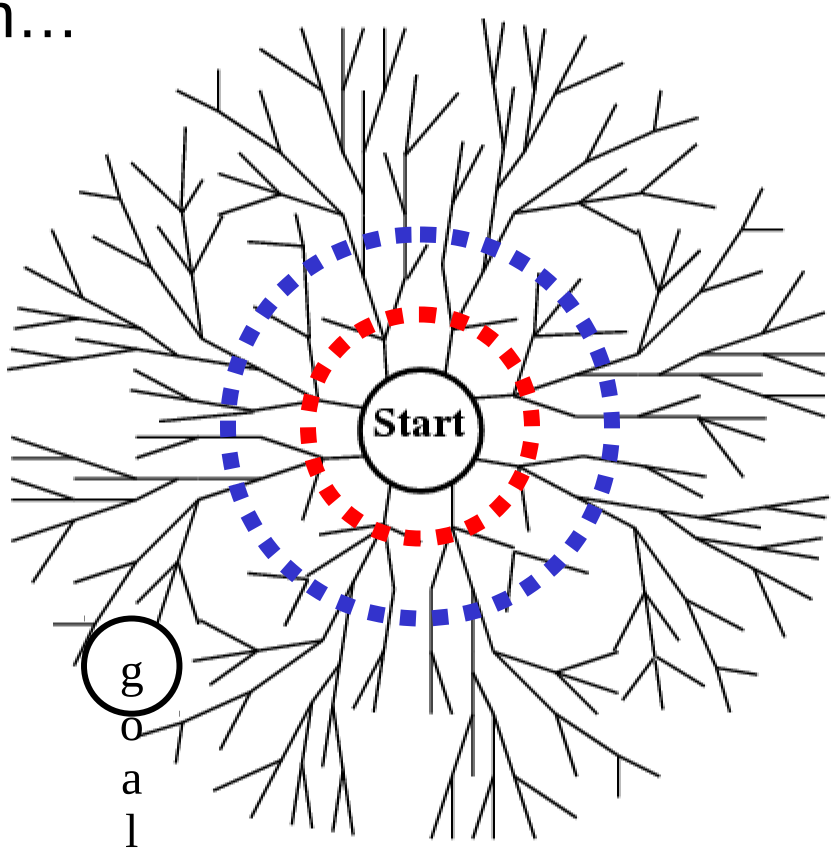
- Uninformed means we only know:
  - The goal test
  - The ***succs***() function
- But not which non-goal states are better: that would be informed search (next lecture).
- For now, we also assume ***succs***() graph is a tree.
  - Won't encounter repeated states.
  - We will relax it later.
- Search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

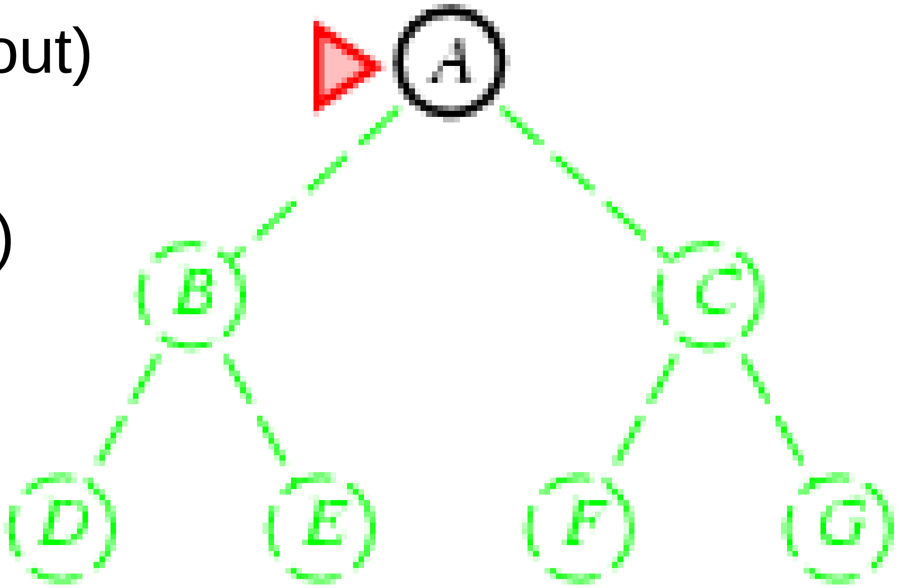
ripple



Breadth-first search (BFS)

Use a queue (First-in First-out)

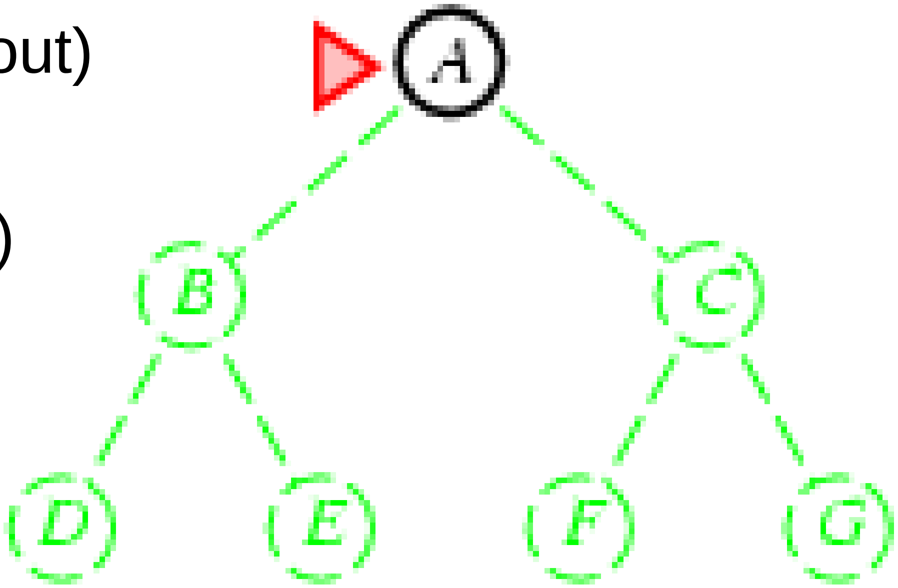
1. en_queue(Initial states)
2. While (queue not empty)
3.   s = de_queue()
4.   if (s==goal) success!
5.   T = succs(s)
6.   en_queue(T)
7. endWhile



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

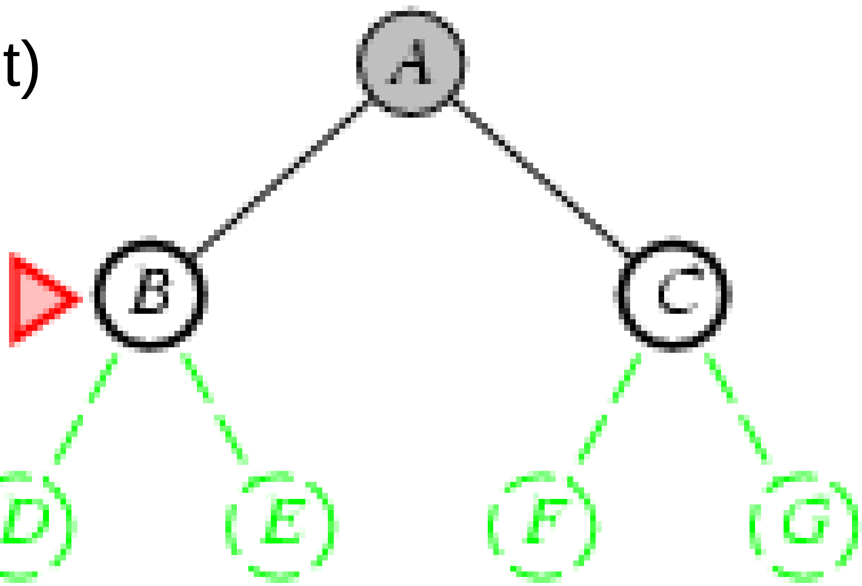


queue (fringe, OPEN)
→ [A] →

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

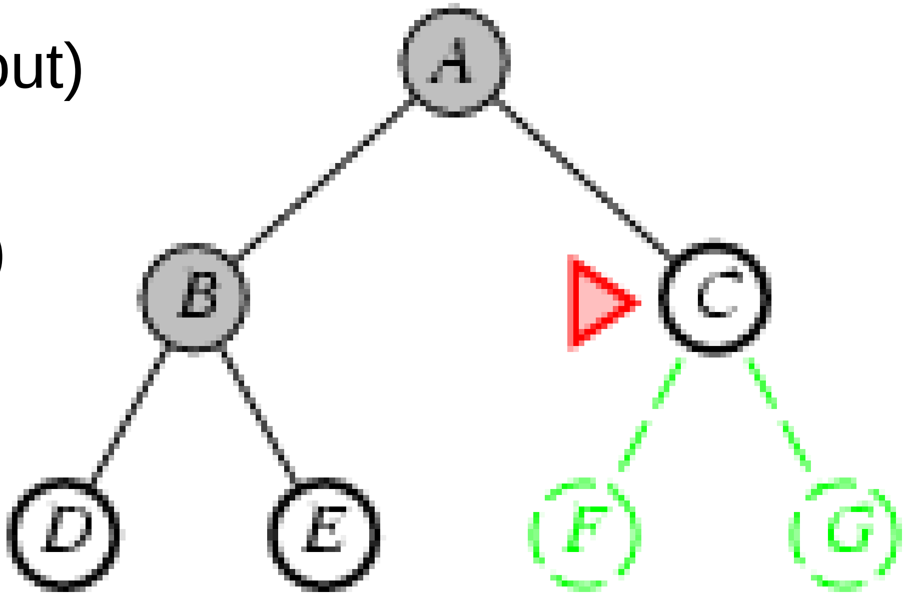


queue (fringe, OPEN)
→ [CB] → A

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

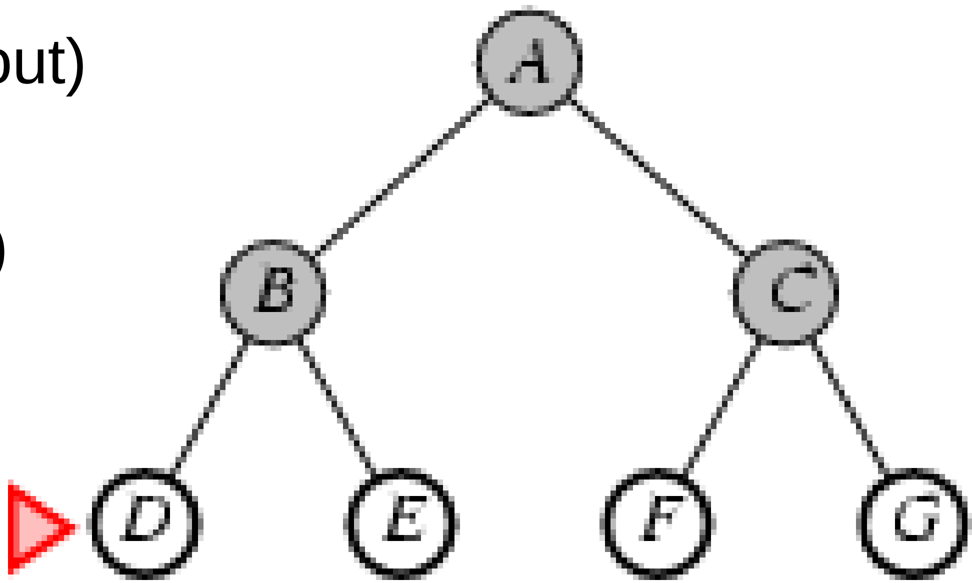


queue (fringe, OPEN)
→ [EDC] → B

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



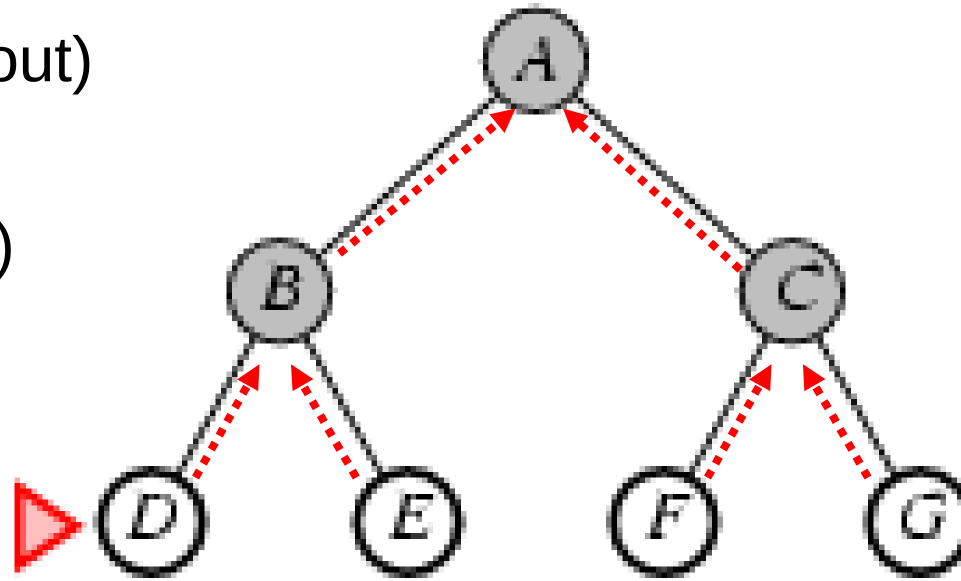
queue (fringe , OPEN)
→[GFED] → C

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)

- en_queue(Initial states)
- While (queue not empty)
-     s = de_queue()
-     if (s==goal) success!
-     T = succs(s)
-     for t in T: t.prev=s
-     en_queue(T)
- endWhile



queue
→[] →G

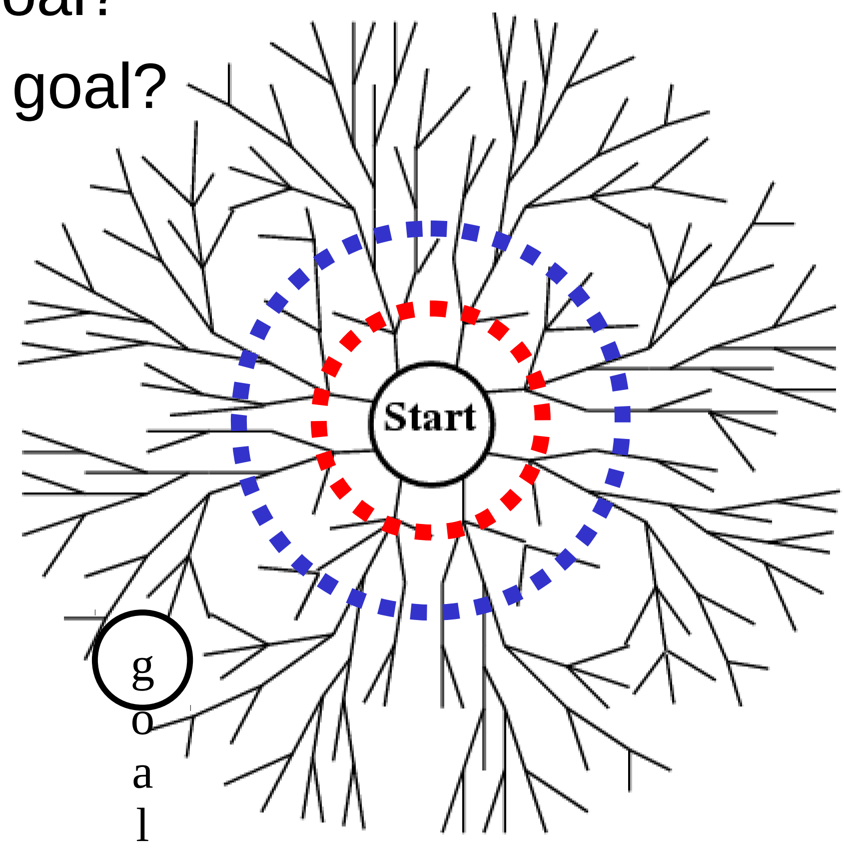
Looking stupid? Indeed. But let’s be consistent…

… until much later we pop 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



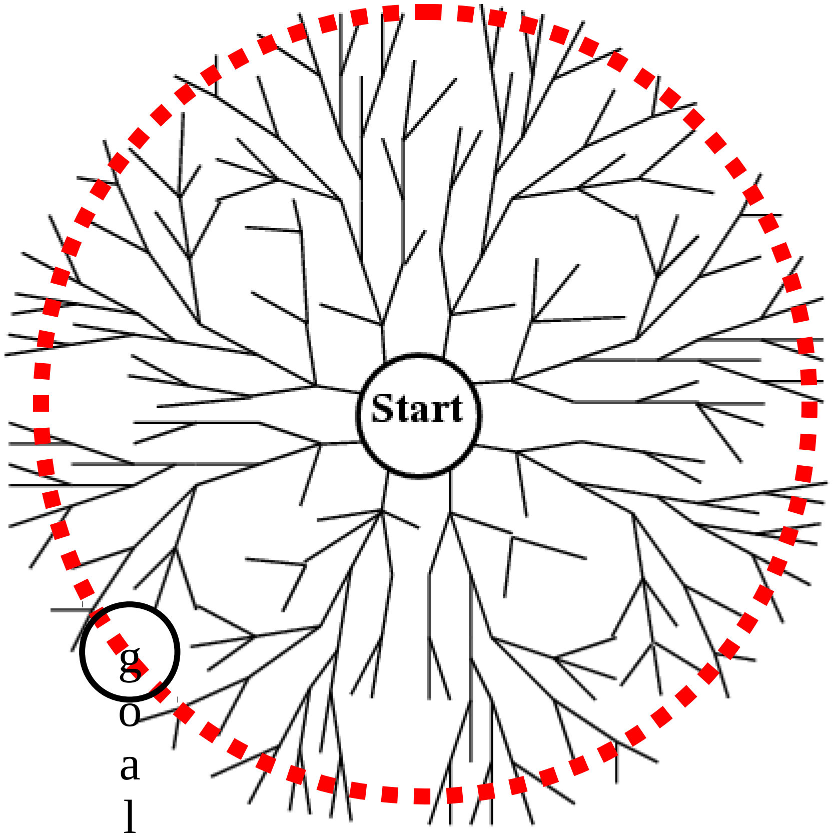
Performance of BFS

Four measures of search algorithms:

- Completeness (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

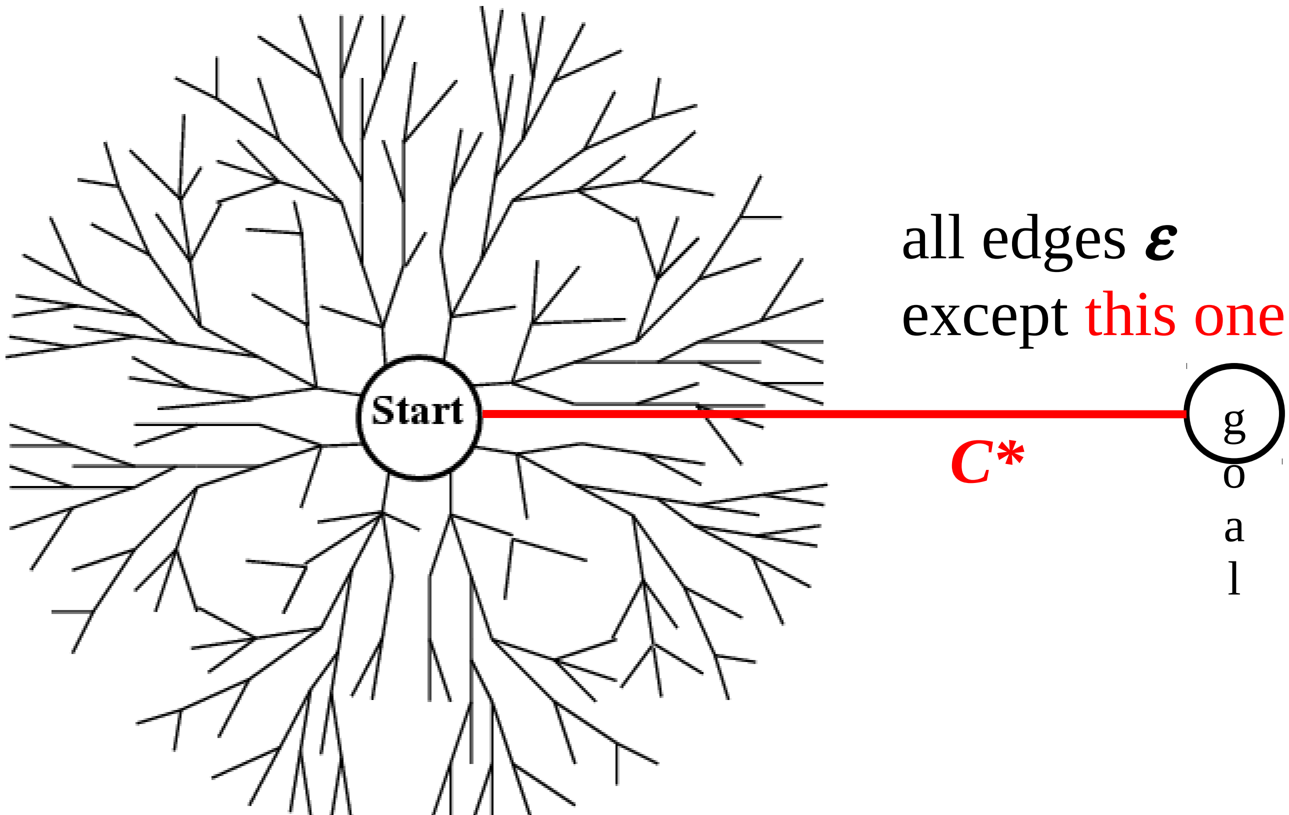
- Find the least-cost goal
- Each node has a path cost from start (= sum of edge costs along the path). Expand the least cost node first.
- Use a priority queue instead of a normal queue
  - Always take out the least cost item
  - Remember *heap*? time $O(log(\#items\ in\ heap))$

That's it*

* Complications on graphs (instead of trees). Later.

Uniform-cost search (UCS)

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



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]	$O(b^d)$	$O(b^d)$
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

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 comple): goal is the last node at radius d.
  - Have to g s at radius d.
  - $b + b^2 + \ldots + b^d \sim O($

Solution: Depth-first search

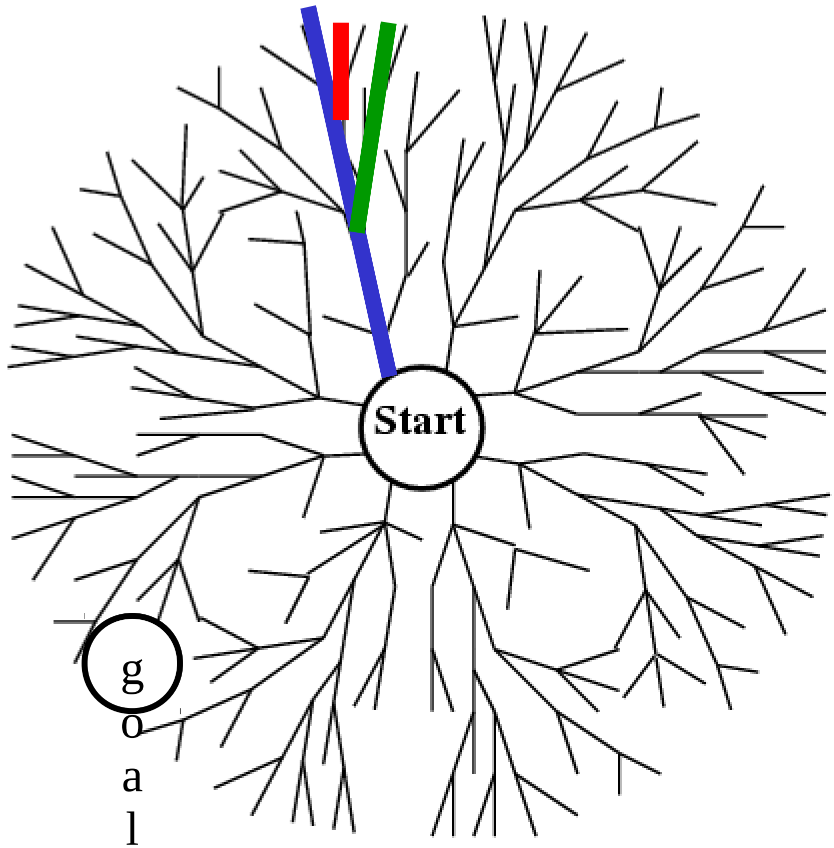
- Space complexity (bad, Figure 3.11)
  - Back points for all generated nodes $O(b^d)$
  - The queue (smaller, but still $O(b^d)$)

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…

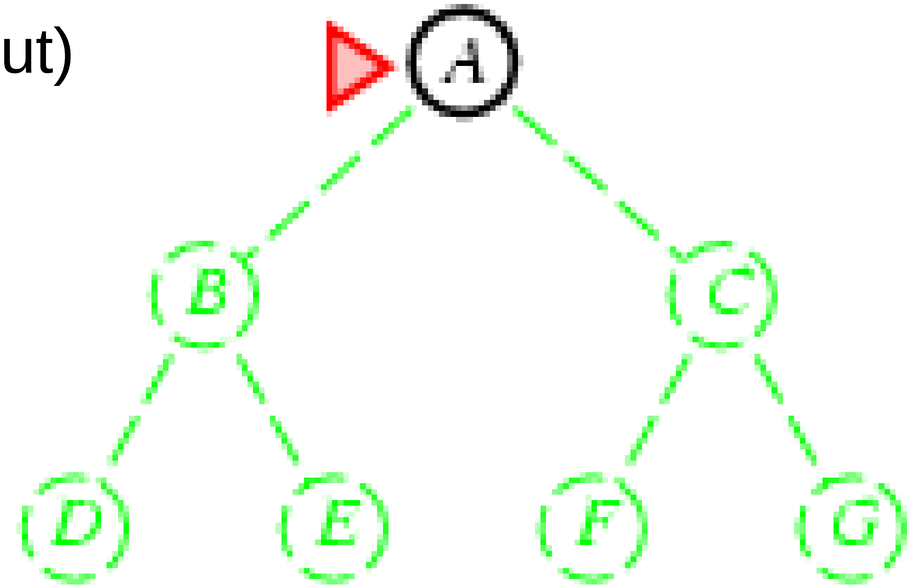
fan



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



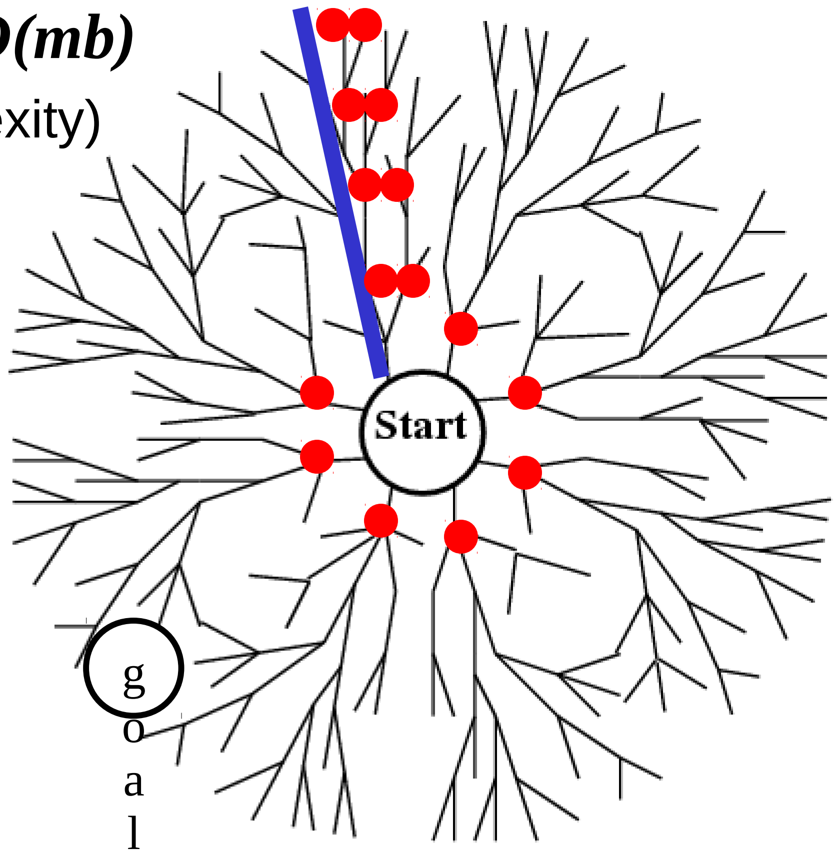
stack (fringe)

[] ⬄

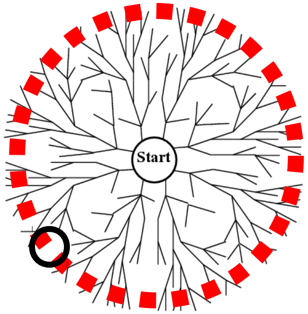
What's in the fringe for DFS?

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

(Space complexity)



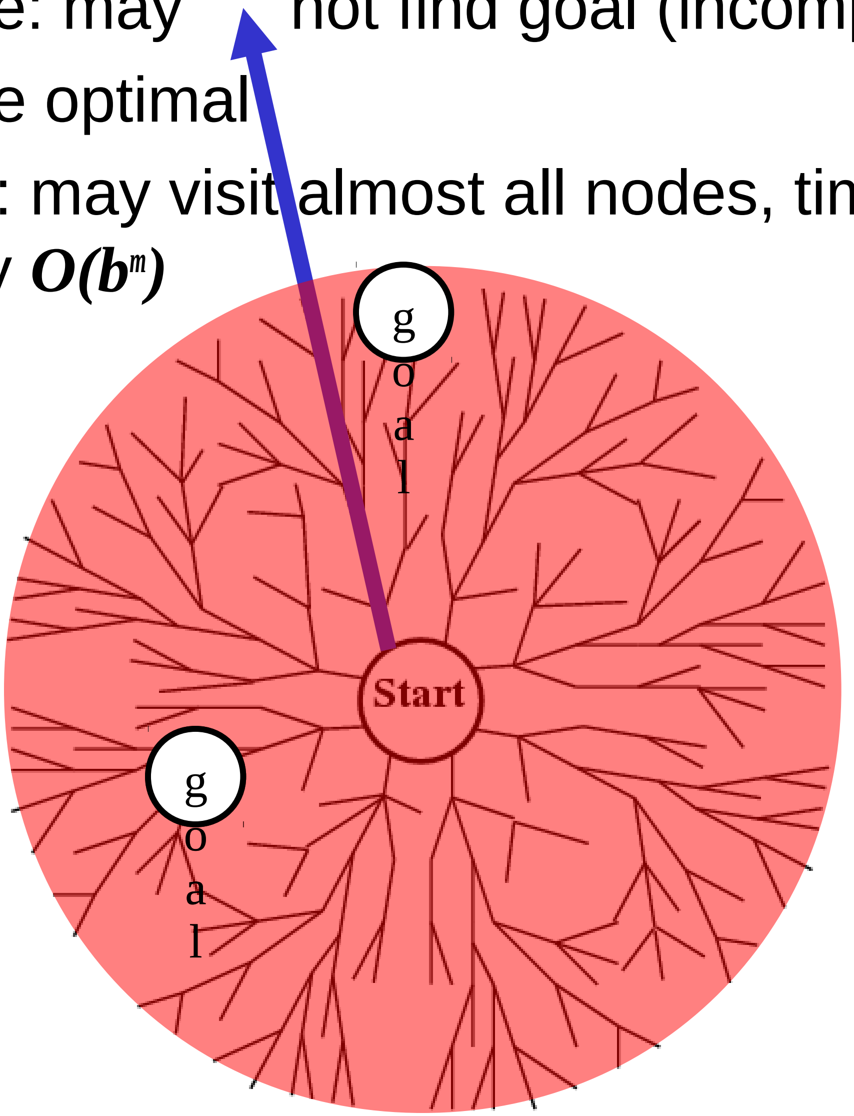
c.f. BFS $\boldsymbol{O(b^d)}$



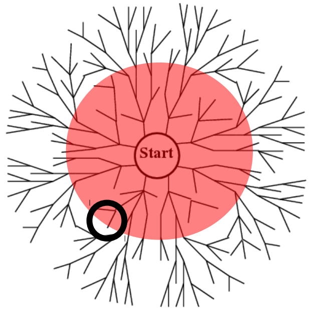
- "backtracking search" even less space
  - generate siblings (if applicable)

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

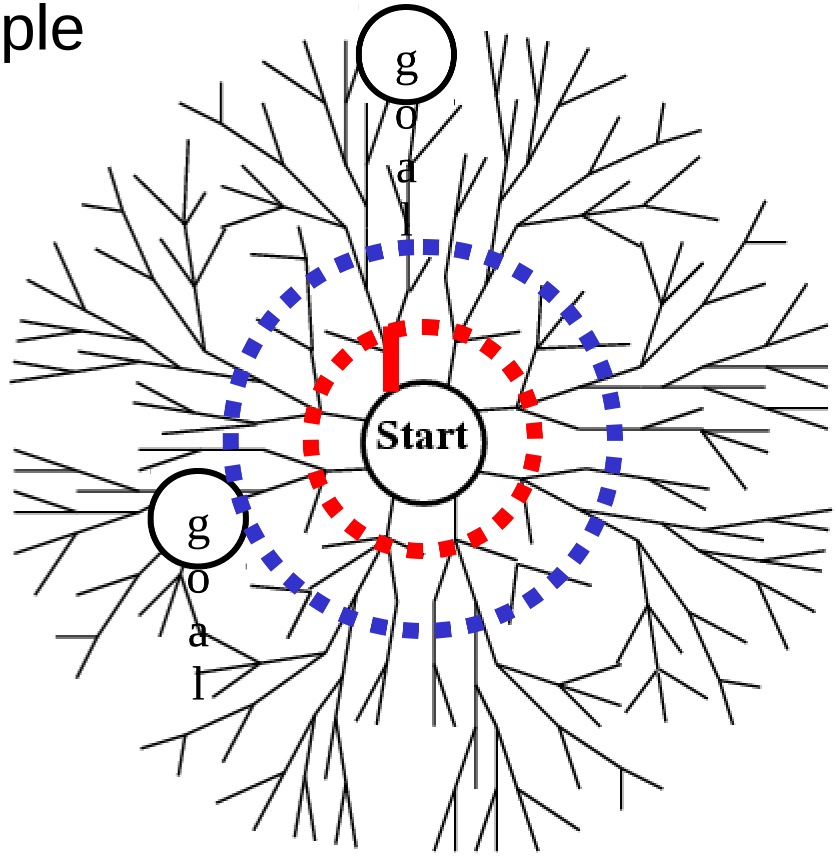
	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$

1. edge cost constant, or positive non-decreasing in depth
- 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
- A huge waste?
  - Each deepening repeats DFS from the beginning
  - No! $db+(d-1)b^2+(d-2)b^3+\ldots+b^d \sim O(b^d)$
  - 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
- edge costs $\geq \varepsilon > 0$. C^* is the best goal path cost.

Performance of search algorithms on trees

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

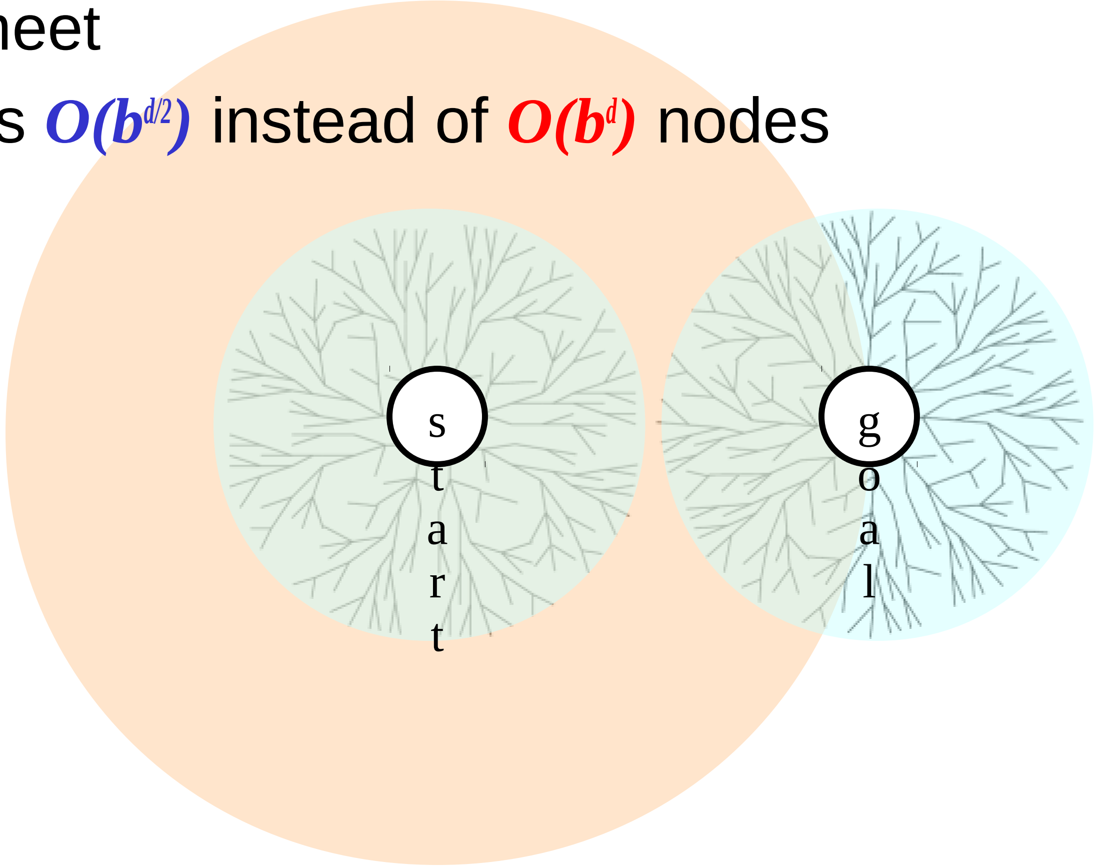
	Complete	optimal	time	space
Breadth-first search	Y		$O(b^d)$	$O(b^d)$
U se			$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
D se			$O(b^m)$	$O(bm)$
Ite de		Y, if [1]	$O(b^d)$	$O(bd)$

How to reduce the number of states we have to generate?

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

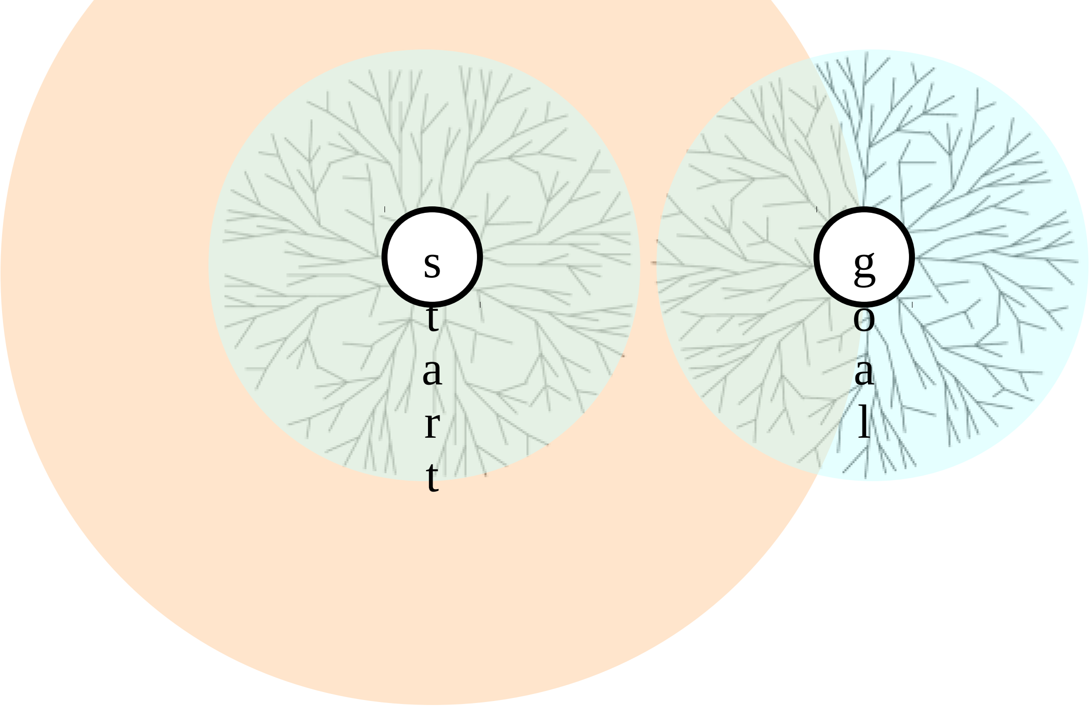
Bidirectional search

- Breadth-first search from both start and goal
- Fringes meet
- Generates $O(b^{d/2})$ instead of $O(b^d)$ nodes



Bidirectional search

- But
  - The fringes are $O(b^{d/2})$
  - How do you start from the 8-queens goals?



Performance of search algorithms on trees

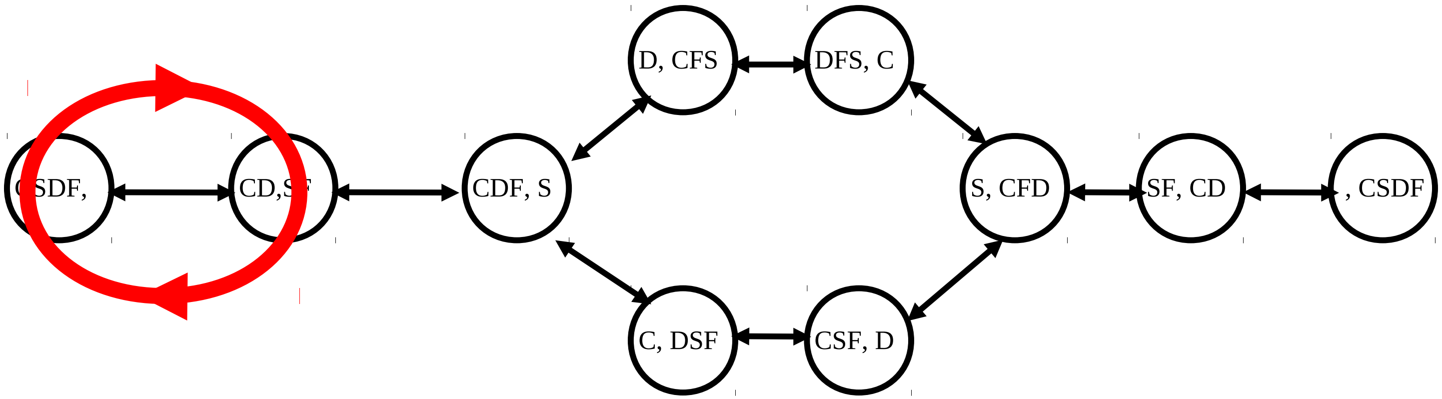
b: branching factor (assume finite)   d: goal depth   m: graph depth

	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]	$O(b^d)$	$O(bd)$
Bidirectional search[3]	Y	Y, if [1]	$O(b^{d/2})$	$O(b^{d/2})$

1. edge cost constant, or positive non-decreasing in depth
- edge costs $\geq \varepsilon > 0$. C* is the best goal path cost.
- both directions BFS; not always feasible.

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?

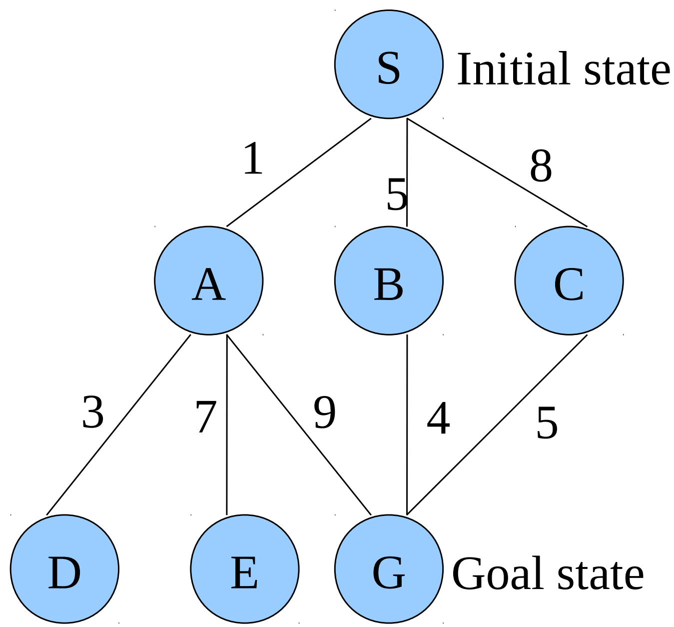
If state space graph is not a tree

- 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.

If state space graph is not a tree

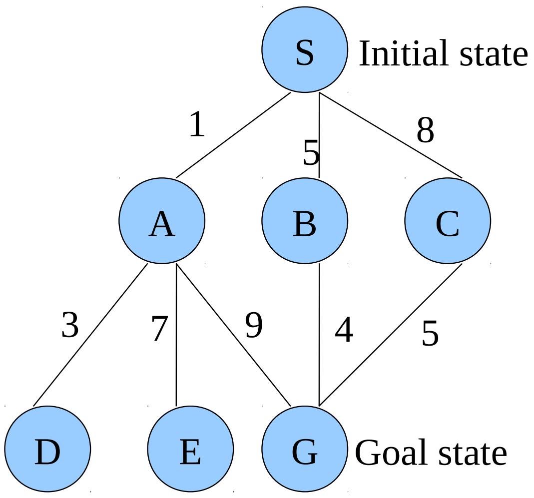
- BFS:
  - Still $O(b^d)$ space complexity, not worse
- DFS:
  - Known as Memorizing DFS (MEMDFS)
    - Space and time now $O(min(N, b^M))$ – much worse!
    - N: number of states in problem
    - M: length of longest cycle-free path from start to anywhere
  - Alternative: Path Check DFS (PCDFS): remember only expanded states on current path (from start to the current node)
    - Space O(M)
    - Time $O(b^M)$

Example



Example

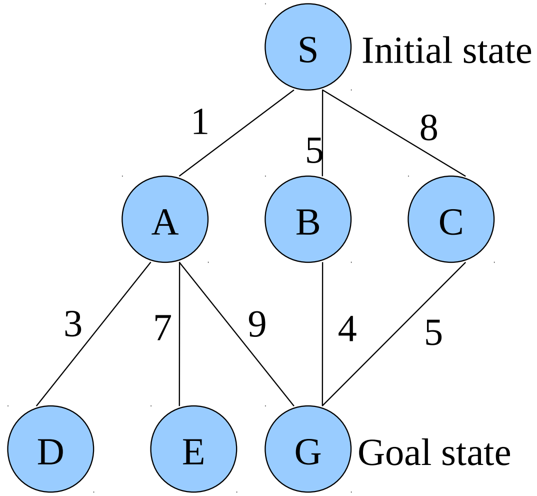
- Depth-First Search: S A D E G
  Solution found: S A G
- Breadth-First Search: S A B C D E G
  Solution found: S A G
- Uniform-Cost Search: S A D B C E G
  Solution found: S B G (This is the only ur
  worries about costs.)
- Iterative-Deepening Search: S A B C S A D E G
  Solution found: S A G



Depth-First Search

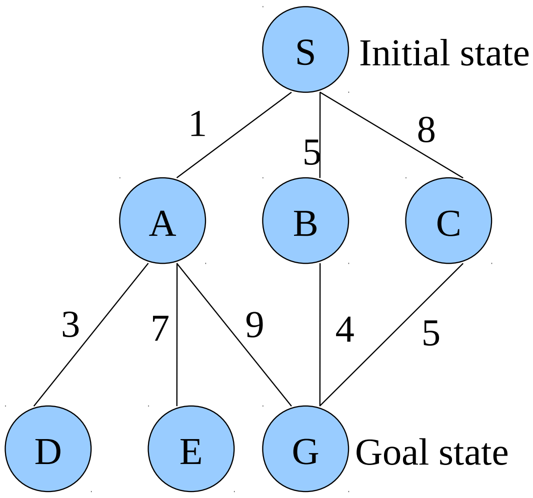
```
expanded
node          nodes list
----          -----------
              { S }
  S           { A B C }
  A           { D E G B C}
  D           { E G B C }
  E           { G B C }
  G           { B C }

Solution path found is S A G  <-- this G has cost 10
Number of nodes expanded (including goal node) = 5
```



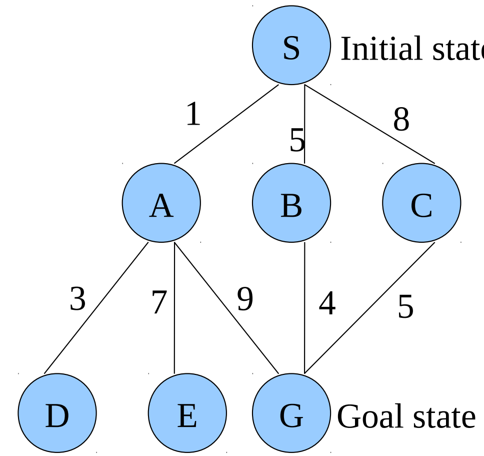
Breadth-First Search

```
expanded
node           nodes list
----           ----------
               { S }
  S            { A B C }
  A            { B C D E G }
  B            { C D E G G' }
  C            { D E G G' G" }
  D            { E G G' G" }
  E            { G G' G" }
  G            { G' G" }
```




```
Solution path found is S A G  <-- this G also has cost 10
Number of nodes expanded (including goal node) = 7
```

Uniform-Cost Search




```
expanded
node          nodes list
----          ----------
              { S }
  S           { A(1) B(5) C(8) }
  A           { D(4) B(5) C(8) E(8) G(10) }   (note, we don't return G)
  D           { B(5) C(8) E(8) G(10) }
  B           { C(8) E(8) G(9) G(10) }
  C           { E(8) G(9) G(10) G(13) }
  E           { G(9) G(10) G(13) }
  G           { }

Solution path found is S B G  <-- this G has cost 9, not 10
Number of nodes expanded (including goal node) = 7
```

Take-home:

- **Problem solving as search**
- **Uninformed search**
  - ➢Breadth-first search
  - ➢Uniform-cost search
  - ➢Depth-first search
  - ➢Iterative deepening
  - ➢Bidirectional search
- **Can you unify them (except bidirectional) using the same algorithm, with different priority functions?**
- **Performance measures:**
  Completeness, optimality, time complexity, space complexity