# Search

Dr. Melanie Martin

CS 4480

September 8, 2010

Based on slides from

http://aima.eecs.berkeley.edu/2nd-ed/slides-ppt/

# Search strategies

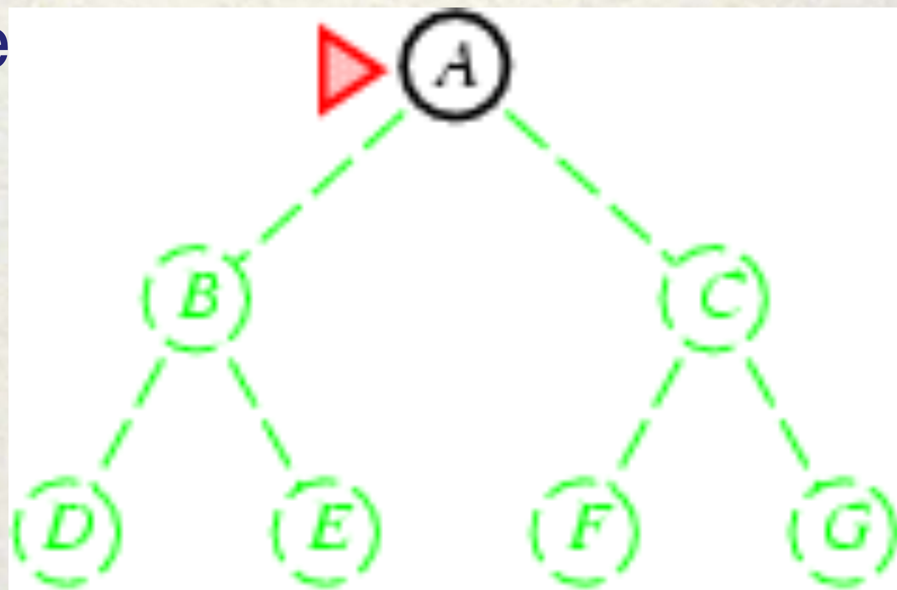
- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

# Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

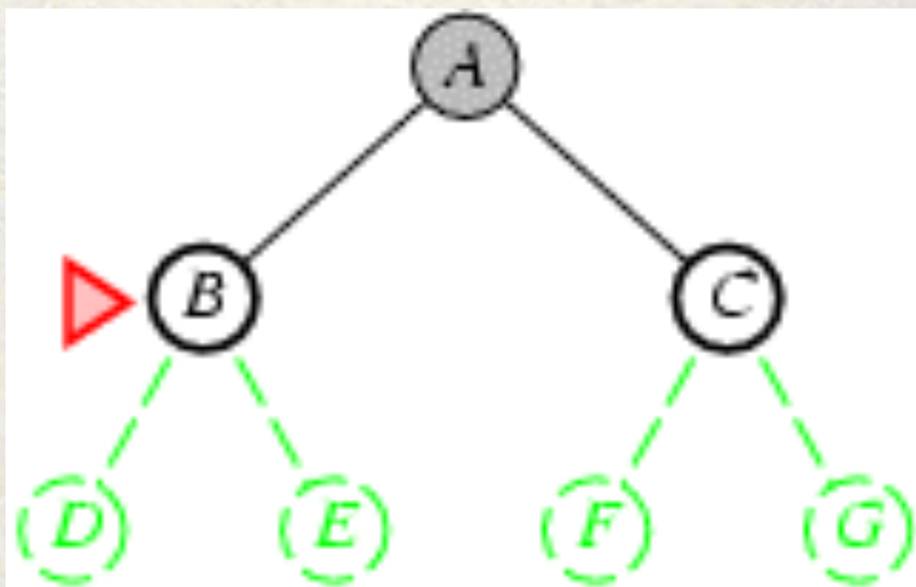
# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new succe



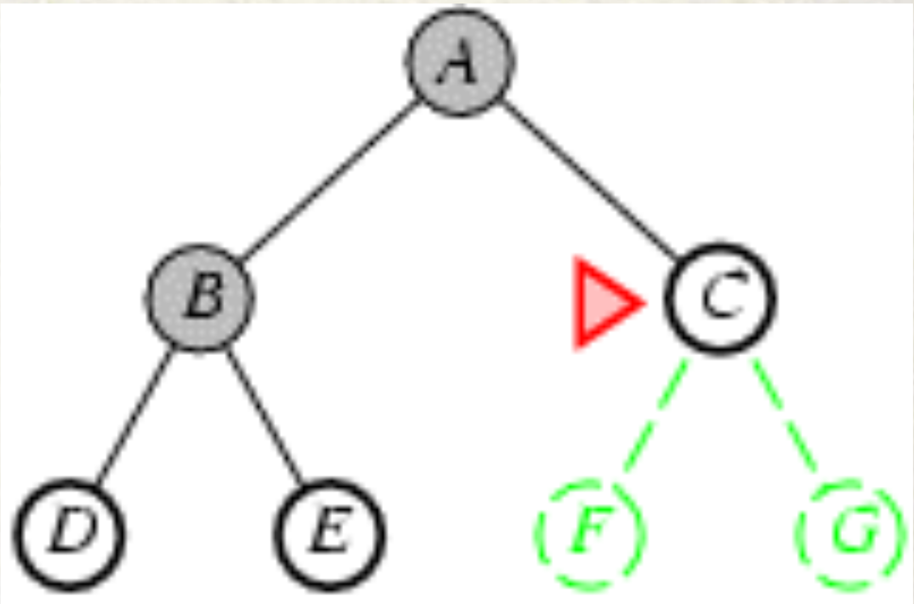
# Breadth-first search

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



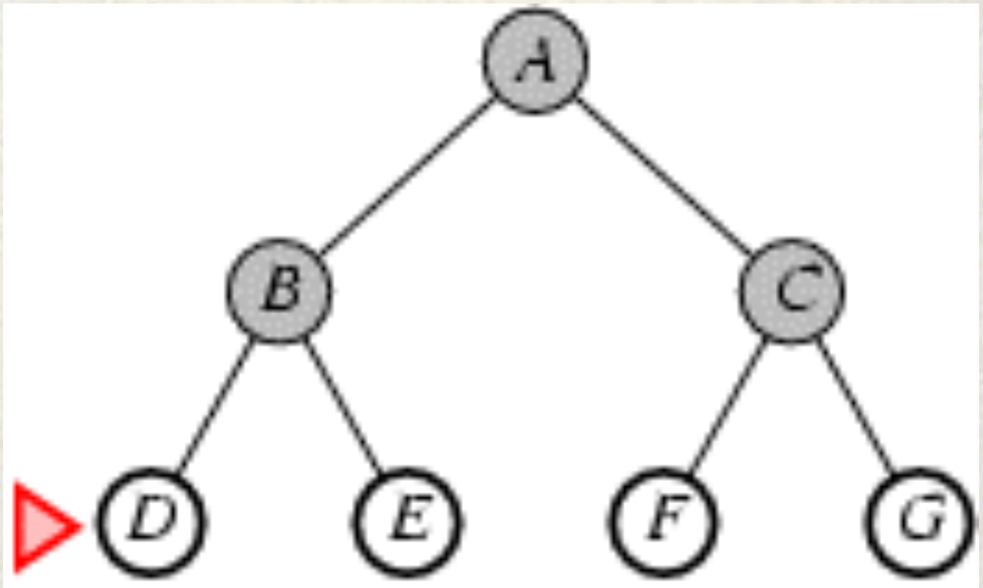
# Breadth-first search

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



# Breadth-first search

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



# Properties of breadth-first search

- Complete? Yes (if $b$ is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

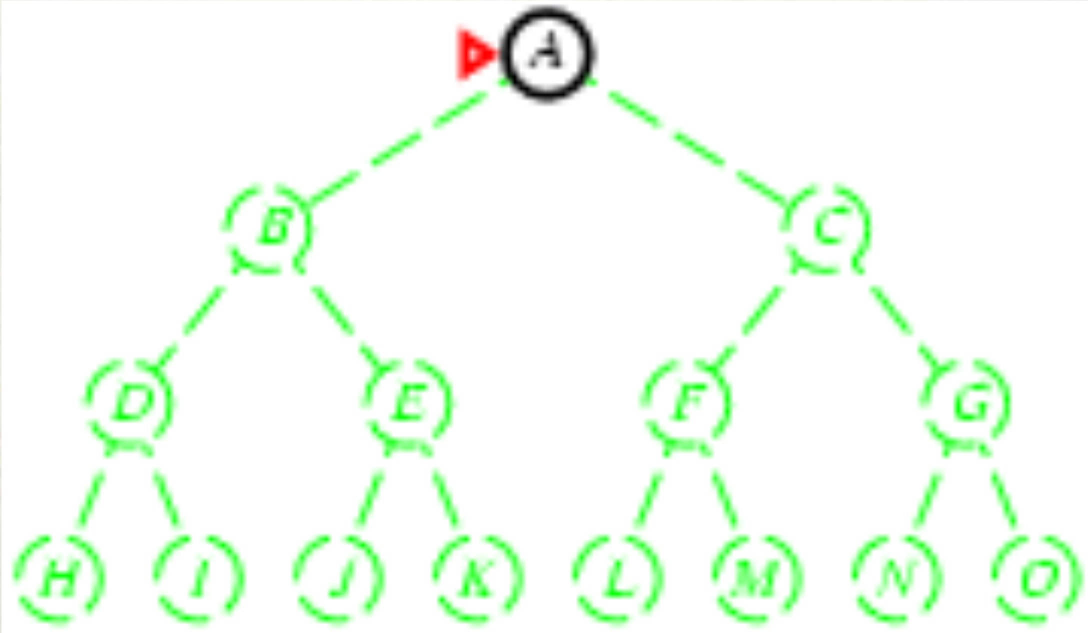
- Space is the bigger problem (more than time)

# Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with *g* ≤ cost of optimal solution, $O(b^{ceiling(C^*/ \varepsilon)})$ where $C^*$ is the cost of the optimal solution
- Space? # of nodes with *g* ≤ cost of optimal solution, $O(b^{ceiling(C^*/ \varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of *g(n)*

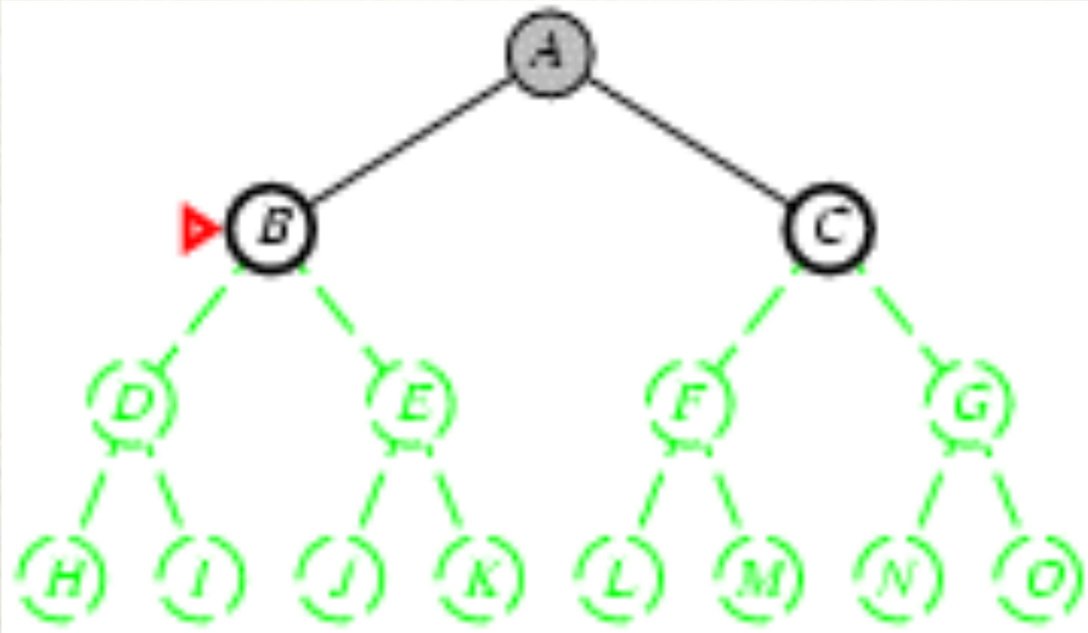
# Depth-first search

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



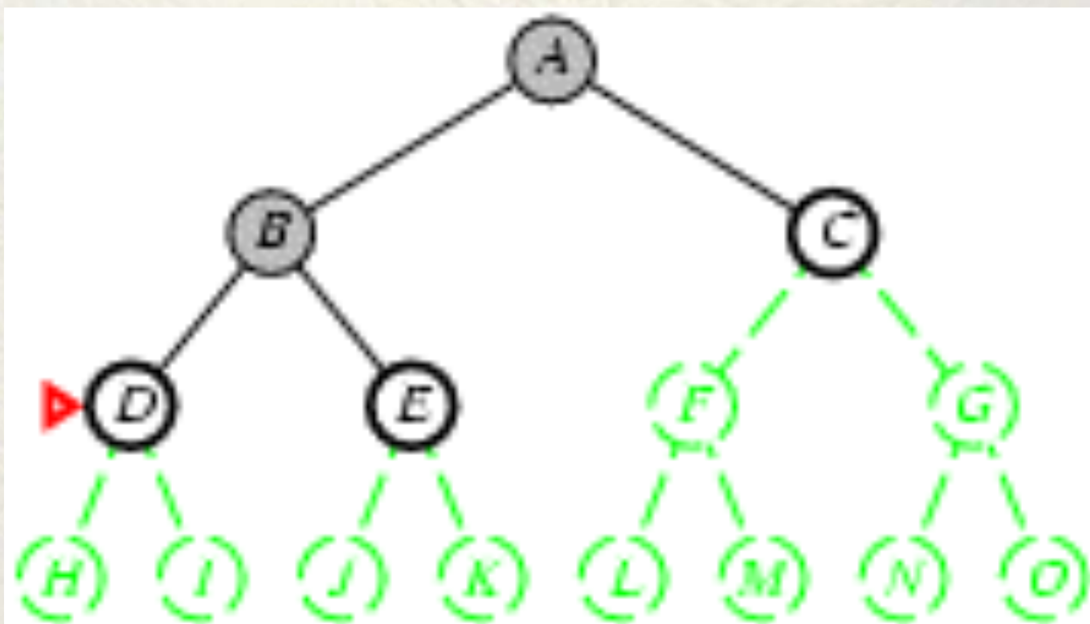
# Depth-first search

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



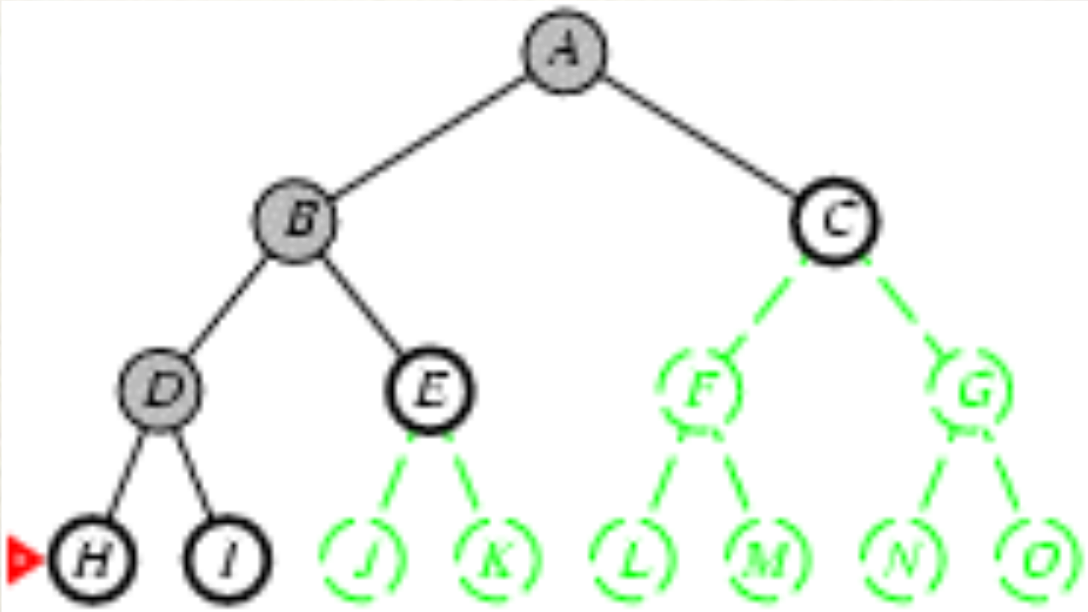
# Depth-first search

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



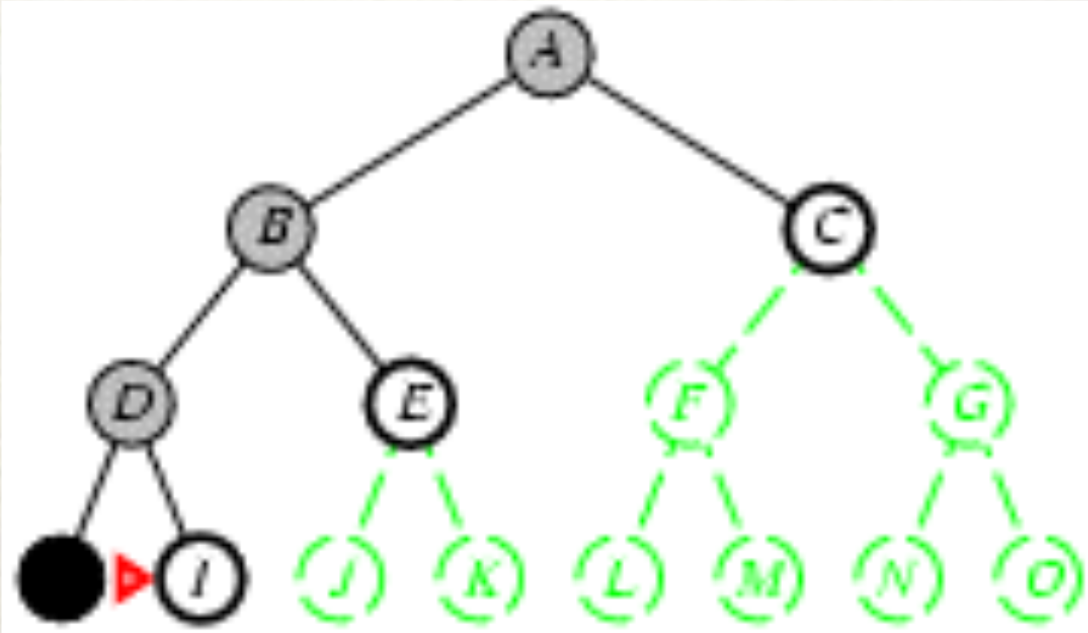
# Depth-first search

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



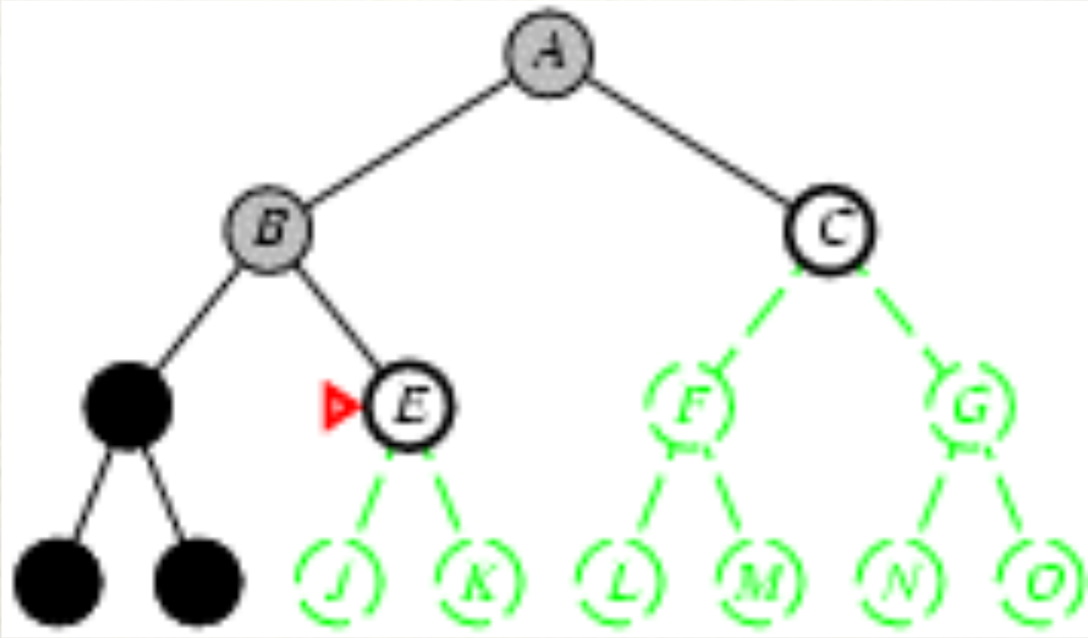
# Depth-first search

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



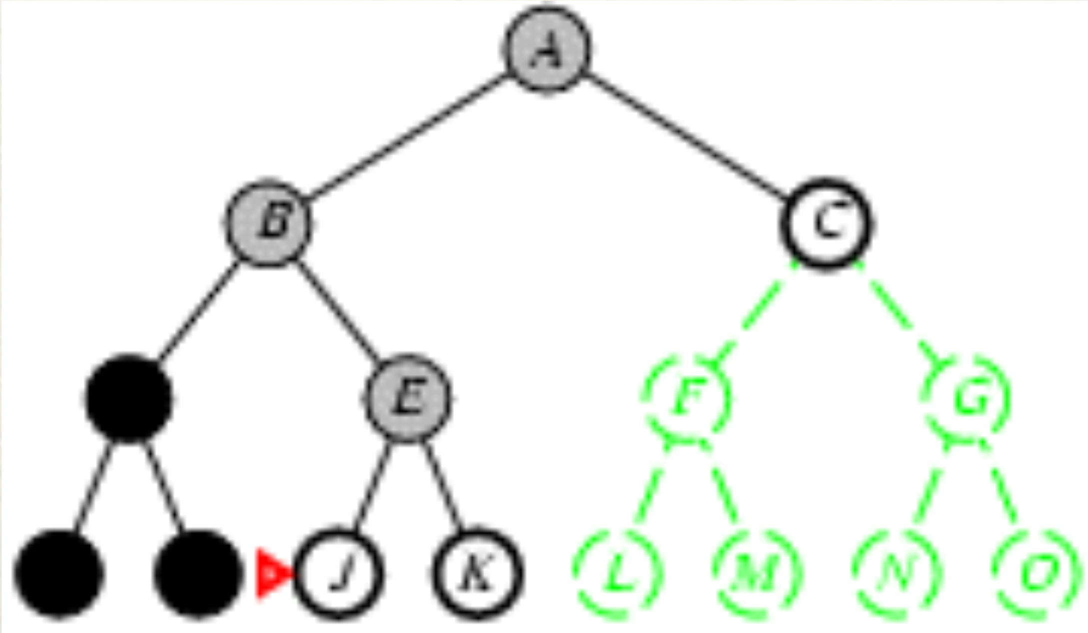
# Depth-first search

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



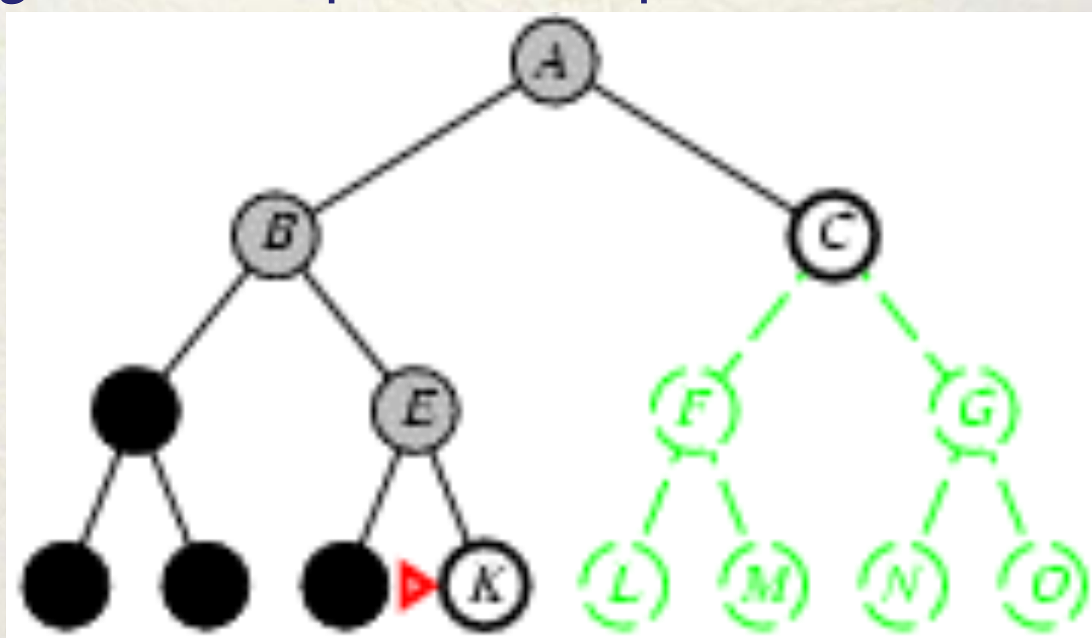
# Depth-first search

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



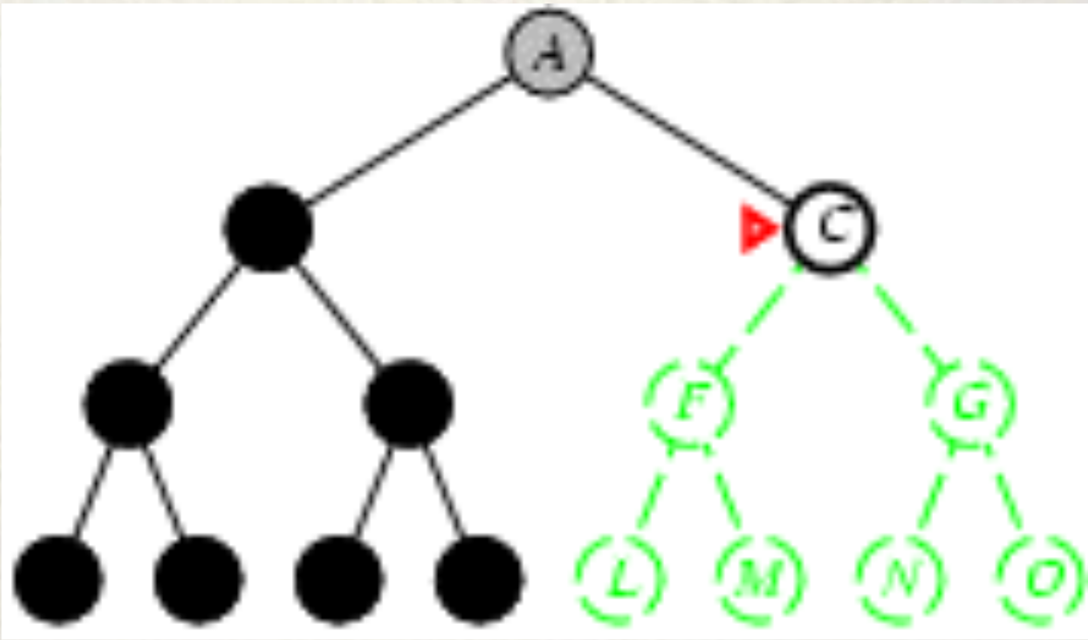
# Depth-first search

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



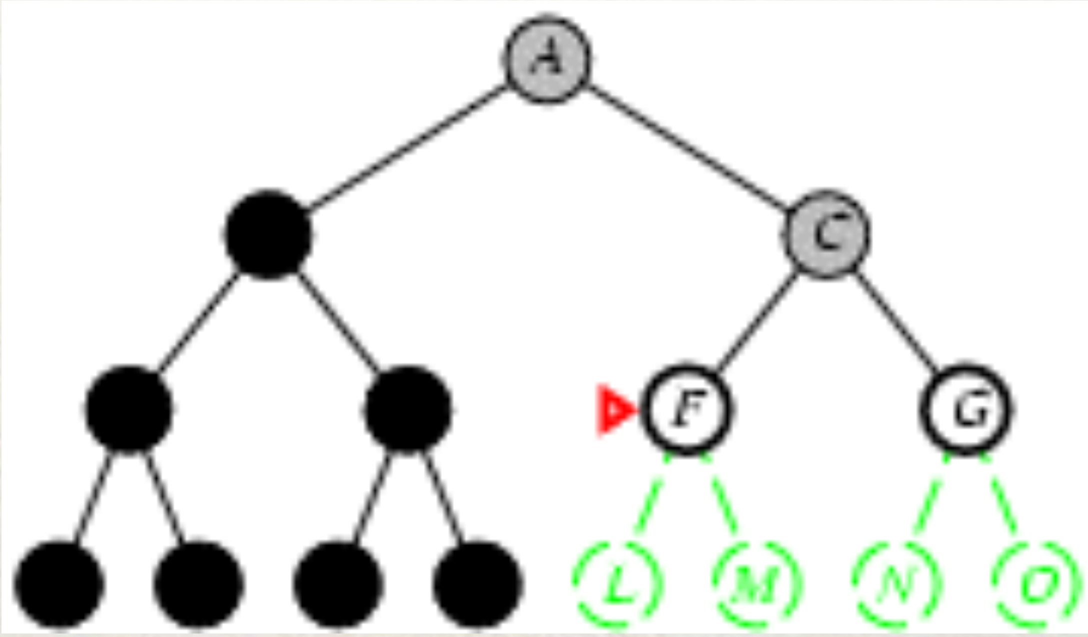
# Depth-first search

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



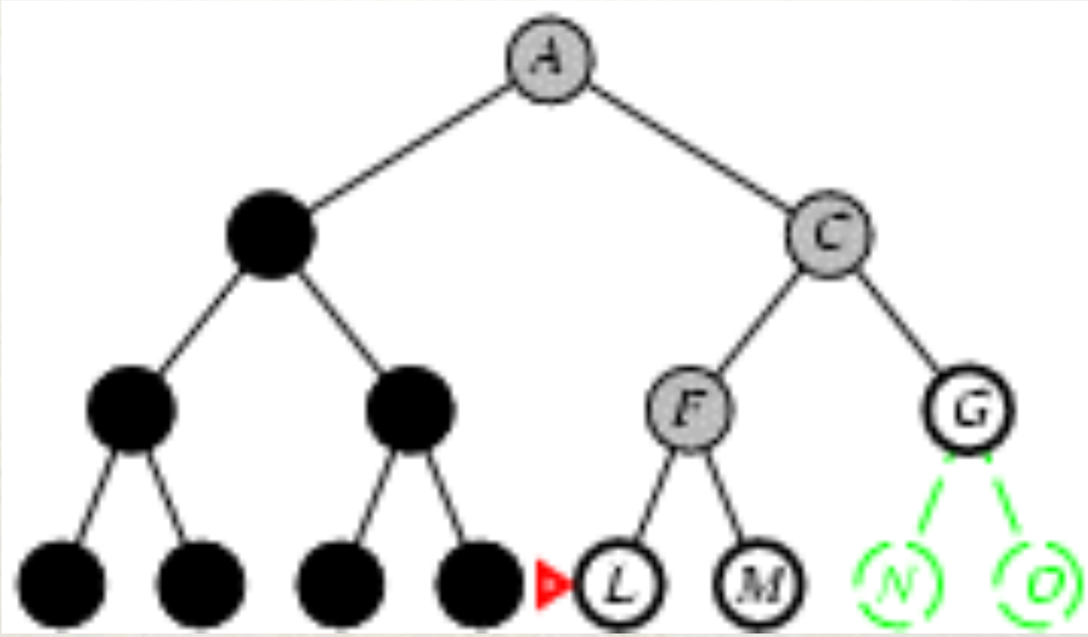
# Depth-first search

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



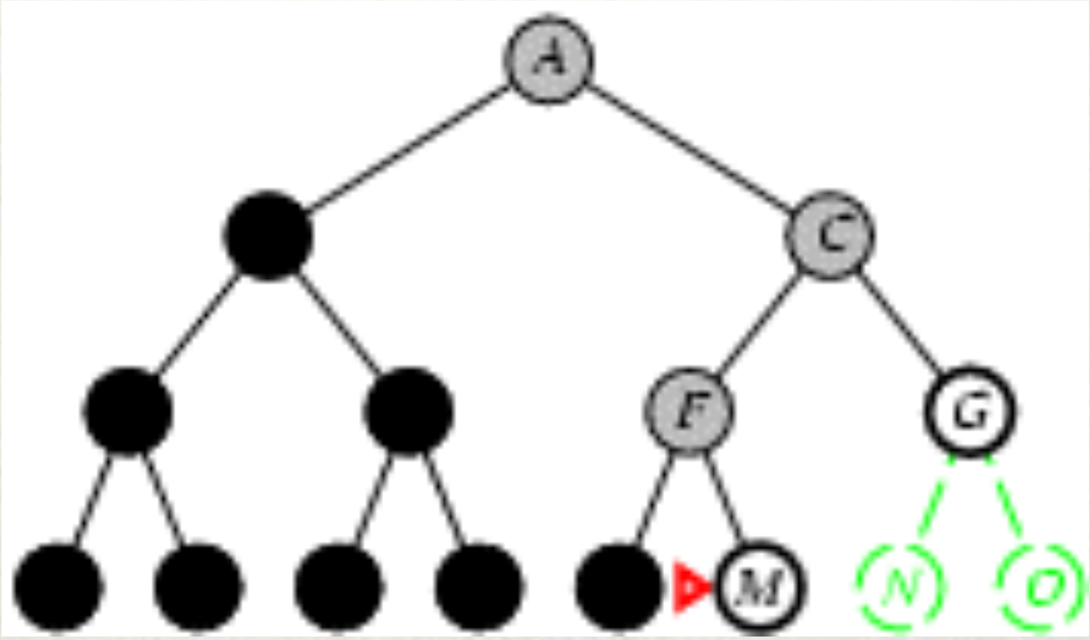
# Depth-first search

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



# Depth-first search

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



# Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    - → complete in finite spaces
- Time? $O(b^m)$: terrible if $m$ is much larger than $d$
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

# Depth-limited search

= depth-first search with depth limit *l*,

i.e., nodes at depth *l* have no successors

- 

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

# Iterative deepening search

**function** ITERATIVE-DEEPENING-SEARCH( *problem*) **returns** a solution, or failure

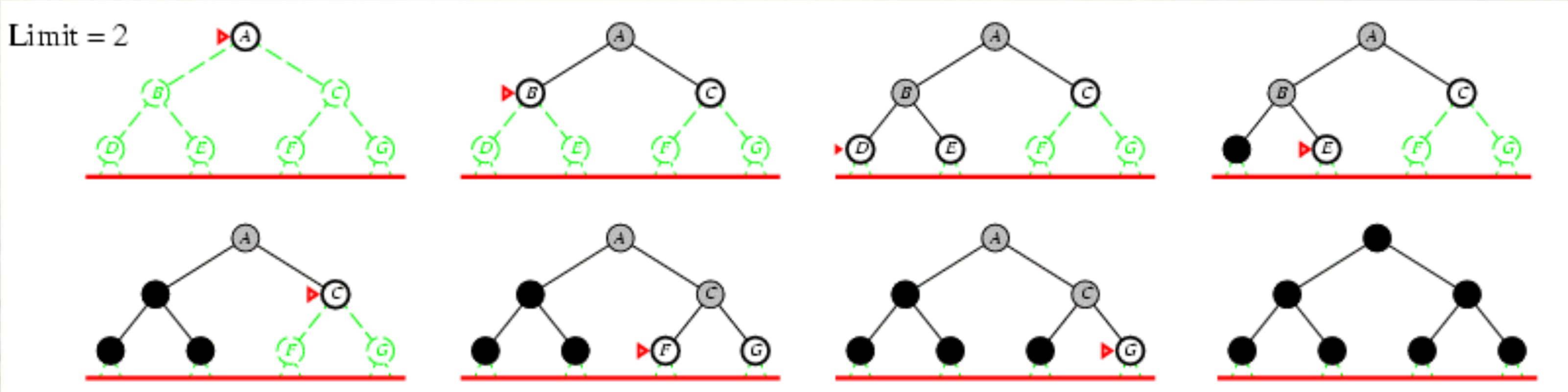
**inputs**: *problem*, a problem

**for** *depth* ← 0 **to** ∞ **do**
  *result* ← DEPTH-LIMITED-SEARCH( *problem*, *depth*)
  **if** *result* ≠ cutoff **then return** *result*

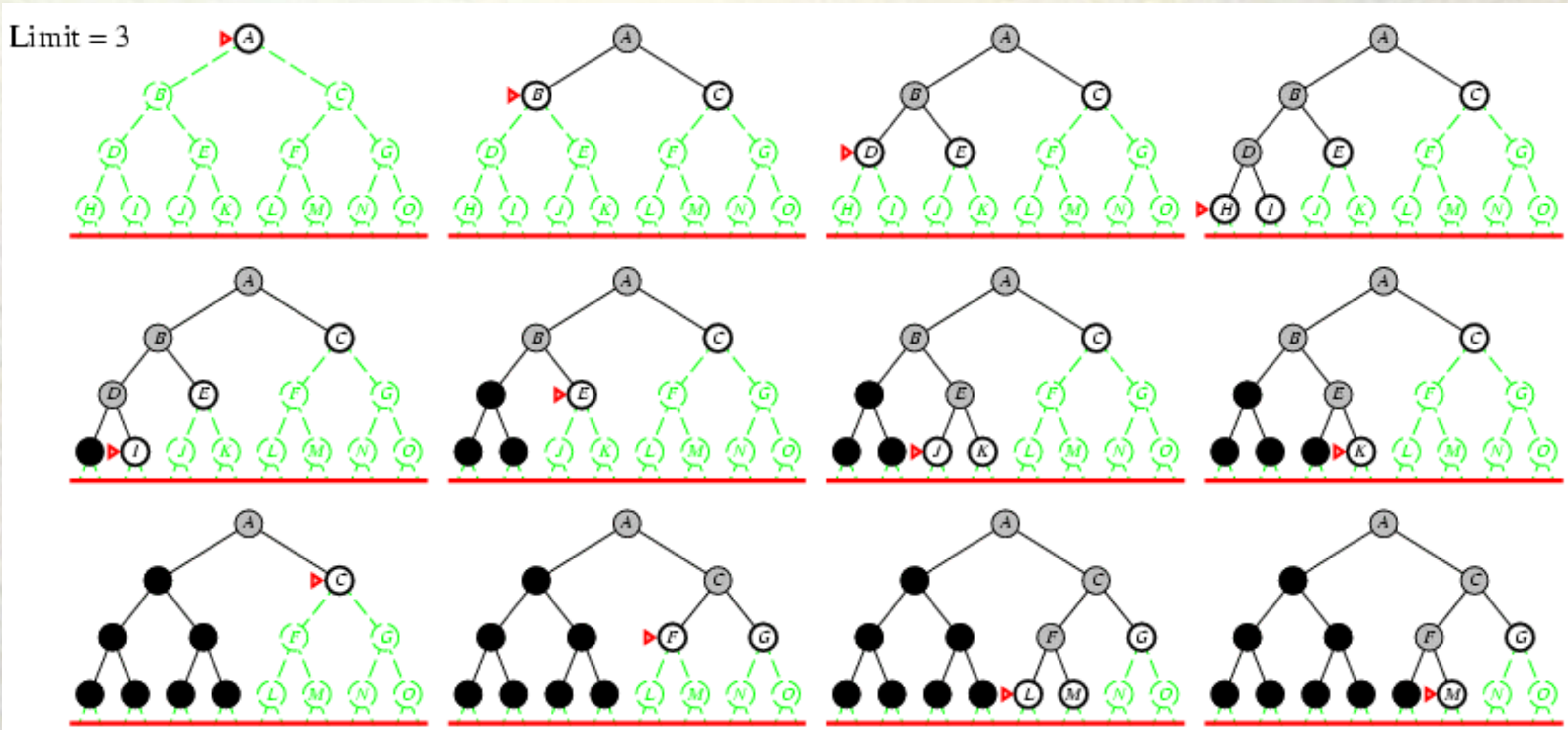
# Iterative deepening search *l* =0

# Iterative deepening search *l* =1

# Iterative deepening search *l* =2



# Iterative deepening search *l* =3



# Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\ b\text{^}^1 + (d-1)b\text{^}^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For *b = 10*, *d = 5*,
  - $N_{DLS}$ = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  - $N_{IDS}$ = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

- Overhead = (123,456 - 111,111)/111,111 = 11%

# Properties of iterative deepening search

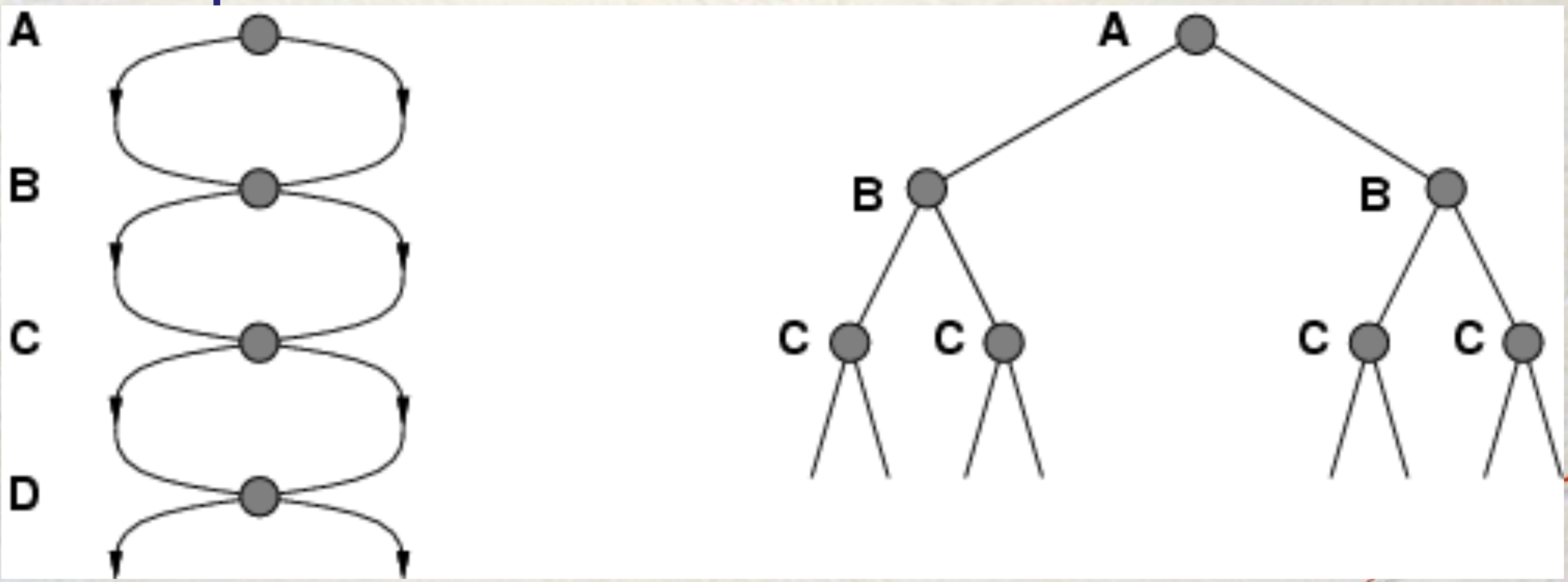
- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

# Summary of algorithms

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening |
|---|---|---|---|---|---|
| Complete? | Yes | Yes | No | No | Yes |
| Time | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(b^m)$ | $O(b^l)$ | $O(b^d)$ |
| Space | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(bm)$ | $O(bl)$ | $O(bd)$ |
| Optimal? | Yes | Yes | No | No | Yes |

# Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!



# Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

# 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

# Coming Next

- Homework: 3.7 a, c; 3.8; 3.9
- Getting book lisp code running
- Reading Chapter 4 in R&N