State-Space Formulation

- **Intelligent agents: problem solving as search**

- Search consists of
  - state space
  - operators
  - start state
  - goal states
- The search graph

- **A** Search Tree **is an effective way to represent the search process**

- **There are a variety of search algorithms, including**
  - Depth-First Search
  - Breadth-First Search
  - Others which use heuristic knowledge (in future lectures)

Uninformed search strategies

uninformed (or blind) search:

While searching you have no clue whether one non-goal state is better than any other. Your search is blind. You don't know if your current exploration is likely to be fruitful.

common blind strategies:

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Iterative deepening search

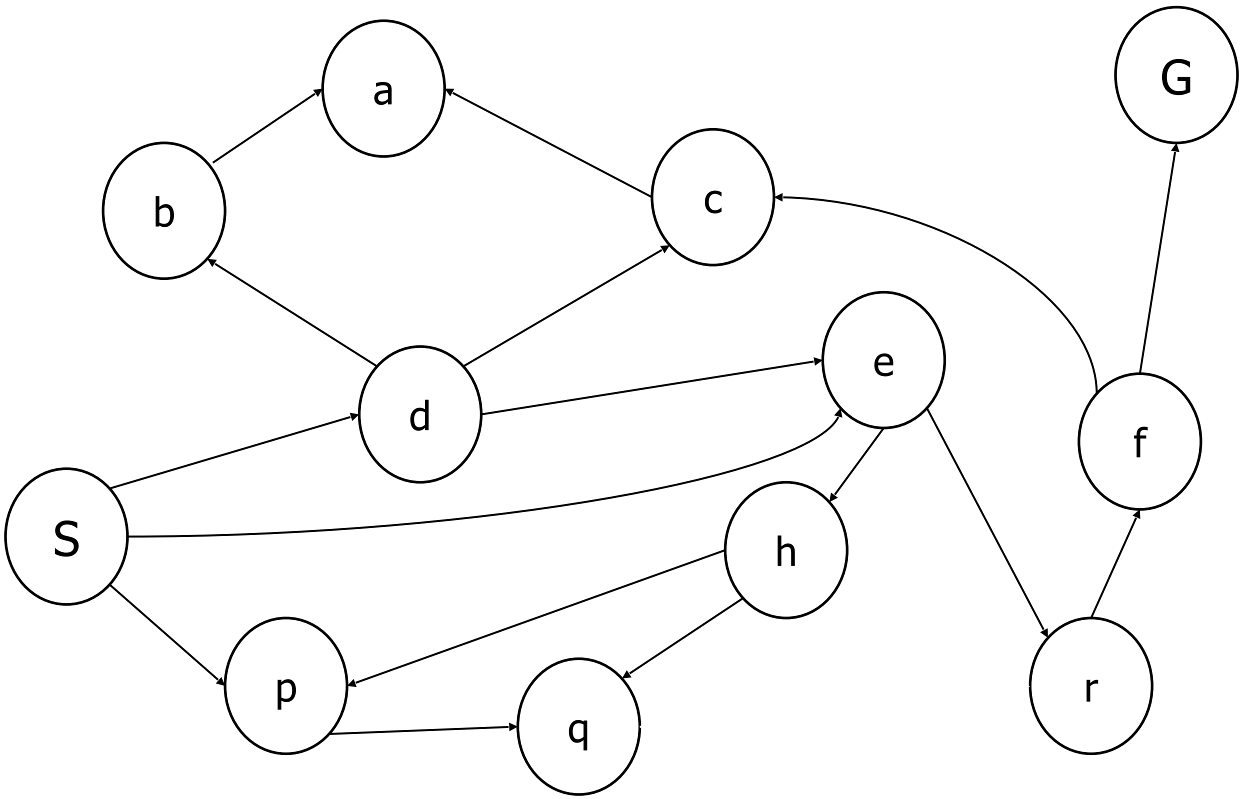
General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

- Important details:
  - Fringe (frontier)
  - Expansion
  - Exploration strategy

- Main question: which fringe nodes to explore?

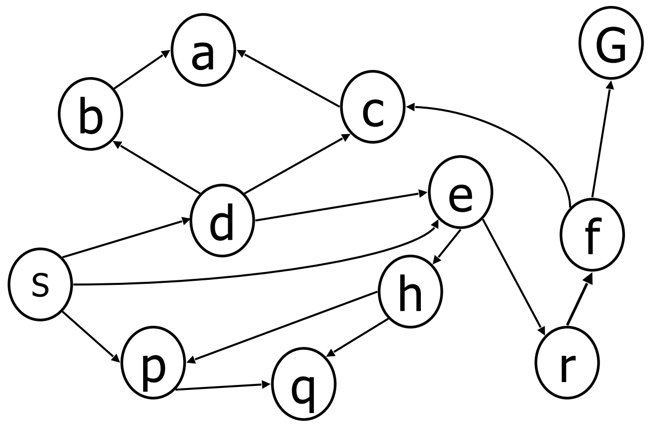
Graph representation of the landscape explored



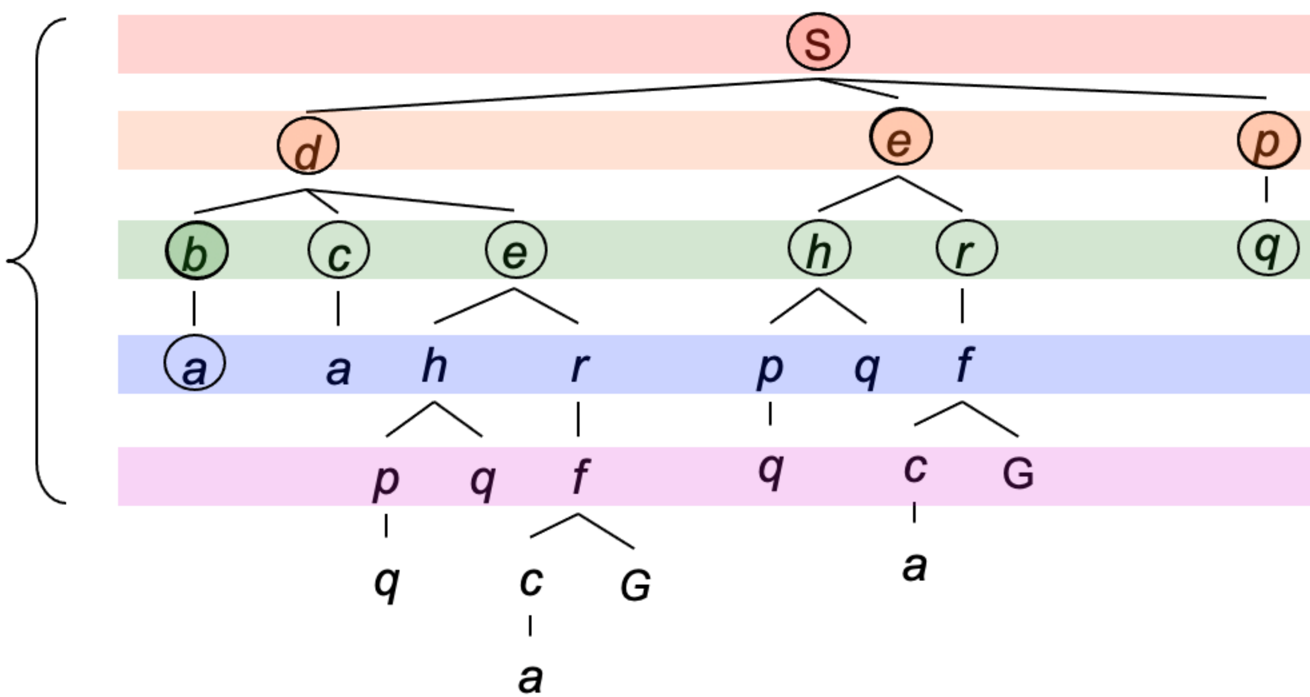
Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue



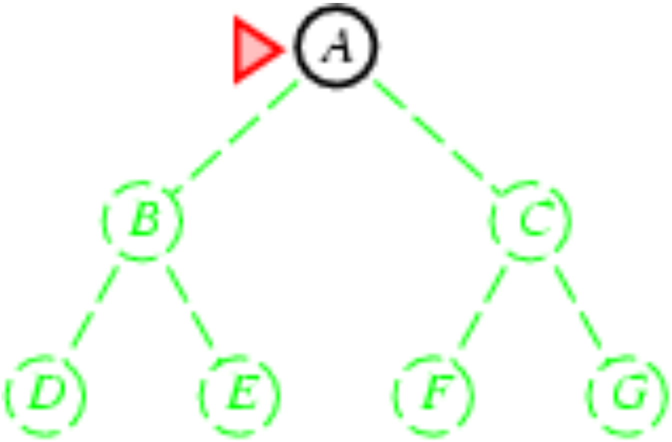
Search Tiers



Breadth-first search

- Expand shallowest unexpanded node
- Fringe: nodes waiting in a queue to be explored, also called OPEN
- Implementation:
    - *fringe* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.

Is A a goal state?

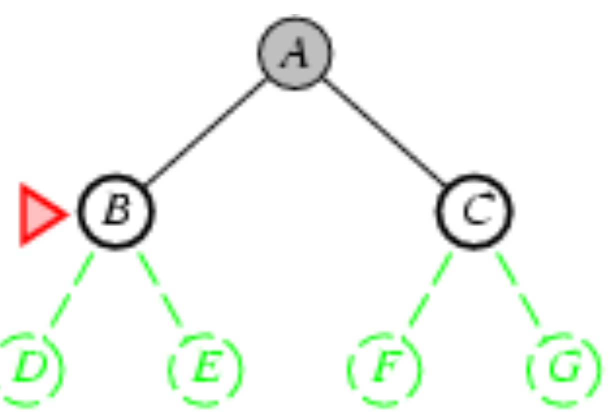


Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

Expand:
fringe = [B,C]

Is B a goal state?

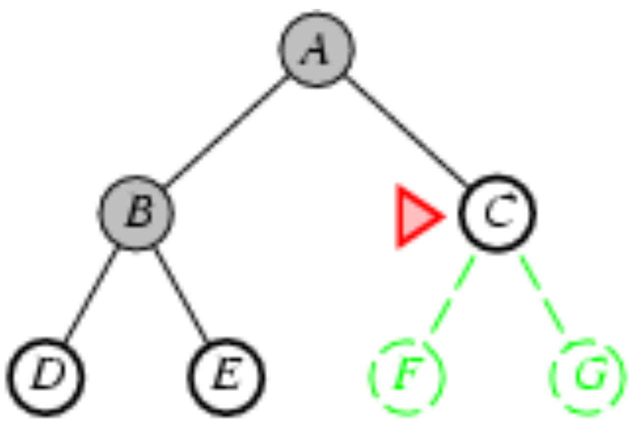


Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

Expand:
fringe=[C,D,E]

Is C a goal state?

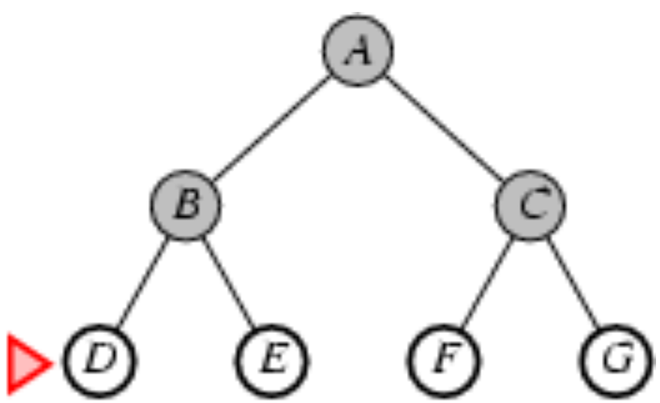


Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

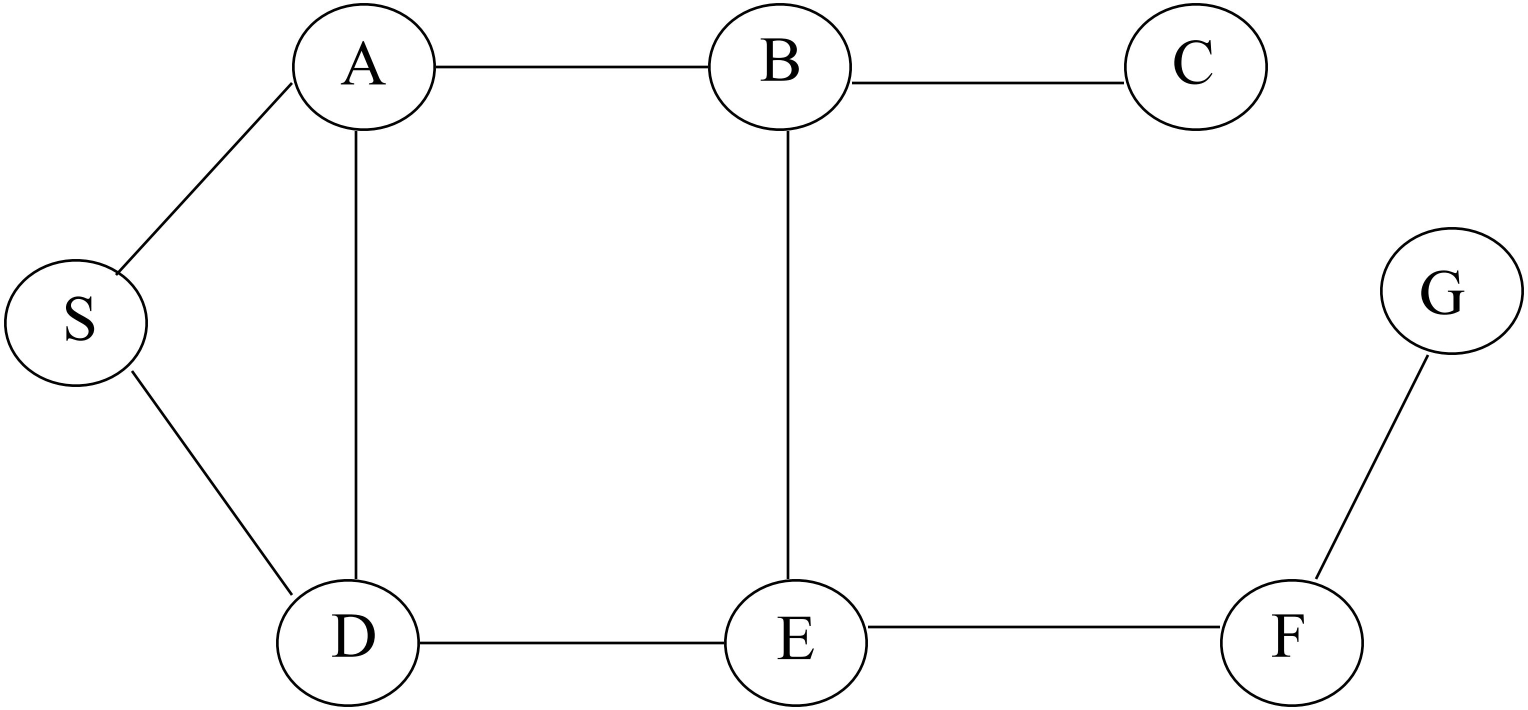
Expand:
fringe=[D,E,F,G]

Is D a goal state?



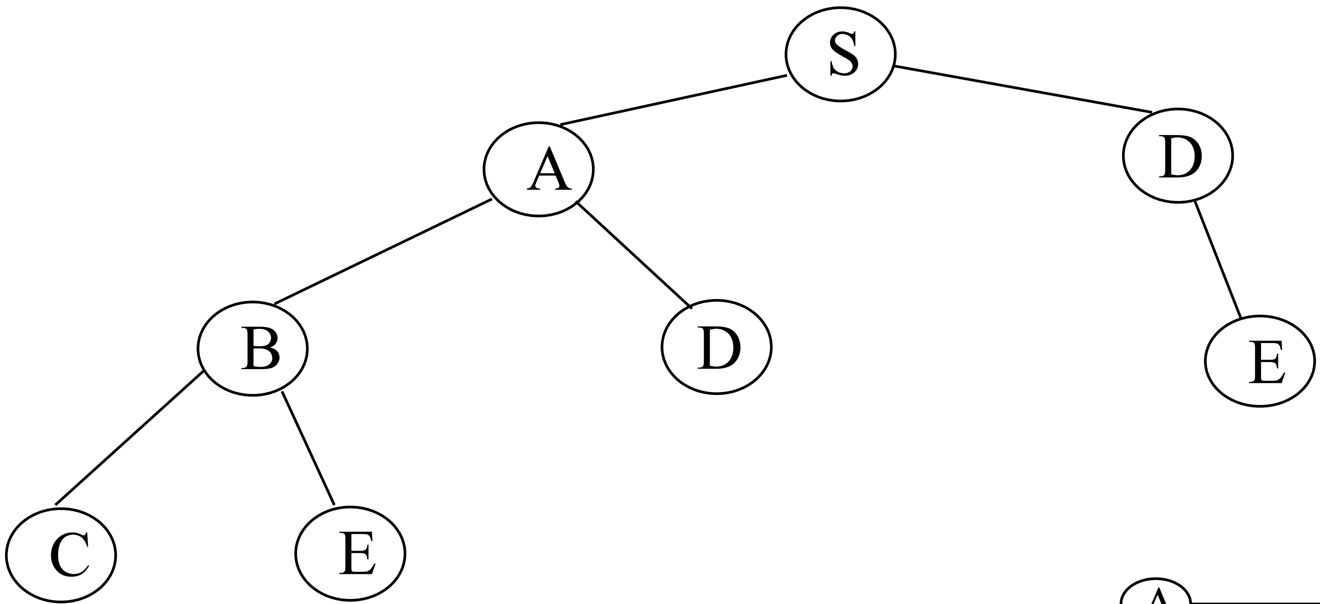
BFS for 8 puzzle

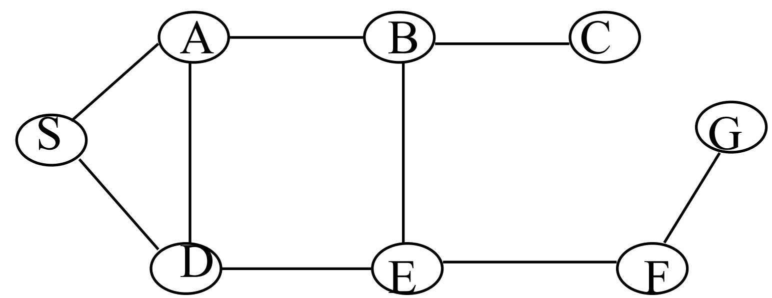
Example: Map Navigation



S = start, G = goal, other nodes = intermediate states, links = legal transitions

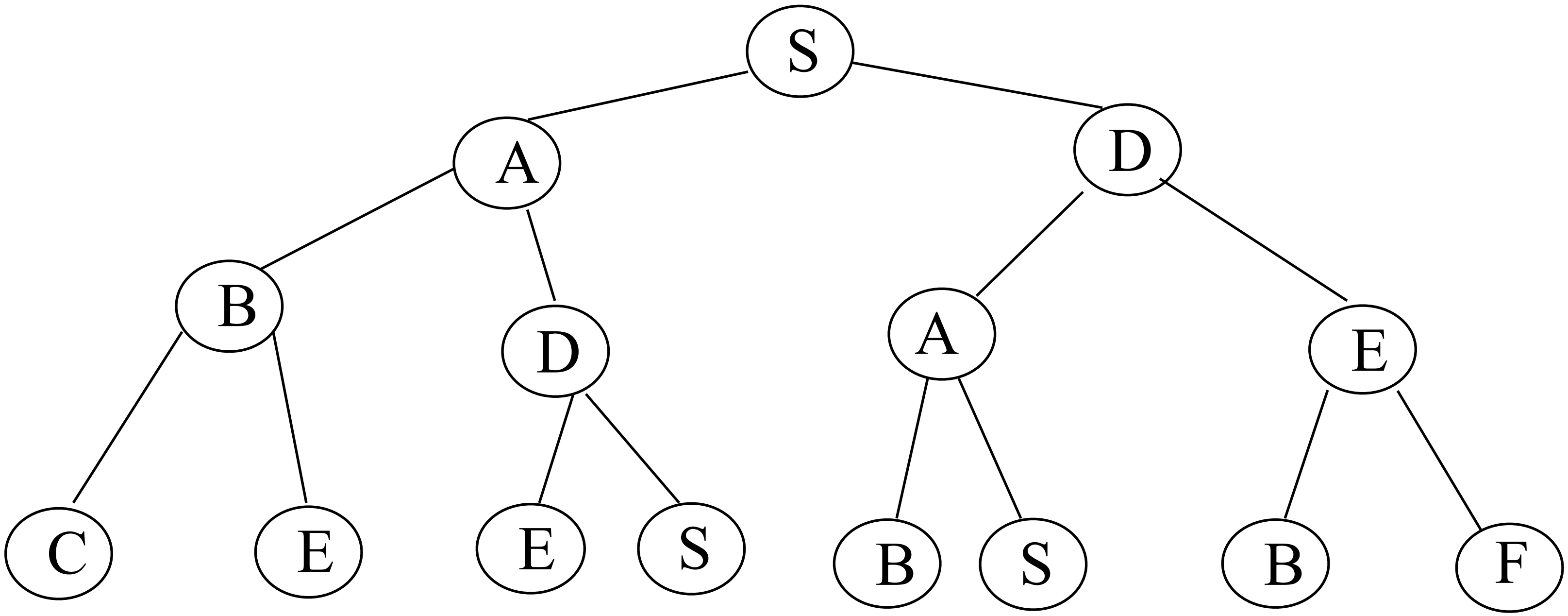
Initial BFS Search Tree





Note: this is the search tree at some particular point in
in the search.

Breadth First Search Tree (BFS)



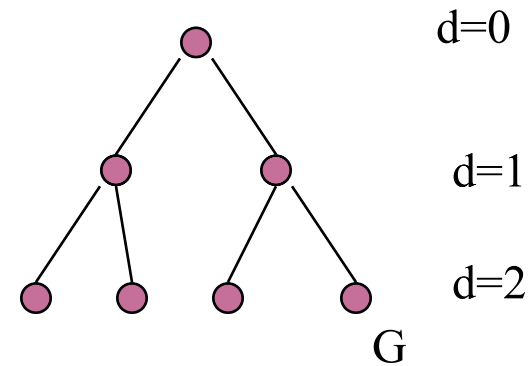
Here BFS is implemented as a tree search with only parent node not added as a child node.

What is the Complexity of Breadth-First Search?

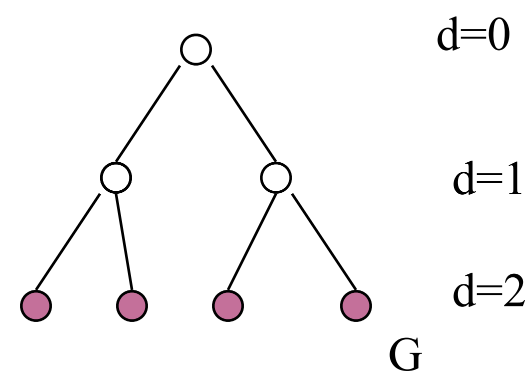
- Time Complexity
  - assume (worst case) that there is 1 goal leaf at the RHS
  - so BFS will **expand** all nodes

    $= 1 + b + b^2 + \ldots\ldots\ldots + b^d$

    $= \mathbf{O\ (b^{d+1})}$



- Space Complexity
  - how many nodes can be in the queue (worst-case)?
  - at depth d there are b^d **unexpanded** nodes in the Q $= \mathbf{O\ (b^d)}$



– Time and space of number of generated nodes is O (b^{d+1})

Examples of Time and Memory Requirements for *tree search version* of Breadth-First Search

Depth of Solution	Nodes Expanded	Time	Memory
0	1	1 millisecond	100 bytes
2	111	0.1 seconds	11 kbytes
4	11,111	11 seconds	1 megabyte
8	10^8	31 hours	11 giabytes
12	10^{12}	35 years	111 terabytes

Assuming b=10, 1000 nodes/sec, 100 bytes/node

Breadth-First Search (BFS) Properties

- Complete (with find a solution in a finite number of steps if one exists)
- Solution Length: optimal (assuming unit cost per move)
- (Can) expand each node once (if checks for duplicates)
- Search Time: $O(b^d)$ which is the size of the state space
- Memory Required: $O(b^d)$
- Drawback: requires space proportional to the state-space (Search time is unavoidable)

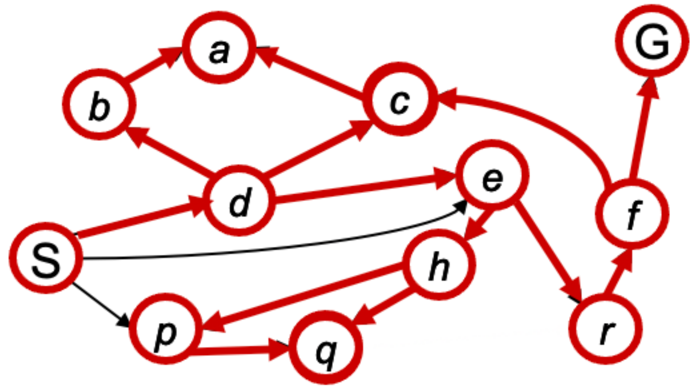
Depth-First-Search

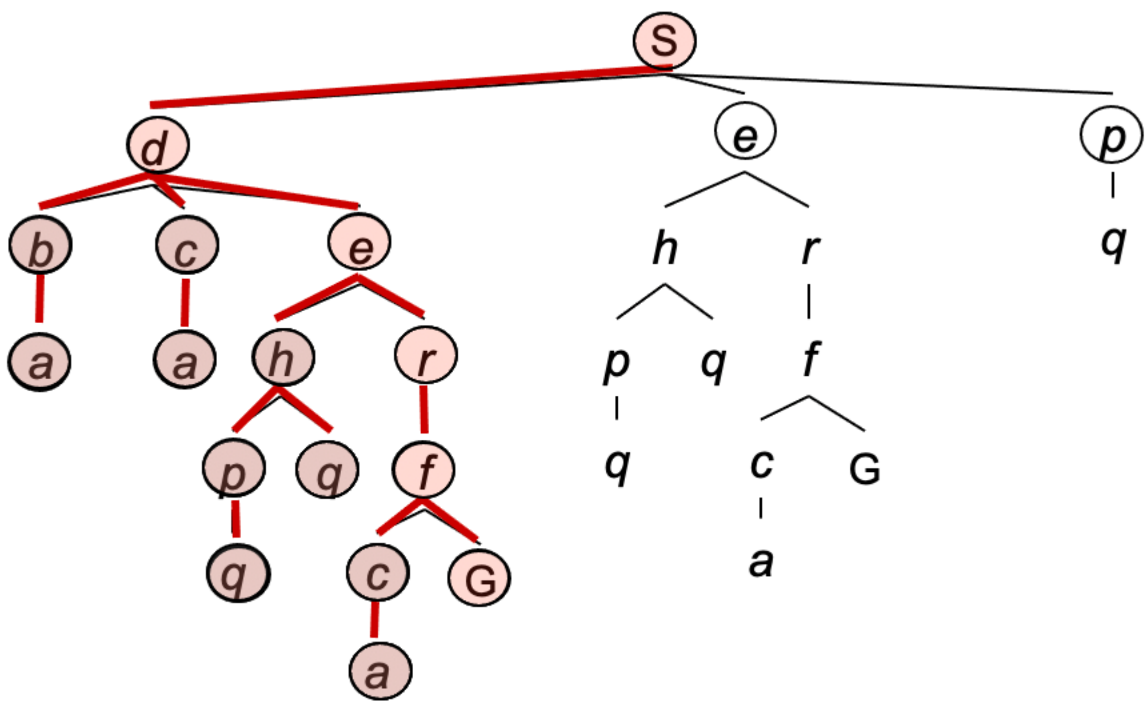
1. Put the start node *s* on OPEN
2. If OPEN is empty exit with failure.
3. Remove the first node *n* from OPEN and place it on CLOSED.
4. If *n* is a goal node, exit successfully with the solution obtained by tracing back pointers from *n* to *s*.
5. Otherwise, <u>expand *n*</u>, generating all its successors attach to them pointers back to *n*, and put them at the top of OPEN *in some order*.
6. Go to step 2.

Depth-First Search

Strategy: exp
a deepest no
first

Implementati
Fringe is a LIF
stack





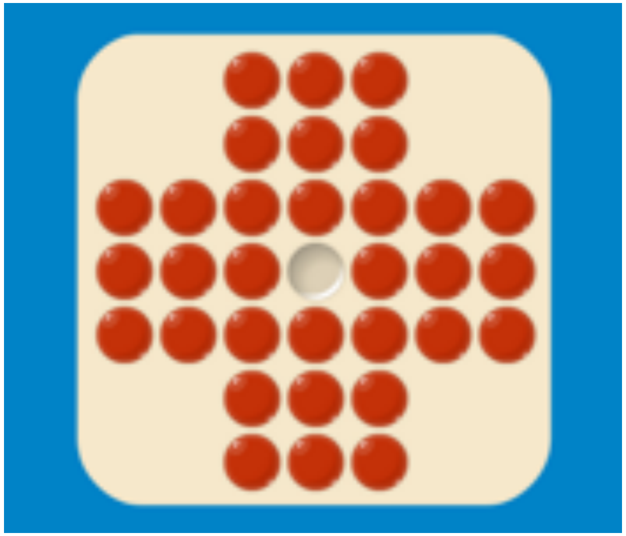
Depth-First-Search (tree search version)

Recursive version of DFS:

```
State DepthFirstSearch(node) {
  if (goalTest(node)) return node;
for each n in successors(node, operators)
   {
     result = DepthFirstSearch(n);
     if (result != FAIL) return result;
   }
return FAIL;
}
```

Example 1

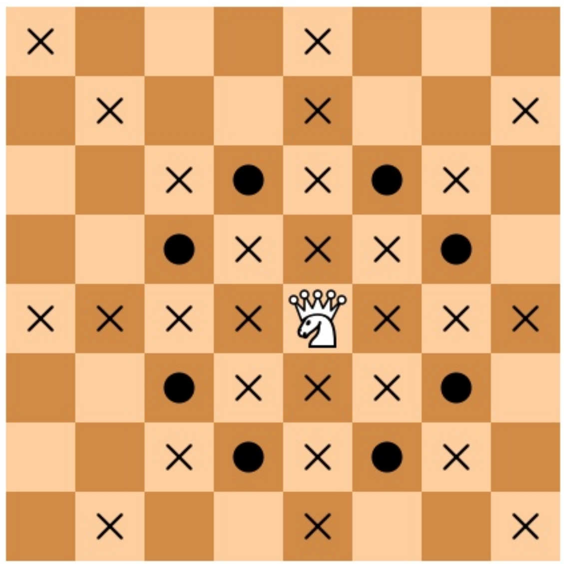
- peg solitaire



Online game: https://webgamesonline.com/peg-solitaire/

Example 2

- placement of mutually non-attacking queens (amazons) in a checker-board. Amazon in row 5 and column 5 attacks all squares marked with x or O



- Maximum of N ar on an N x N board. For what N can we actuall

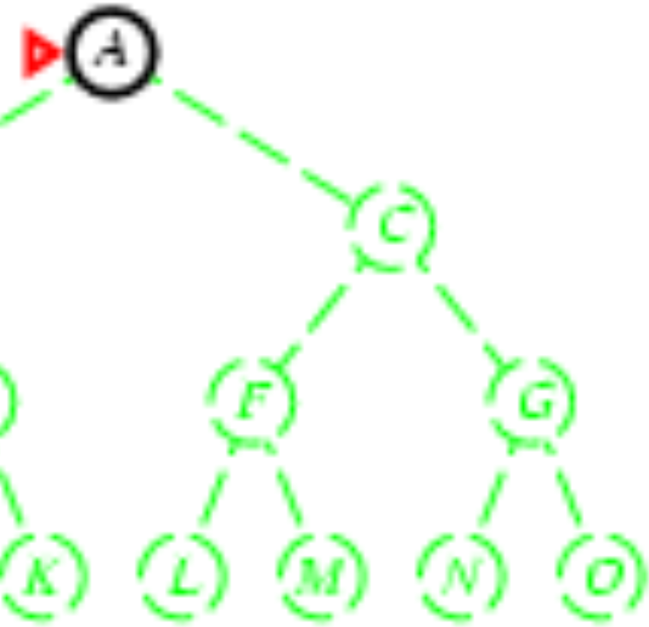
Graph structure of state space graph

- The graph could be finite or infinite
- The graph could be directed or undirected
- The graph could be cyclic or acyclic
- The graph could be a tree (or not)

- Irrespective of the structure, the search algorithms fall into tree search or graph search version.

Depth-first search

Is A a goal state?

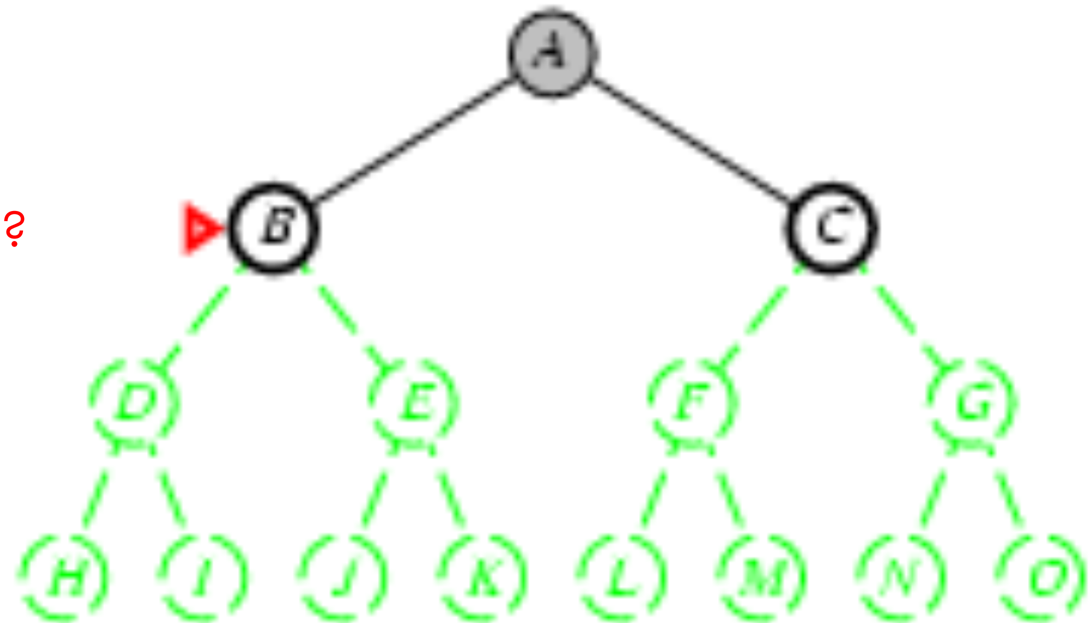


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[B,C]

Is B a goal state?

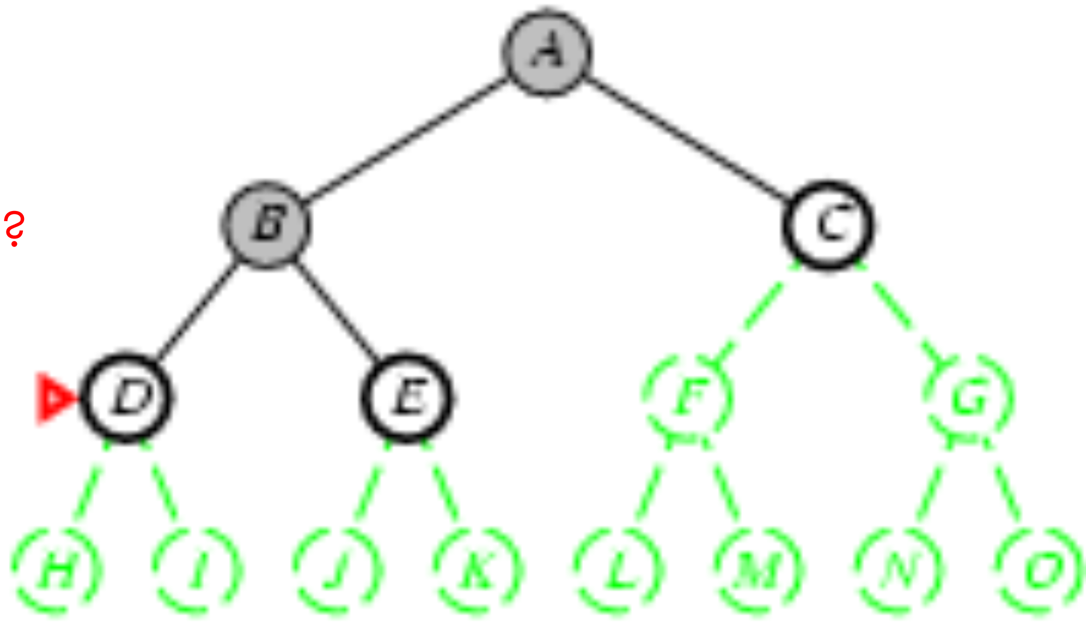


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[D,E,C]

Is D = goal state?

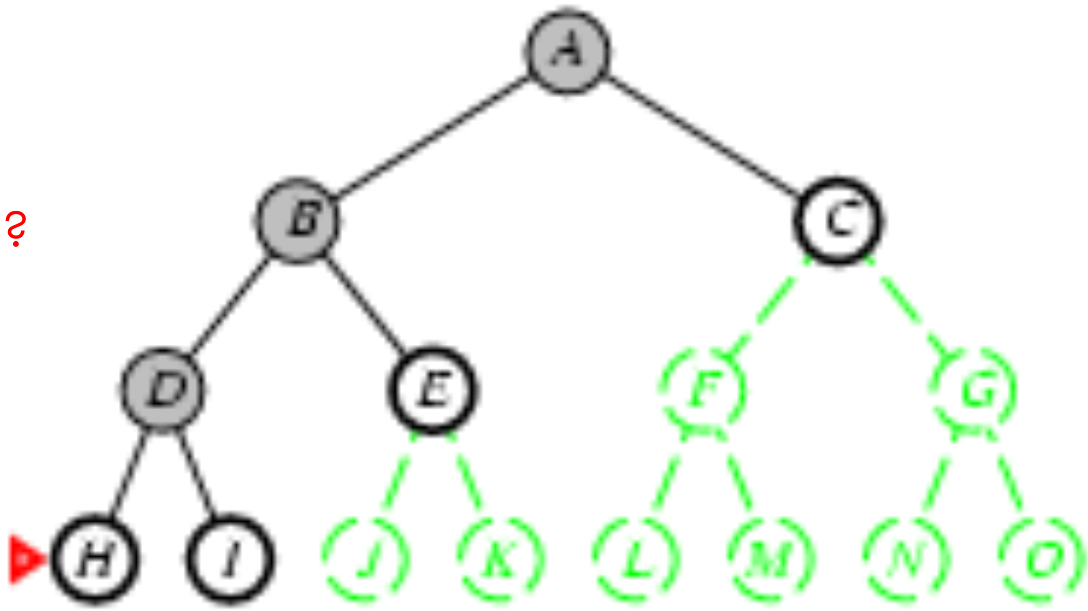


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[H,I,E,C]

Is H = goal state?

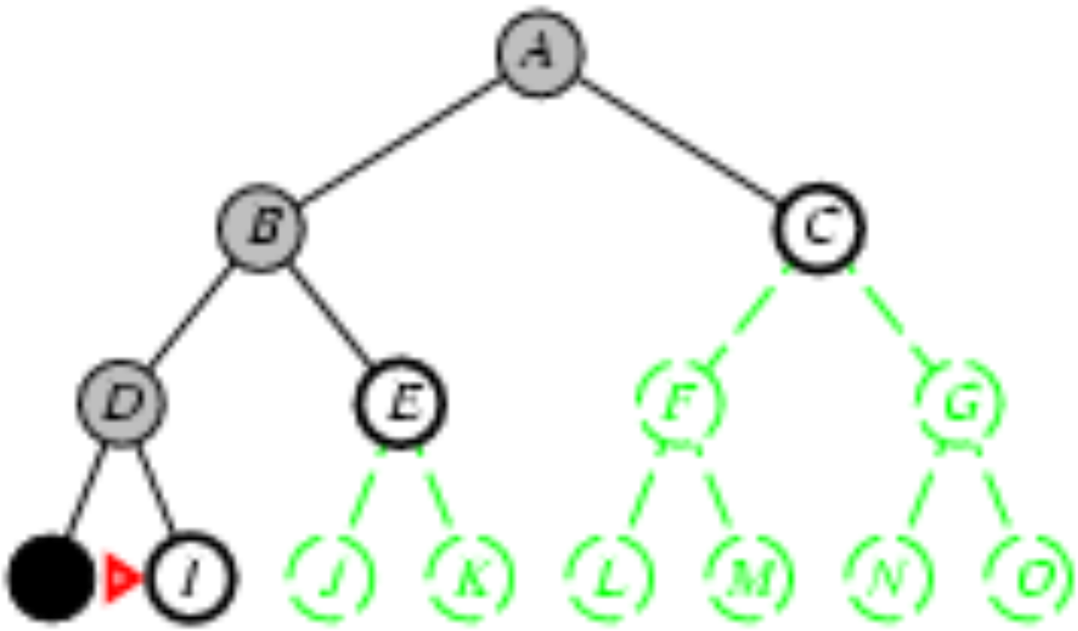


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[I,E,C]

Is I = goal state?

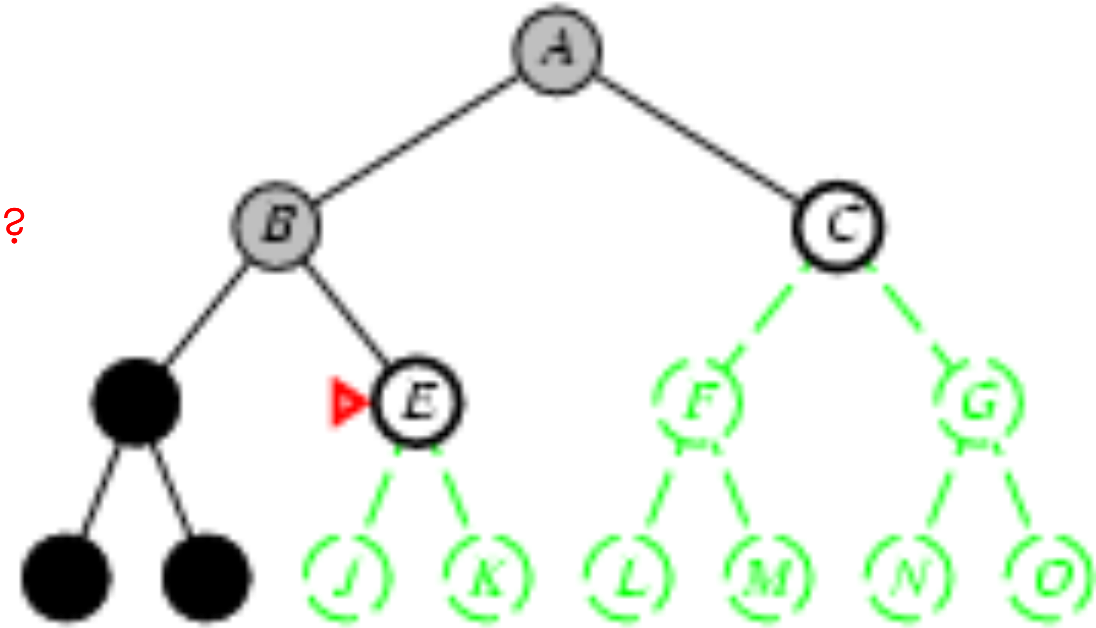


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[E,C]

Is E = goal state?

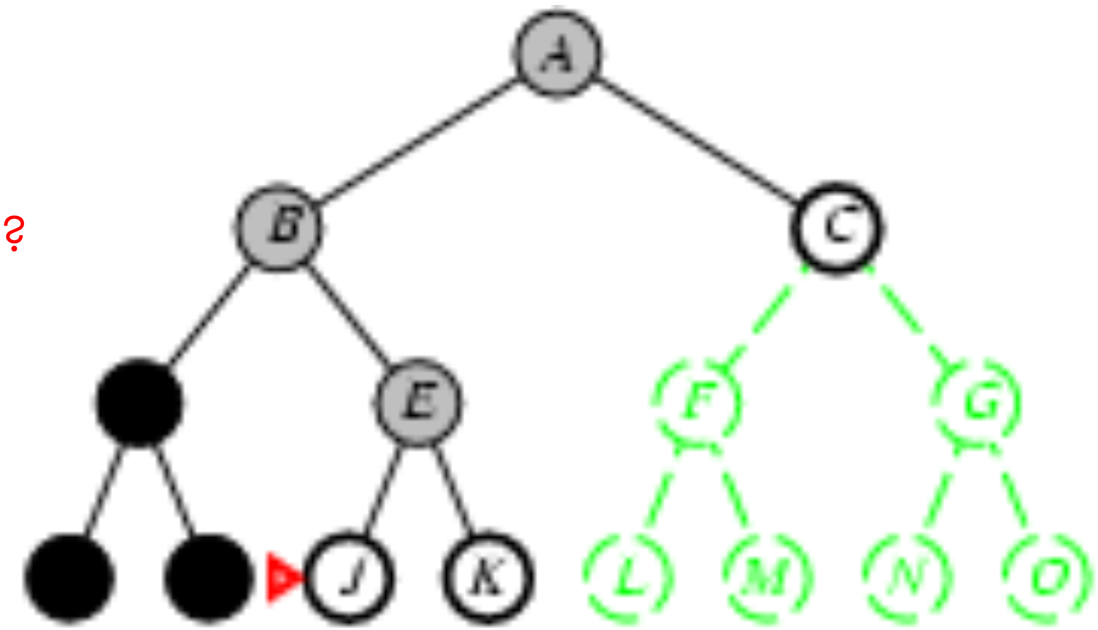


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[J,K,C]

Is J = goal state?

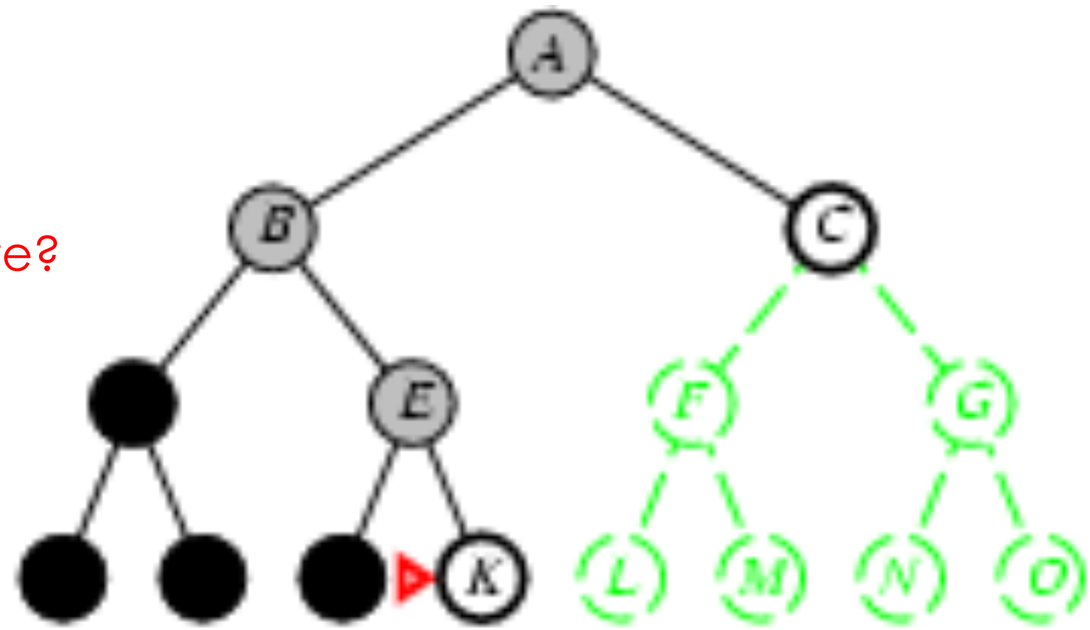


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[K,C]

Is K = goal state?

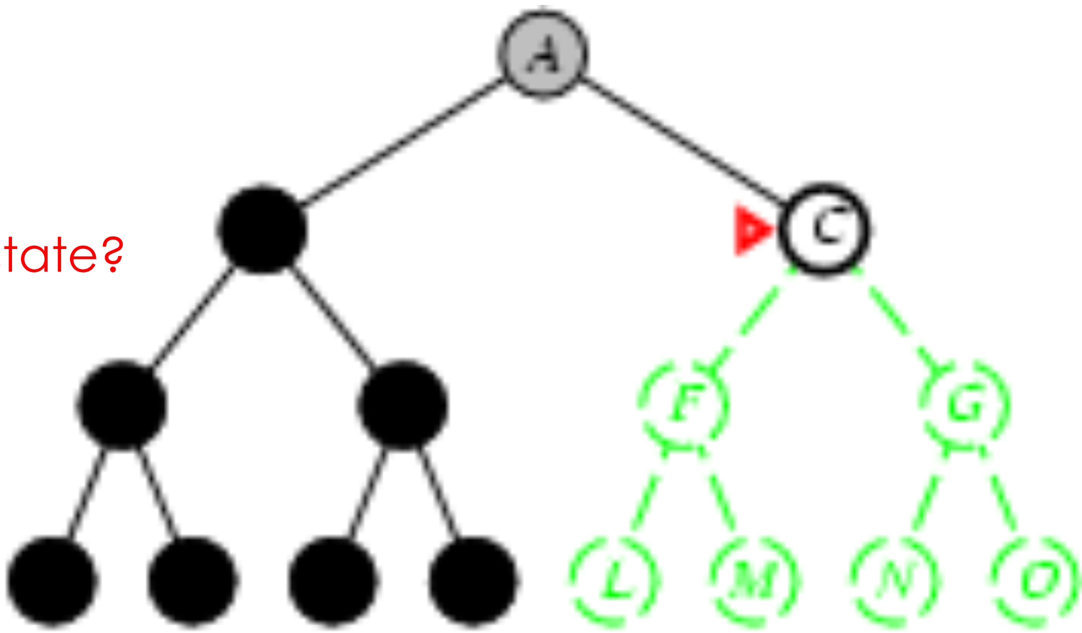


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[C]

Is C = goal state?

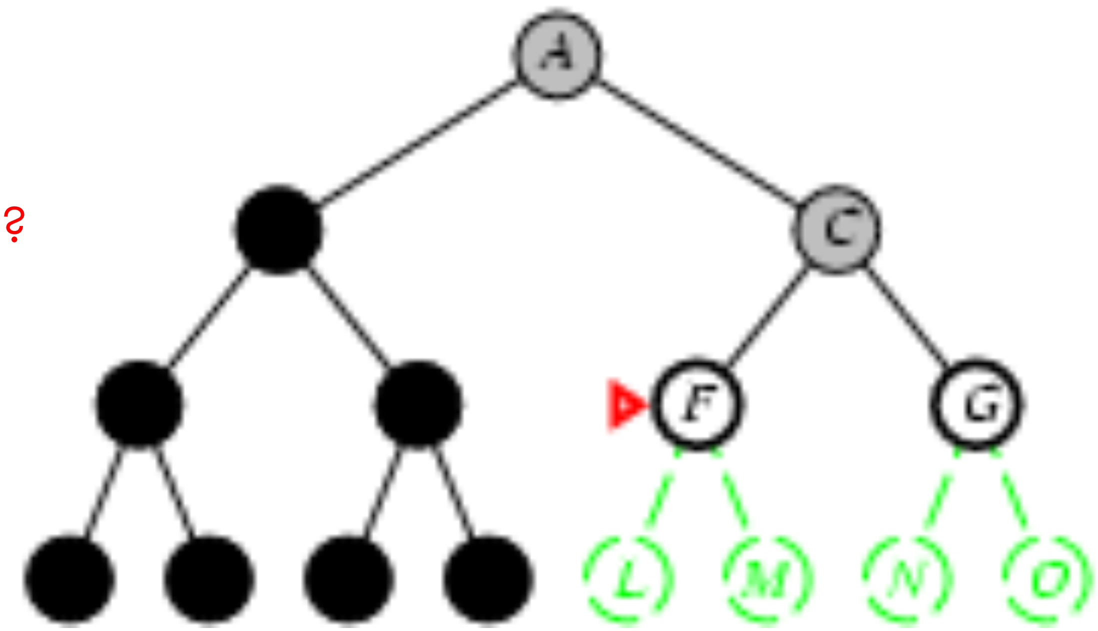


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[F,G]

Is F = goal state?

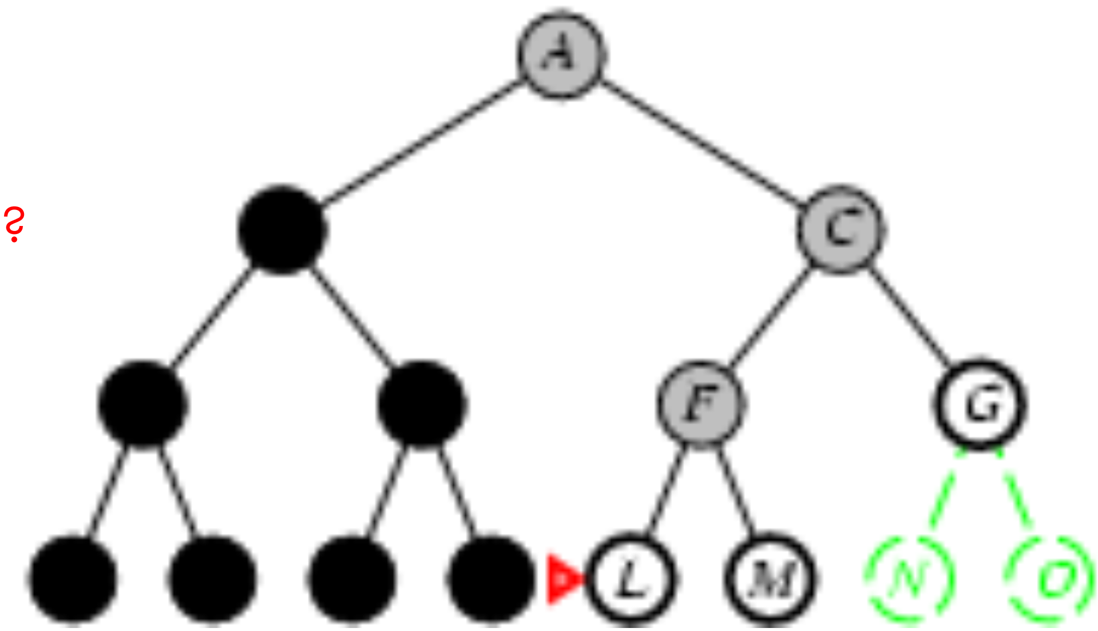


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[L,M,G]

Is L = goal state?

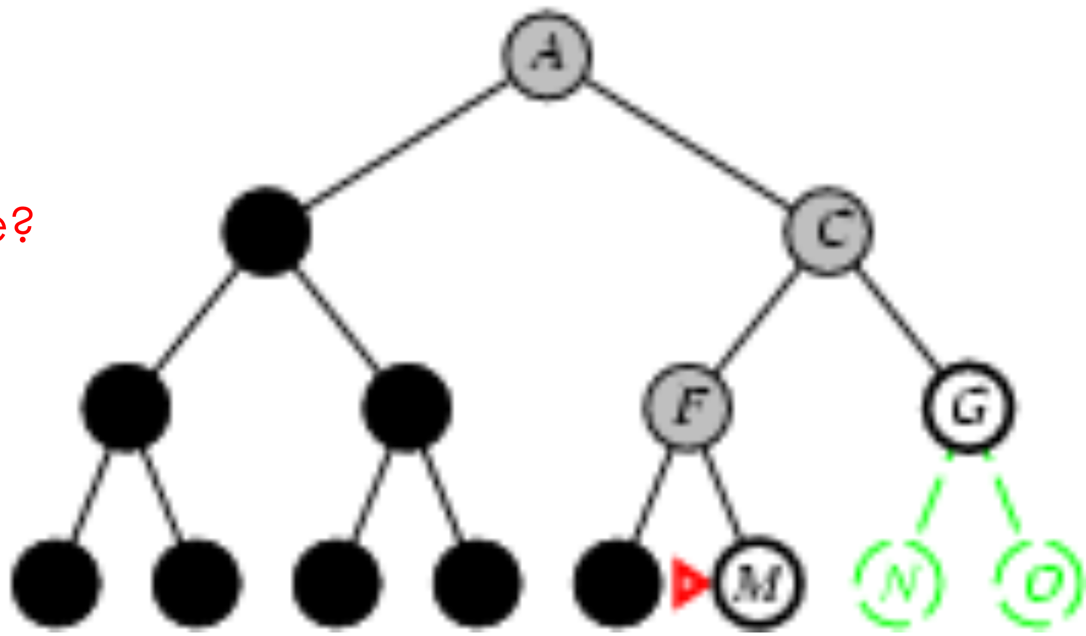


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[M,G]

Is M = goal state?

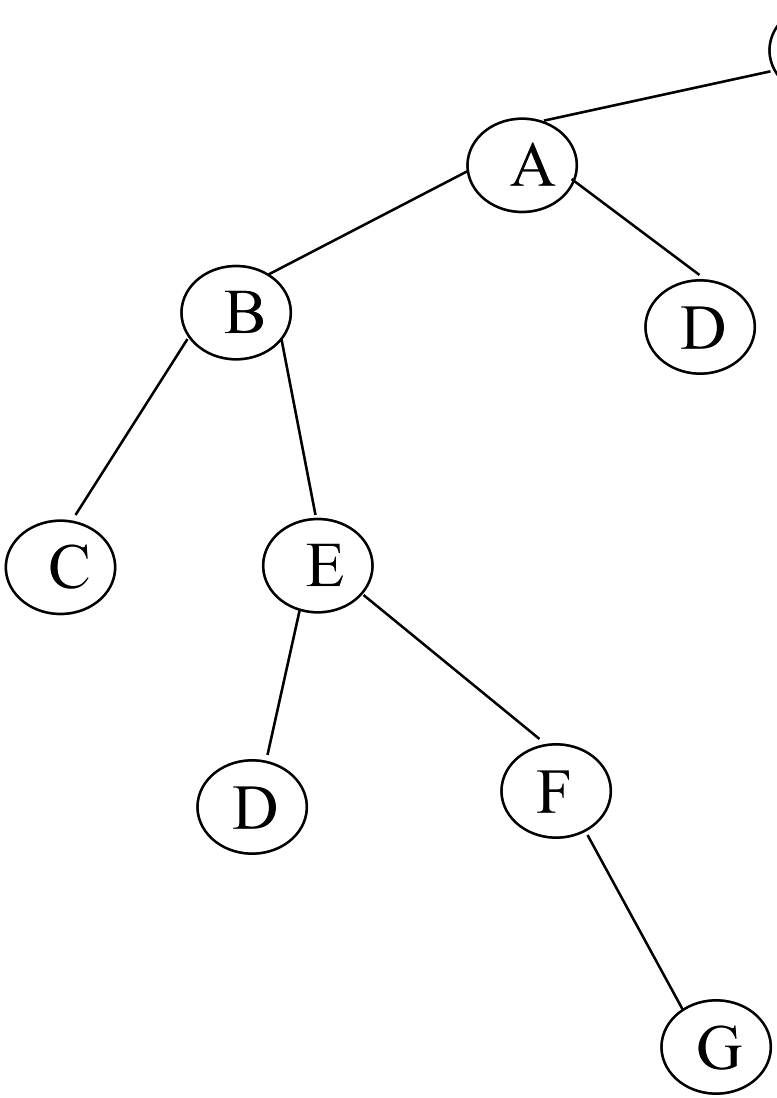


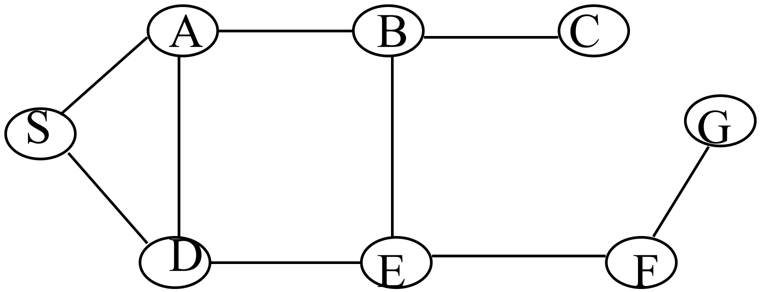
Example DFS

Project 1 – placing amazons

```
def amazonDFS(B, n):
    # B is a vector such that B[i] = the col number in
which the amazon is placed in row i
    # n is the dimension of the board for which a solution
is sought
    # pre-condition: B is a valid partial solution
    if len(B) == n:   # solved
        print(B)
        return True
    for j in range(n):
        # looking for a candidate for B[k] which will be an
integer from 0 to n-1
        if not(attack(B, n, j)):
            B.append(j)
            if amazonDFS(B, n):
                return True
            else:
                B.remove(j)
    print(B)
    return False
```

Depth First Search tree – Graph search version





Here, to avoid repeated states assume we don't expand any child node which appears already in the path from the root S to the parent. (Again, one could use other strategies)

What is the Complexity of Depth-First Search?

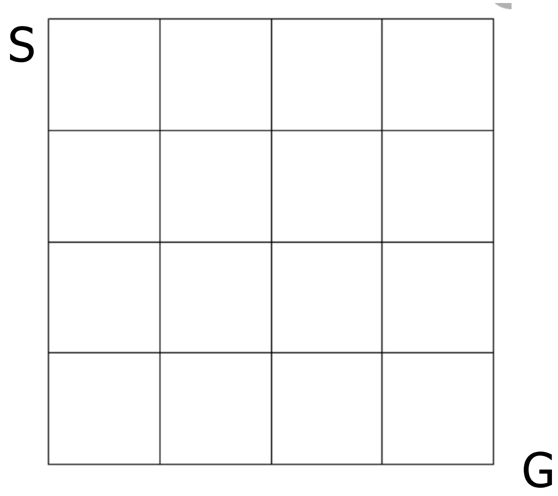
- Time Complexity
  - assume (worst case) that there is 1 goal leaf at the RHS
  - DFS will expand all nodes

    $= 1 + b + b^2 + \ \ldots \ + b^d$

    $= O\ (\mathbf{b^d})$
- Space Complexity (iterative version)
  - how many nodes can be in the stack (worst-case)?
  - at depth $l < d$ we have $b - 1$ nodes
  - at depth d we have b nodes
  - total $= (d-1)*(b-1) + b =$ **$O(bd)$**
- In recursive version, the actual nodes being saved in recursive call stack so implicitly $O(bd)$ nodes are kept.

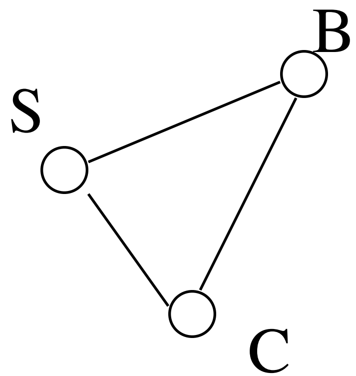
Repeated states

- Failure to detect repeated states can make the search tree much larger than N = size of the size space. (Also true in the case of BFS).

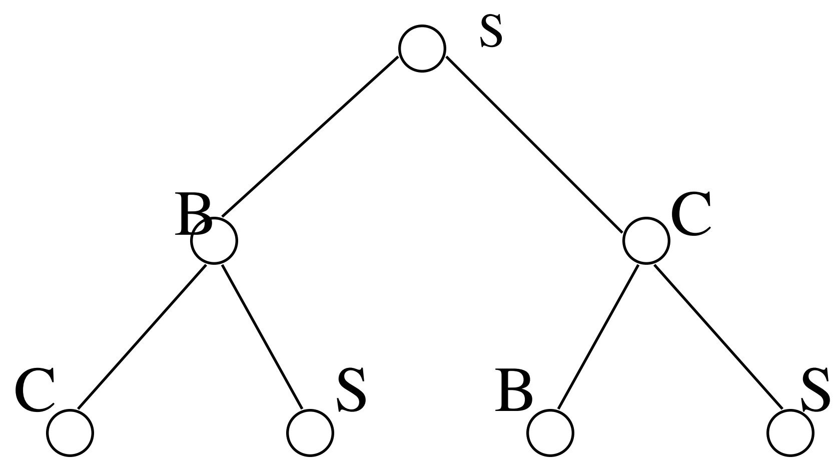


Question: How many times will a node appear in the DFS tree?

Solutions to repeated states



State Space



Example of a Search Tree

- Method 1
  - do not create paths containing cycles (loops)
- Method 2
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states

- Method 1 is most practical, work well on most problems

Properties of depth-first search

- Complete? No.
  - If state space is not finite, the search may never terminate even if solution 2 is used.
- Time? $O(b^m)$ with m = maximum depth
- terrible if m is much larger than d
  - but if there are many solutions, DFS can be much faster than BFS.
- Space? $O(bm)$, i.e., linear space! (we only need to remember a single path + expanded unexplored nodes)
- Optimal? No (It may find a non-optimal goal first)

Comparing DFS and BFS

- Same worst-case time Complexity, but
  - In the worst-case BFS is always better than DFS
  - Sometime, on the average DFS is better if:
    - many goals, no loops and no infinite paths
- BFS is much worse memory-wise
    - DFS is linear space
    - BFS may store the whole search space.
- In general
    - BFS is better if goal is not deep, if infinite paths, if many loops, if small search space
    - DFS is better if many goals, not many loops,
    - DFS is much better in terms of memory

Iterative Deepening (DFS)

- Every iteration is a DFS with a depth cutoff.

Iterative deepening (ID)

1. i = 1
2. While no solution do

   DFS from initial state S_0 with cutoff I

   If found goal, stop and return solution

   else, increment cutoff

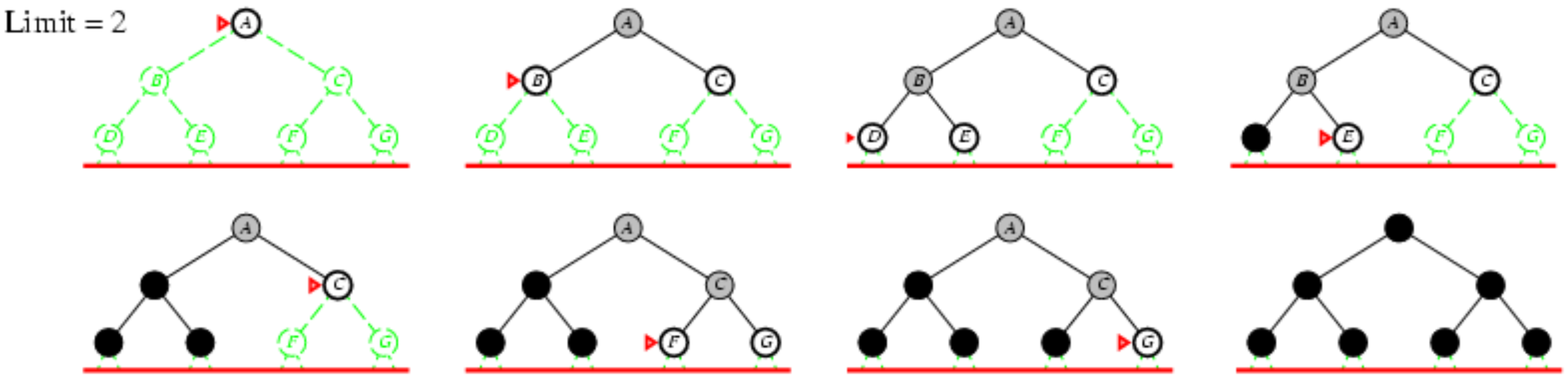
Comments:

- ID implements BFS with DFS
- Only one path in memory
- So it combines the better features of BFS and DFS at a slightly higher cost than

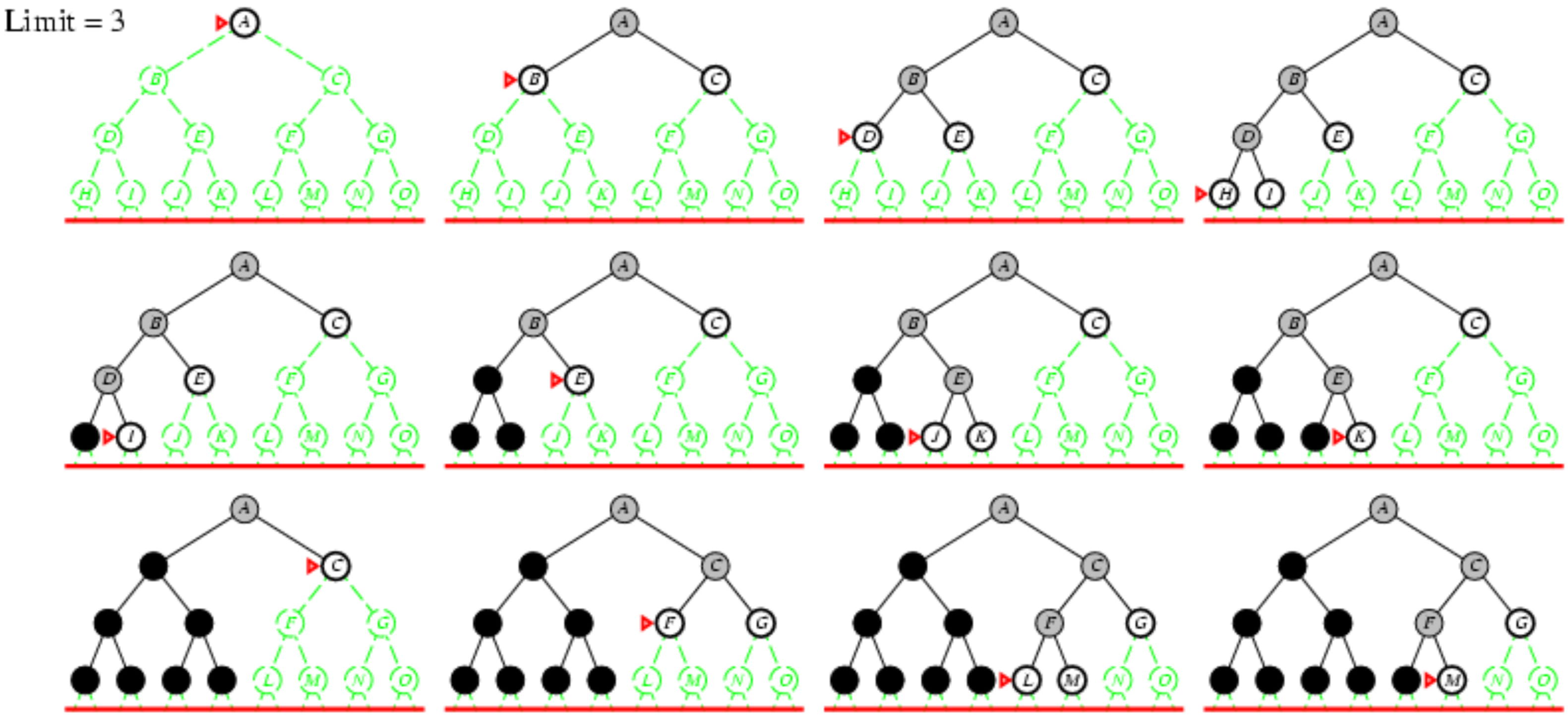
Iterative deepening search L=0

Iterative deepening search *L*=1

Iterative deepening search *L*=2



Iterative Deepening Search *L*=3



Iterative deepening search

Properties of iterative deepening search

- Complete? Yes
- Time? $O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1 or increasing function of depth.

Iterative Deepening Time (DFS)

- Time:

  - BFS time is $O(b^n)$
  - b is the branching degree
  - ID is asymptotically like BFS
  - For b=10 d=5 d=cut-off
  - DFS = 1+10+100,…,=111,111
  - IDS = 123,456

Comments on Iterative Deepening Search

- Complexity
  - Space complexity = O(bd)
    - (since its like depth first search run different times)
  - Time Complexity
    - $1 + (1+b) + (1 + b + b^2) + \ldots\ldots.(1 + b + \ldots.b^d)$
    - $= O(b^d)$
      (i.e., asymptotically the same as BFS or DFS in the worst case)
    - The overhead in repeated searching of the same subtrees is small relative to the overall time
      - e.g., for b=10, only takes about 11% more time than BFS
- A useful practical method
  - combines
    - guarantee of finding an optimal solution if one exists (as in BFS)
    - space efficiency, O(bd) of DFS
    - But still has problems with loops like DFS

Bidirectional Search

- Idea
  - Simultaneously search forward from S and backwards from G
  - stop when both "meet in the middle"
  - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult
    - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?
- Complexity
  - time complexity is best: $O(2\ b^{(d/2)}) = O(b^{(d/2)})$, worst: $O(b^{d+1})$
  - memory complexity is the same

Weighted edge case: Uniform Cost Search

- Expand lowest-cost OPEN node ($g(n)$)
- In BFS $g(n) = depth(n)$

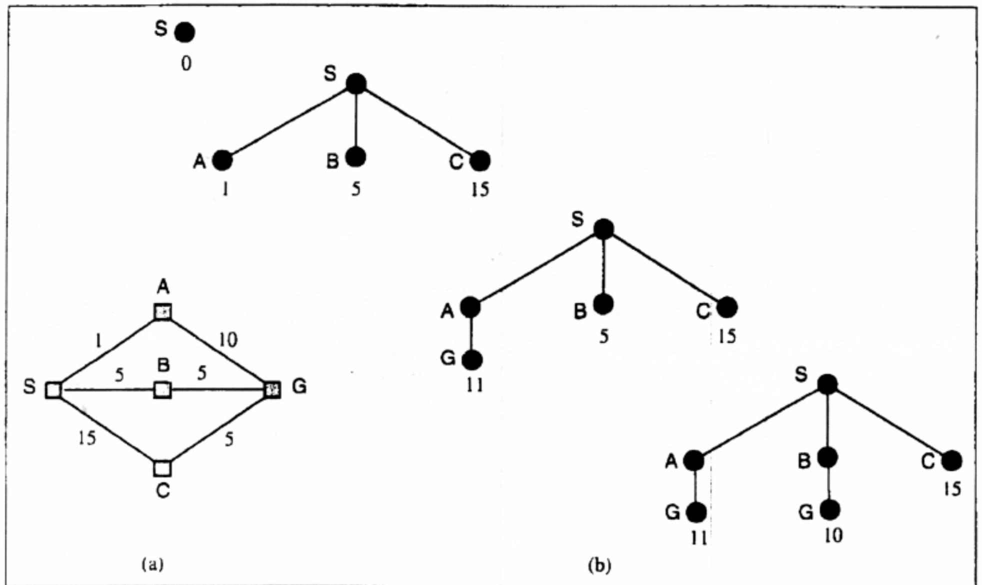


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

- Requirement
  - $g(\text{successor}(n)) \geq g(n)$

Uniform cost search – Tree version

1. Put the start node *s* on OPEN
2. If OPEN is empty exit with failure.
3. Remove the first node *n* from OPEN and place it on CLOSED.
4. If *n* is a goal node, exit successfully with the solution obtained by tracing back pointers from *n* to *s*.
5. Otherwise, expand *n*, generating all its successors attach to them pointers back to *n*, and put them at the *end* of OPEN ***in order of shortest cost from the root node***
6. Go to step 2.

Uniform cost search – Graph Version

1. Put the start node *s* on OPEN
2. If OPEN is empty exit with failure.
3. Remove the first node *n* from OPEN and place it on CLOSED.
4. If *n* is a goal node, exit successfully with the solution obtained by tracing back pointers from *n* to *s*.
5. Otherwise, expand *n*, generating all its successors not in CLOSED set, attach to them pointers back to *n*, and put them at the *end* of OPEN ***in order of shortest cost from the root node***
6. Go to step 2.

Uniform-cost search

Implementation: *fringe* = queue ordered by path cost
Equivalent to breadth-first if all step costs all equal.

Complete? Yes, if step cost $\geq \varepsilon$
(otherwise it can get stuck in infinite loops)

Time? # of nodes with *path cost* $\leq$ cost of optimal solution.

Space? # of nodes on paths with path cost $\leq$ cost of optimal solution.

Optimal? Yes, for any step cost.

Comparison of Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Figure 3.18 Evaluation of search strategies. b is the branching factor; d is the depth of solution; m is the maximum depth of the search tree; l is the depth limit.

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

- Variety of uninformed search strategies

- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

Summary

- A review of search
  - a search space consists of states and operators: it is a graph
  - a search tree represents a particular exploration of search space
- There are various strategies for “uninformed search”
  - breadth-first
  - depth-first
  - iterative deepening
  - bidirectional search
  - Uniform cost search
  - Depth-first branch and bound
- Repeated states can lead to infinitely large search trees
  - we looked at methods for for detecting repeated states
- All of the search techniques so far are “blind” in that they do not look at how far away the goal may be: next we will look at informed or heuristic search, which directly tries to minimize the distance to the goal. Example we saw: greedy search