CS:4420 Artificial Intelligence

Spring 2019

Uninformed Search

Cesare Tinelli

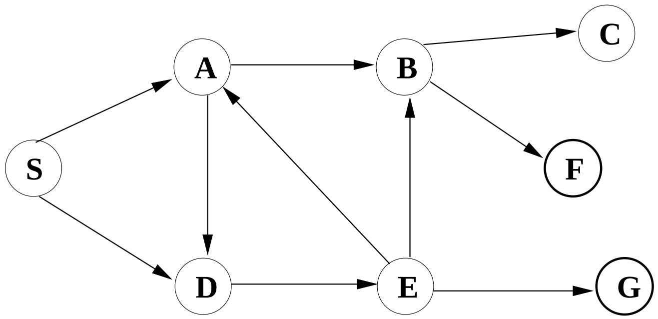
The University of Iowa

Readings

- Chap. 3 of [Russell and Norvig, 3rd edition]

More on Graphs

A *graph* is a set of *nodes* and *edges* between them

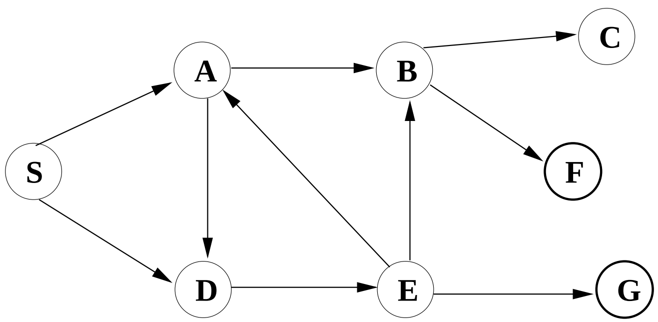


A graph is *directed* if its edges can be traversed only in a specified direction

When an edge is directed from n_i to n_j

- it is uniquely identified by the pair (n_i, n_j)
- n_i is a parent (or *predecessor*) of n_j
- n_j is a *child* (or *successor*) of n_i

Directed Graphs



A *path*, of length $k \geq 0$, is a sequence $\langle (n_1, n_2), (n_2, n_3), \ldots, (n_k, n_{k+1}) \rangle$ of k *successive* edges [a]

Ex: $\langle \rangle$, $\langle (S, D) \rangle$, $\langle (S, D), (D, E), (E, B) \rangle$

For $1 \leq i < j \leq k + 1$,

- N_i is a *ancestor* of N_j; N_j is a *descendant* of N_i

A graph is *cyclic* if it has a path starting and ending with the same node. *Ex:* $\langle (A, D), (D, E), (E, A) \rangle$

[a] Note that a path of length $k > 0$ contains $k + 1$ nodes

From Search Graphs to Search Trees

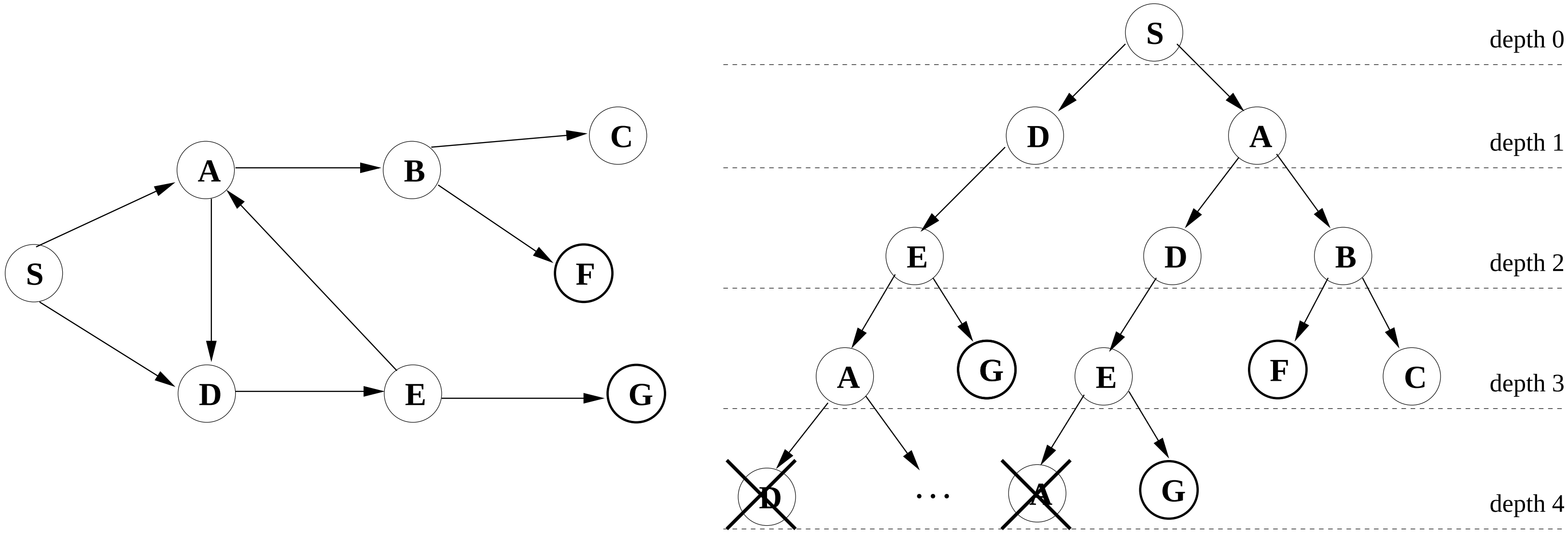
The set of all possible paths of a graph can be represented as a tree

- A *tree* is a directed acyclic graph all of whose nodes have at most one parent
- A *root* of a tree is a node with no parents
- A *leaf* is a node with no children
- The *branching factor* of a node is the number of its children

Graphs can be turned into trees by duplicating nodes and breaking cyclic paths, if any

From Graphs to Trees

To unravel a graph into a tree choose a root node and trace every path from that node until you reach a leaf node or a node already in that path



Note:

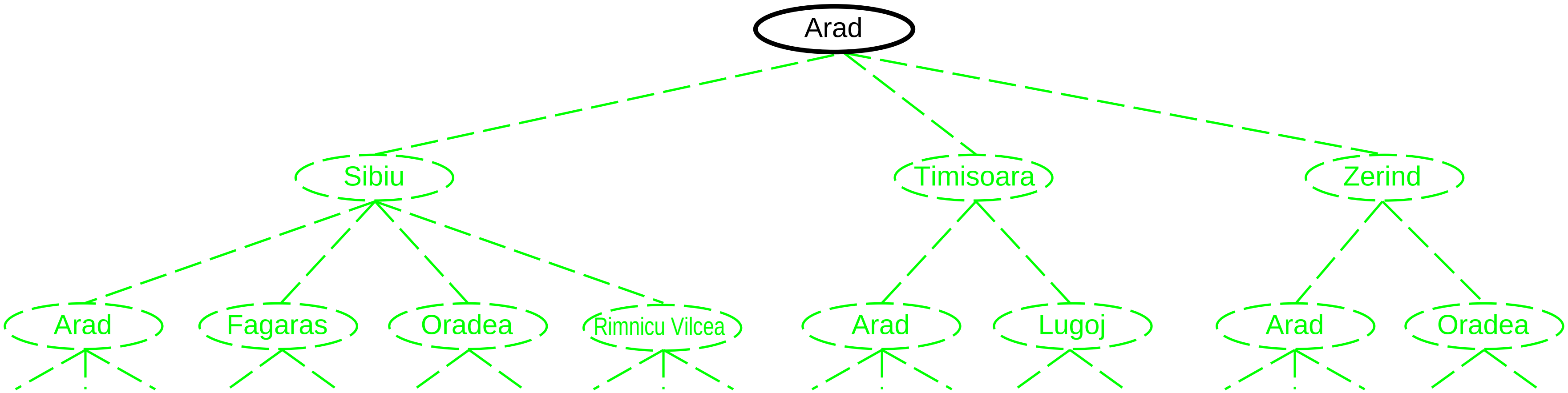
- must remember which nodes have been visited
- a node may get duplicated several times in the tree
- the tree has infinite paths if and only if the graph has infinite non-cyclic paths

Tree Search Algorithms

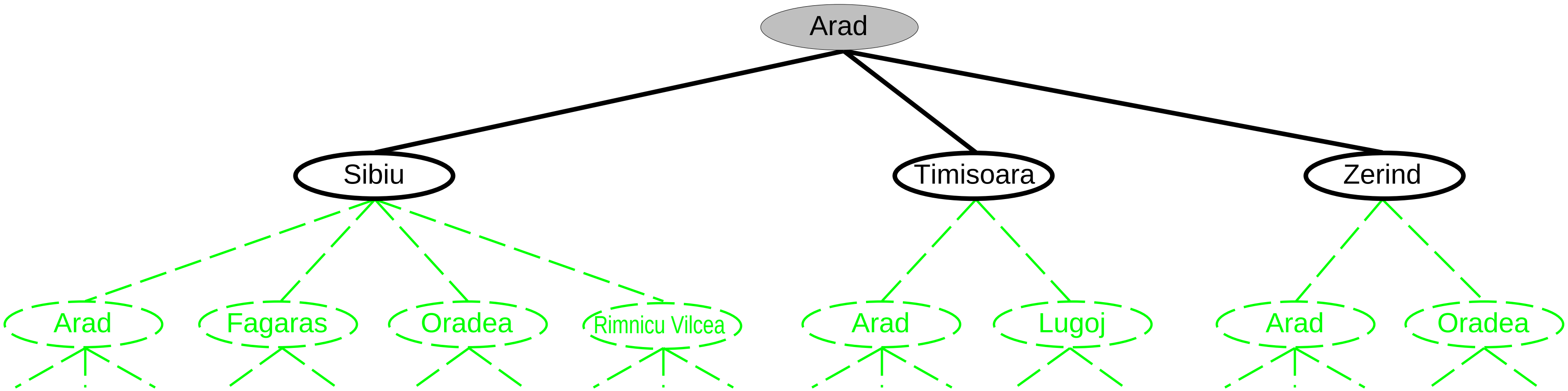
Basic Idea: offline, simulated exploration of state space by generating successors of already-explored states

```
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
        else
            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 its successors to the tree
    done
```

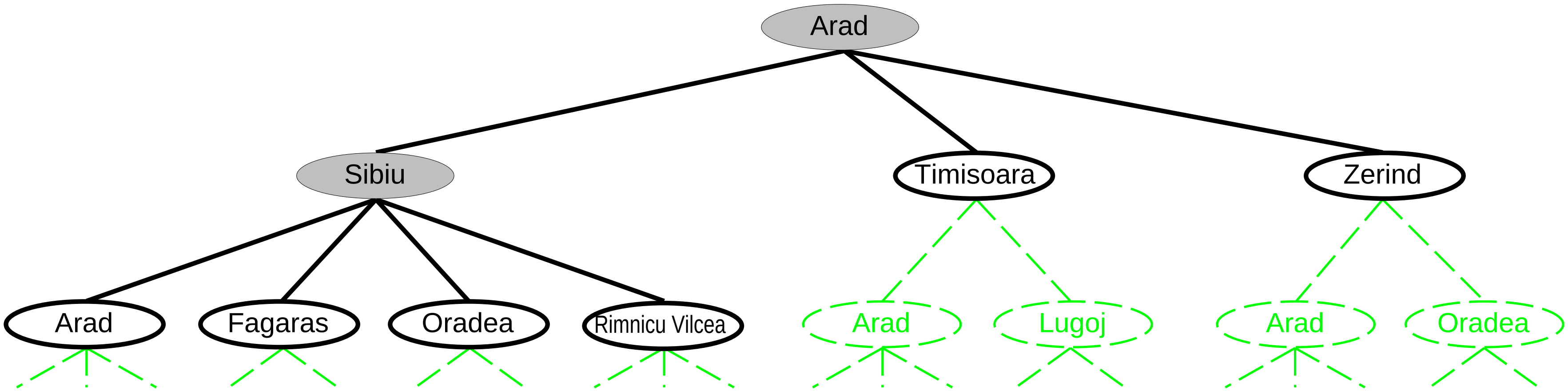
Tree Search Example



Tree Search Example

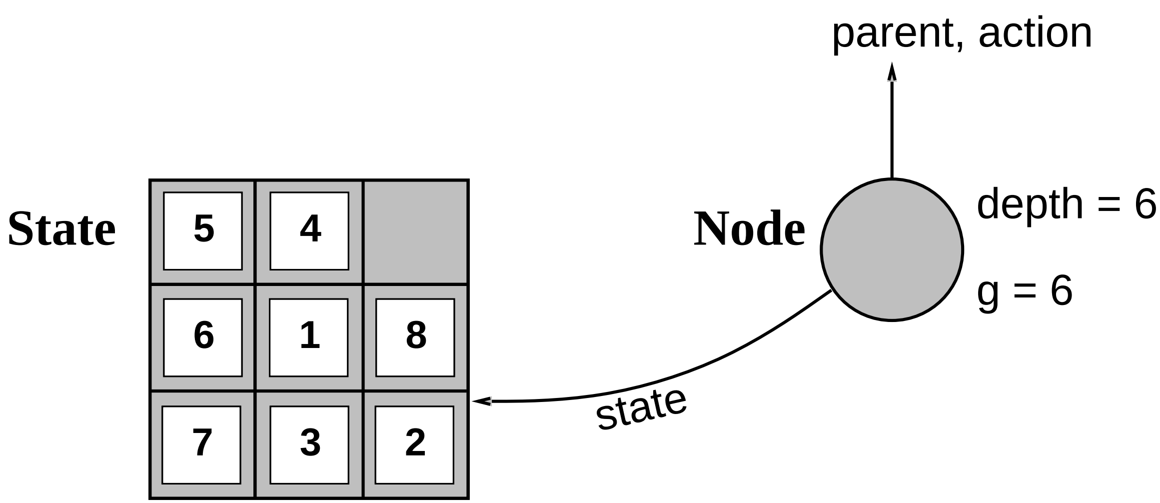


Tree Search Example



Implementation: states vs. nodes

- A *state* is a (representation of) a physical configuration
- A *node* is a data structure constituting part of a search tree (and includes such info as parent, children, depth, *path cost* $g(x)$)
- States do not have parents, children, depth, or path cost!



Search Strategies

A strategy is defined by picking the order of node expansion

Strategies are evaluated along the following dimensions:

- *solution completeness*: does it always find a solution if one exists?
- *time complexity*: number of nodes generated/expanded
- *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 ∞)

Search Strategies

Uninformed (or Blind) Search Strategies

- Little or no information about the search space is available
- All we know is how to generate new states and recognize a goal state

Informed (or Heuristic) Search Strategies

- An estimate of the number of steps or the path cost from current state to goal state is available
- The estimate is not perfect (otherwise no search is needed!) but can help prune the search space considerably

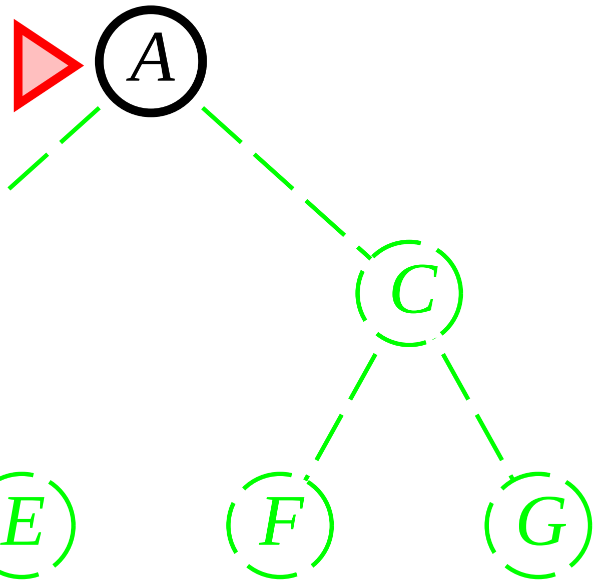
Some Uninformed Search Strategies

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening (depth-first) search

Breadth-First Search

Strategy: Expand shallowest unexpanded node

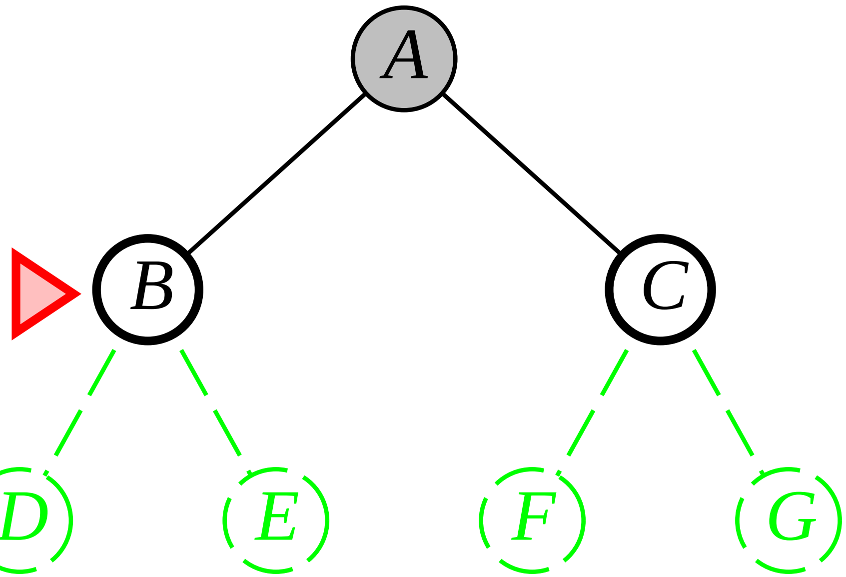
Implementation: the current set of unexpanded nodes, the *fringe (or frontier)*, is processed as FIFO queue, i.e., new successors go at end



Breadth-First Search

Strategy: Expand shallowest unexpanded node

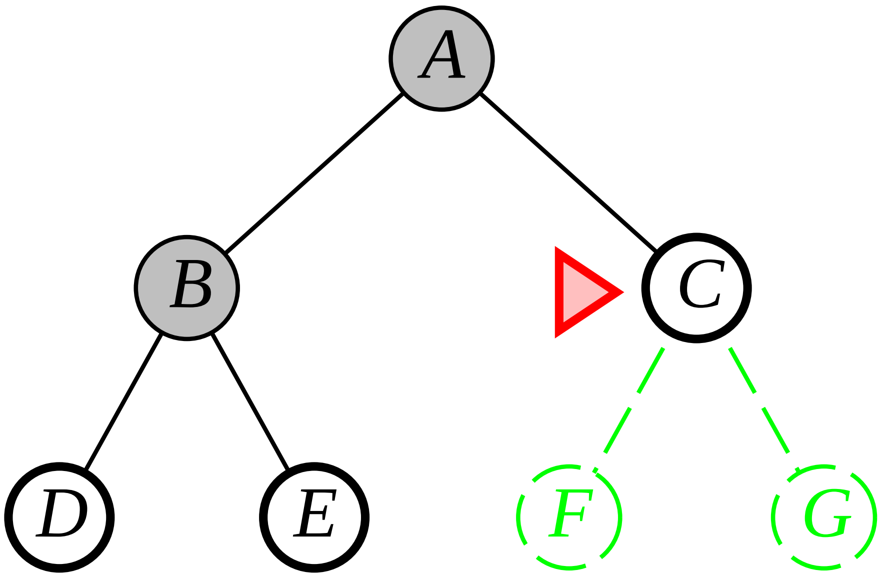
Implementation: the current set of unexpanded nodes, the *fringe (or frontier)*, is processed as FIFO queue, i.e., new successors go at end



Breadth-First Search

Strategy: Expand shallowest unexpanded node

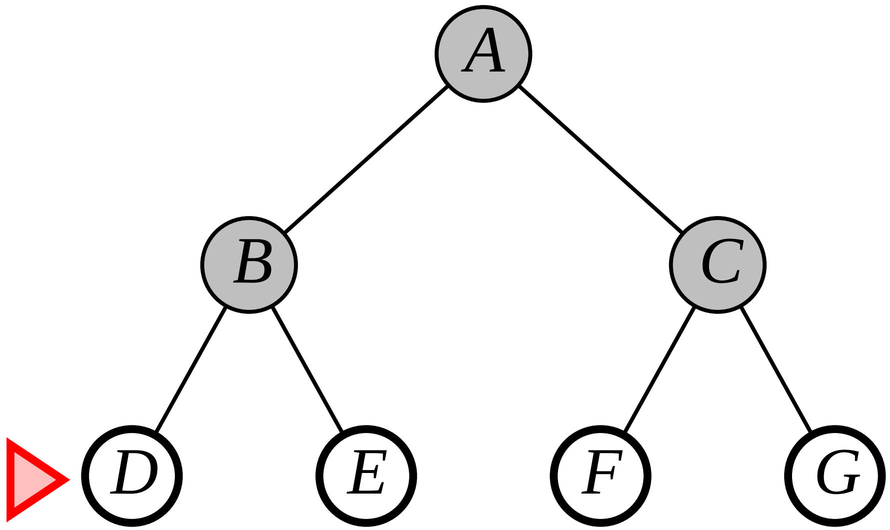
Implementation: the current set of unexpanded nodes, the *fringe (or frontier)*, is processed as FIFO queue, i.e., new successors go at end



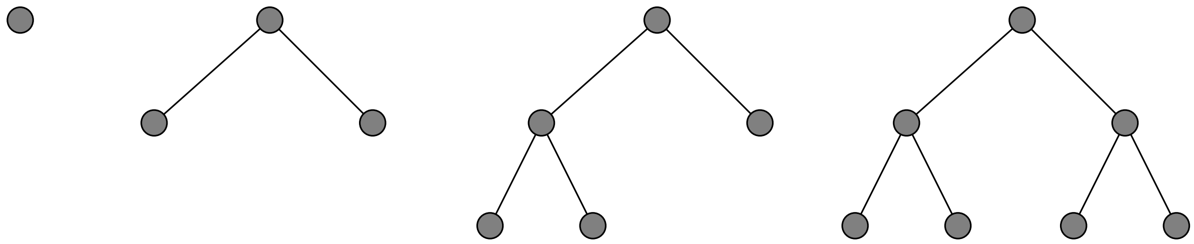
Breadth-First Search

Strategy: Expand shallowest unexpanded node

Implementation: the current set of unexpanded nodes, the *fringe (or frontier)*, is processed as FIFO queue, i.e., new successors go at end



Cost of Breadth-First Search



Worst-case Time Complexity (no. of node expansions)

All nodes must be expanded to find a goal state. We must process these many nodes:

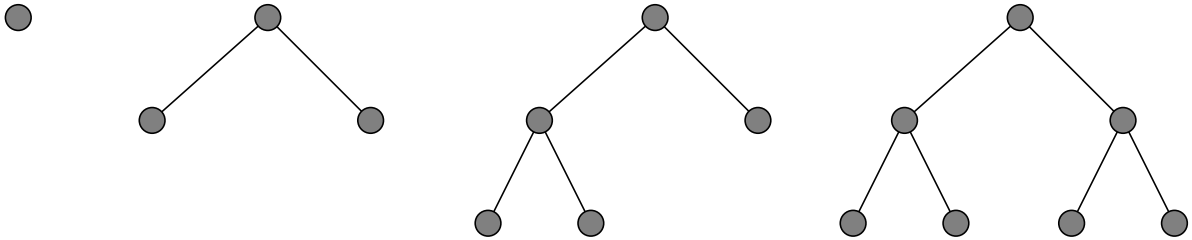
$O(1 + b + b^2 + \ldots + b^d + b(b^d - 1)) = O(b^{d+1})$ (exponential time)

where $b =$ maximum branching factor

$d =$ depth of shallowest goal state

Note: The above assumes that the search space if finite. What if it is not?

Cost of Breadth-First Search



Worst-case Space Complexity (no. of nodes in memory)

All nodes at depth d of the search tree are in the fringe when the procedure finds the goal state

The number of nodes at depth d in a tree with branching factor b is

$$O(b^{d+1}) \qquad \text{(exponential space)}$$

Cost of Breadth-First Search

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

$b = 10$, time/node=1ms, mem/node= 100bytes

- Exponential complexity problems become soon unmanageable
- Memory requirements are a bigger problem than time requirements

Optimality of Breadth-First Search

Breadth-first search is clearly complete.

Optimality of Breadth-First Search

Breadth-first search is clearly complete. Is it optimal?

Optimality of Breadth-First Search

Breadth-first search is clearly complete. Is it optimal? It depends

Optimality of Breadth-First Search

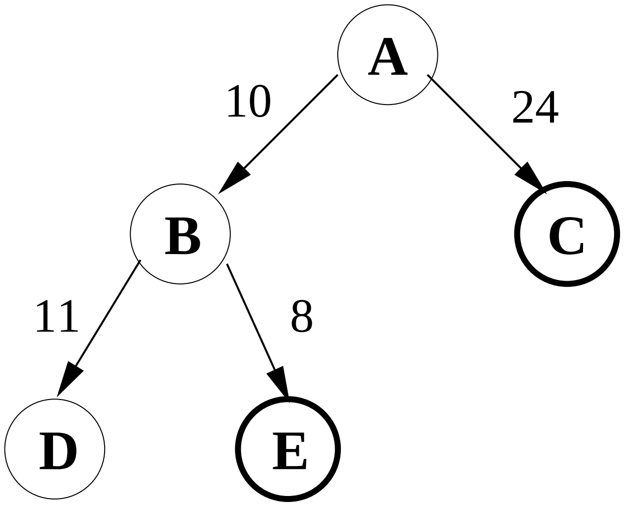
Breadth-first search is clearly complete. Is it optimal? It depends

- Breadth-first search always finds the shallowest goal state
- The path to that goal state, however, may have a higher cost than one to a deeper goal state

Optimality of Breadth-First Search

Breadth-first search is clearly complete. Is it optimal? It depends

- Breadth-first search always finds the shallowest goal state
- The path to that goal state, however, may have a higher cost than one to a deeper goal state



Cost of $\mathbf{AC}$: 24

Cost of $\mathbf{ABE}$: $10 + 8 = 18$

If we are looking for least-cost solutions, breadth-first is suboptimal unless all step costs are identical

Uniform-Cost Search

Assumption: A path cost function g such that $g(p) - g(p') \geq \epsilon > 0$ for all paths p and proper subpaths p' of p

Strategy: Expand least-cost unexpanded node

Implementation: $fringe =$ priority queue ordered by path cost

Uniform-Cost Search

Assumption: A path cost function g such that $g(p) - g(p') \geq \epsilon > 0$ for all paths p and proper subpaths p' of p

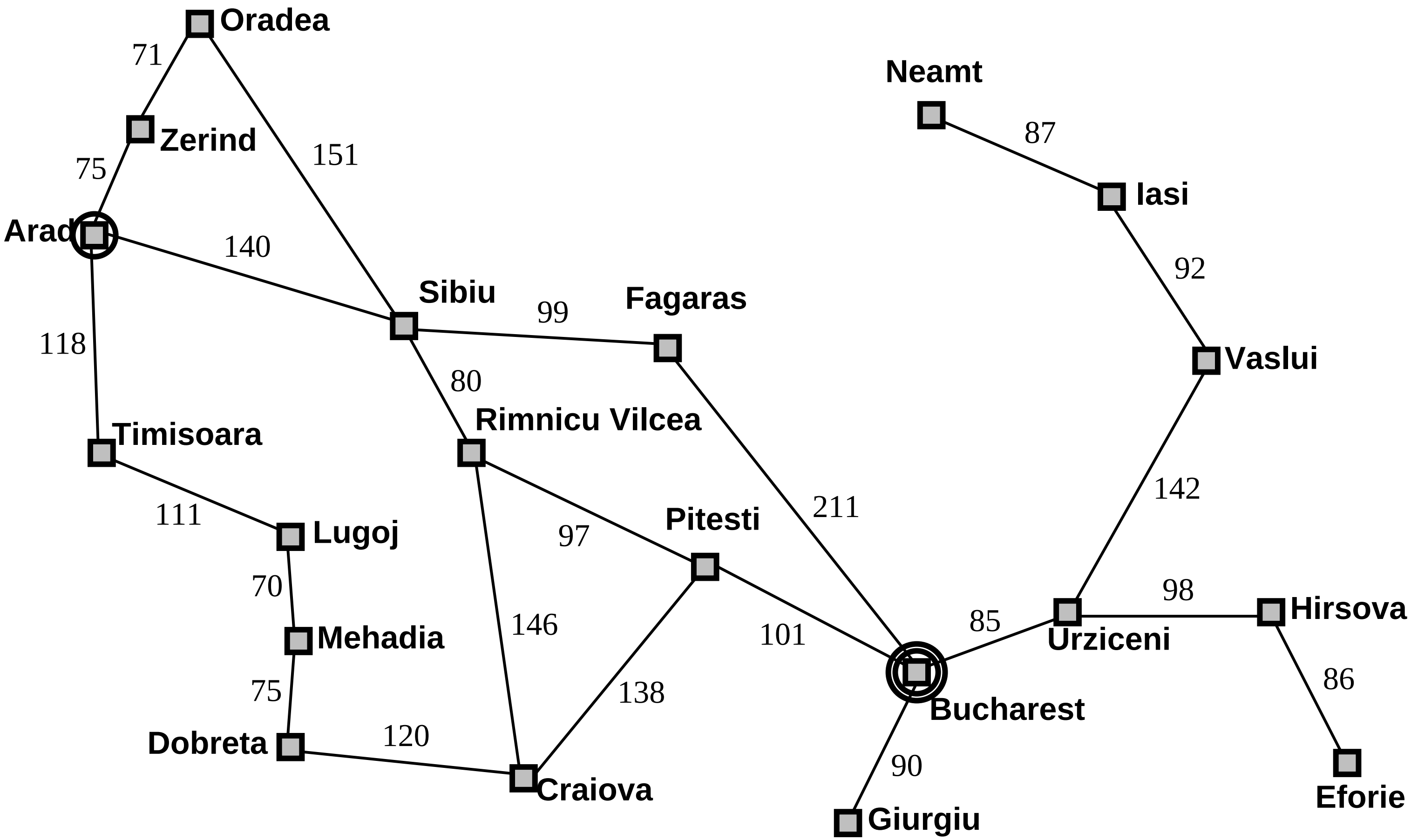
Strategy: Expand least-cost unexpanded node

Implementation: $fringe =$ priority queue ordered by path cost

Equivalent to breadth-first if step costs all equal

Uniform-Cost Search: Example

Path cost = sum of step costs



Exercise: Find cheapest route from Sibiu to Bucharest

Properties of Uniform-Cost Search

Assumption: A path cost function g such that $g(p) - g(p') \geq \epsilon > 0$ for all paths p and proper subpaths p' of p

Strategy: Expand least-cost unexpanded node

Implementation: $fringe =$ priority queue ordered by path cost

Complete?

Time complexity?

Space complexity?

Optimal?

Properties of Uniform-Cost Search

Assumption: A path cost function g such that $g(p) - g(p') \geq \epsilon > 0$ for all paths p and proper subpaths p' of p

Strategy: Expand least-cost unexpanded node

Implementation: $fringe =$ priority queue ordered by path cost

Complete? Yes (with step cost $\geq \epsilon$)

Time complexity?

Space complexity?

Optimal?

Properties of Uniform-Cost Search

Assumption: A path cost function g such that $g(p) - g(p') \geq \epsilon > 0$ for all paths p and proper subpaths p' of p

Strategy: Expand least-cost unexpanded node

Implementation: $fringe =$ priority queue ordered by path cost

Complete? Yes (with step cost $\geq \epsilon$)

Time complexity? $\#$ of paths p with $g(p) \leq$ cost of optimal solution: $O(b^{\lceil C^*/\epsilon \rceil})$ where C^* is the cost of the optimal solution

Space complexity?

Optimal?

Properties of Uniform-Cost Search

Assumption: A path cost function g such that $g(p) - g(p') \geq \epsilon > 0$ for all paths p and proper subpaths p' of p

Strategy: Expand least-cost unexpanded node

Implementation: $fringe =$ priority queue ordered by path cost

Complete? Yes (with step cost $\geq \epsilon$)

Time complexity? $\#$ of paths p with $g(p) \leq$ cost of optimal solution: $O(b^{\lceil C^*/\epsilon \rceil})$ where C^* is the cost of the optimal solution

Space complexity? Same as time complexity: $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?

Properties of Uniform-Cost Search

Assumption: A path cost function g such that $g(p) - g(p') \geq \epsilon > 0$ for all paths p and proper subpaths p' of p

Strategy: Expand least-cost unexpanded node

Implementation: $fringe =$ priority queue ordered by path cost

Complete? Yes (with step cost $\geq \epsilon$)

Time complexity? $\#$ of paths p with $g(p) \leq$ cost of optimal solution: $O(b^{\lceil C^*/\epsilon \rceil})$ where C^* is the cost of the optimal solution

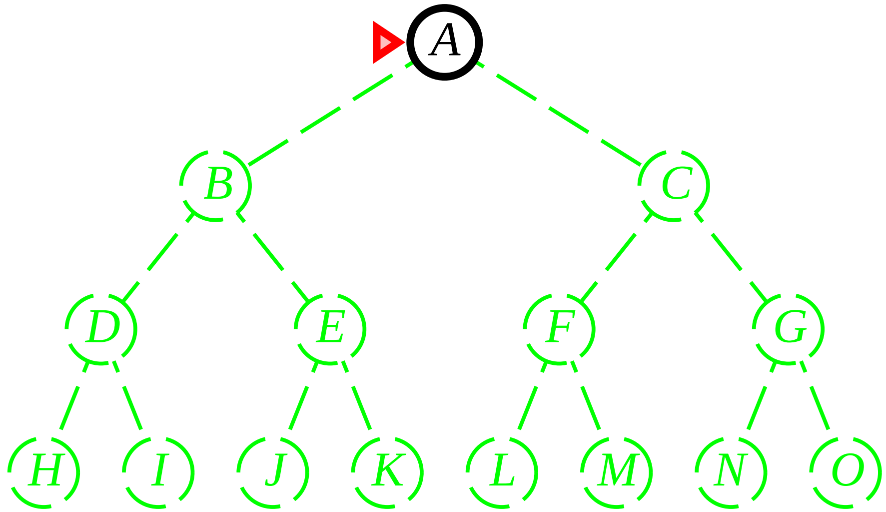
Space complexity? Same as time complexity: $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal? Yes, since nodes are expanded in increasing order of g

Depth-First Search

Strategy: Expand deepest unexpanded node

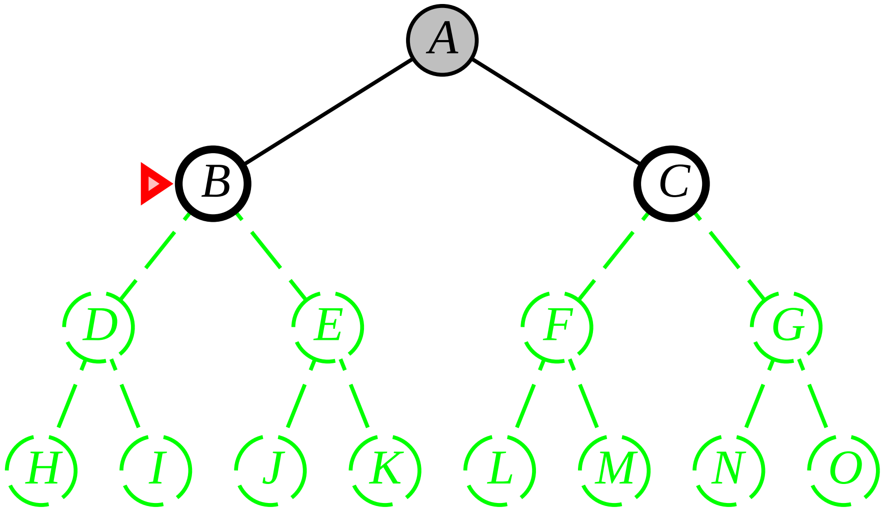
Implementation: $fringe =$ LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

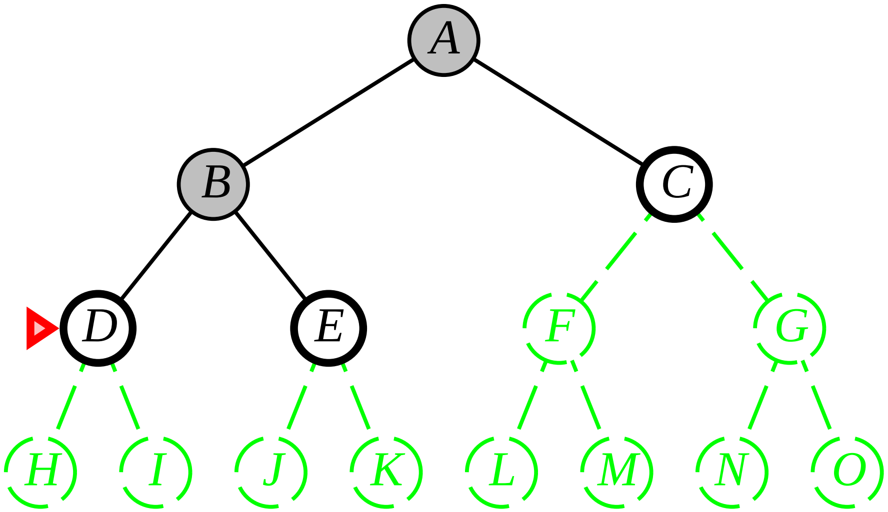
Implementation: $fringe =$ LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

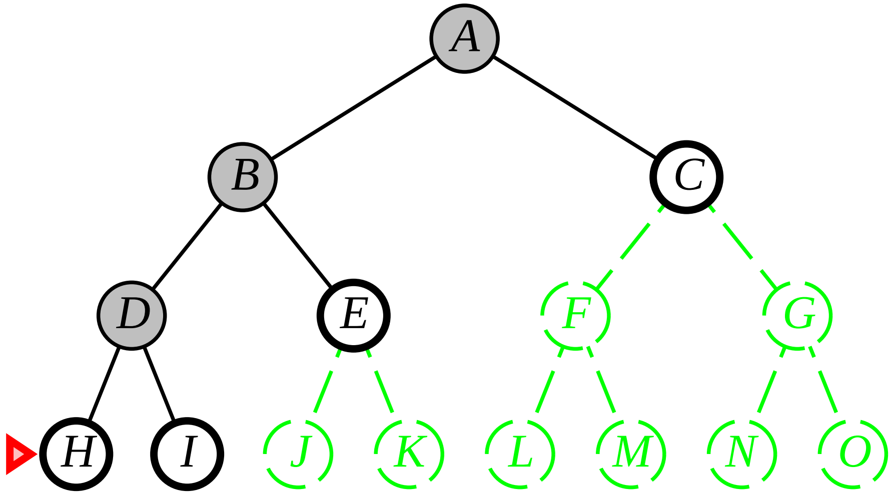
Implementation: $fringe =$ LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

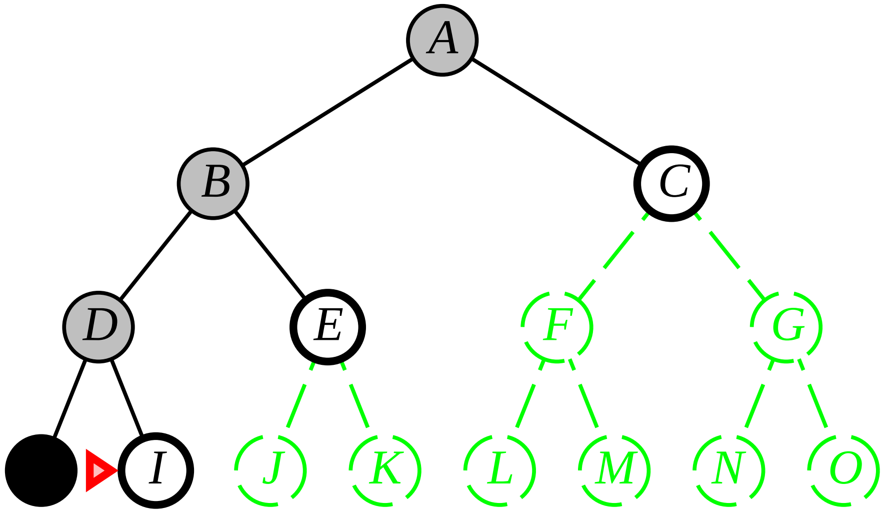
Implementation: $fringe =$ LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

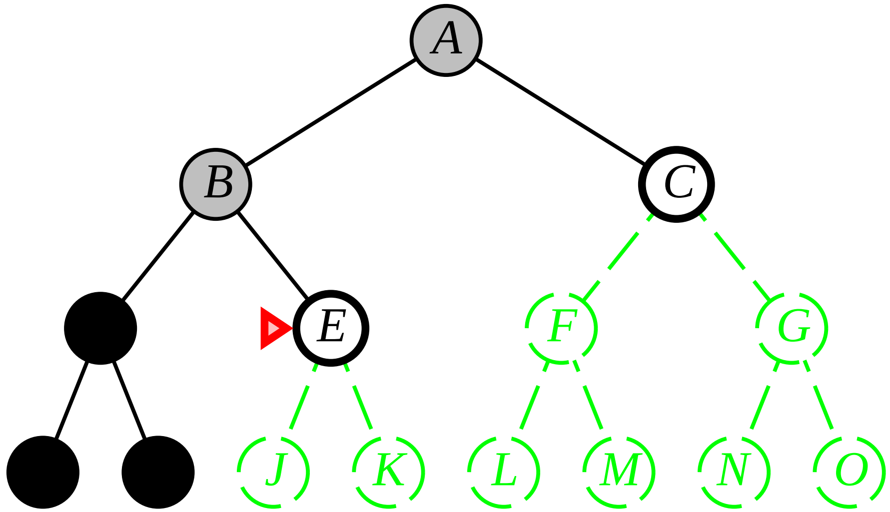
Implementation: $fringe$ = LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

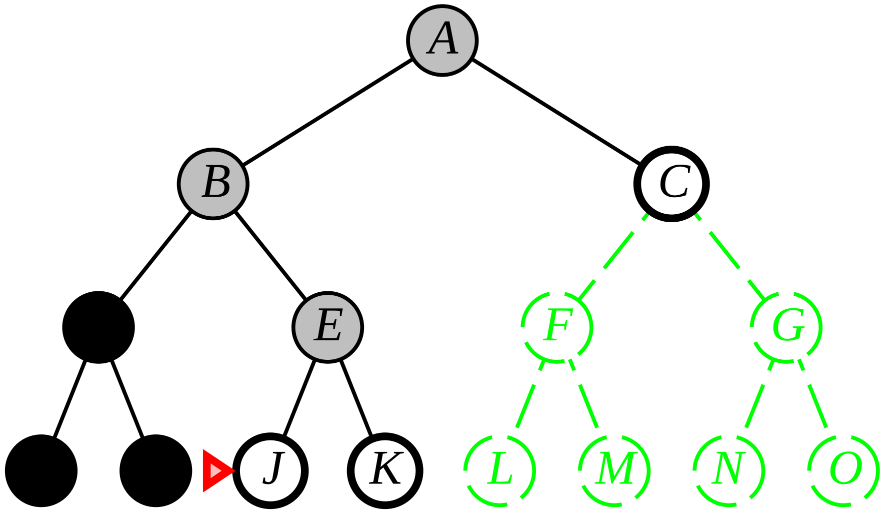
Implementation: $fringe =$ LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

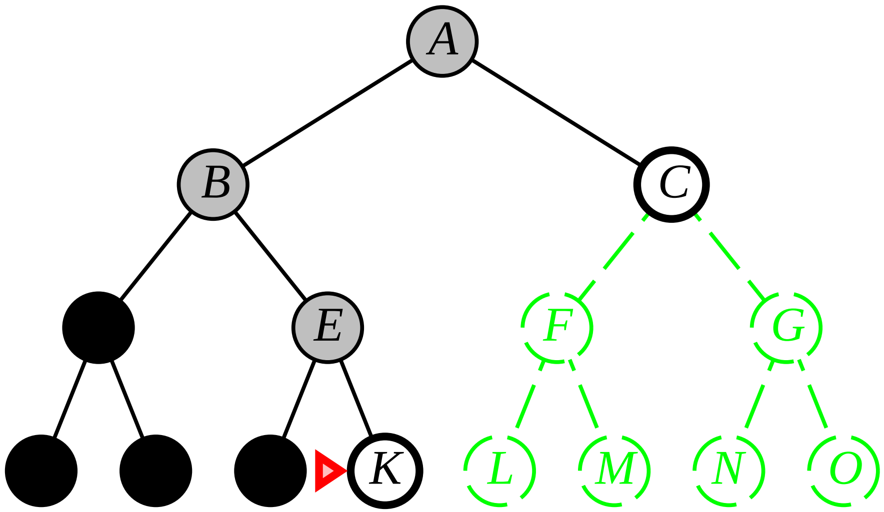
Implementation: $fringe =$ LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

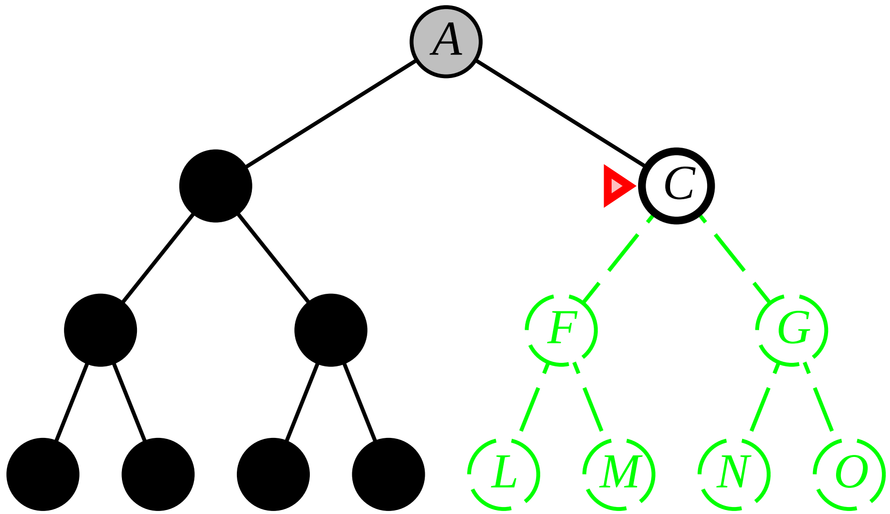
Implementation: $fringe$ = LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

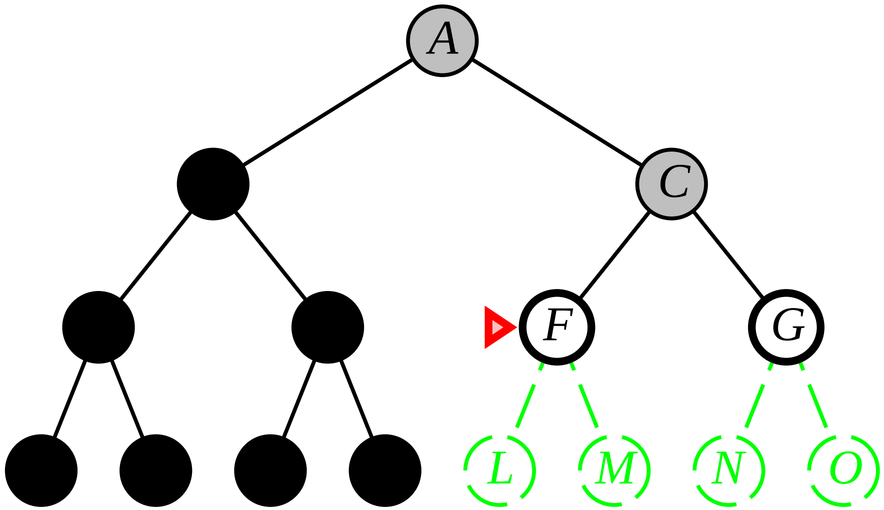
Implementation: $fringe =$ LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

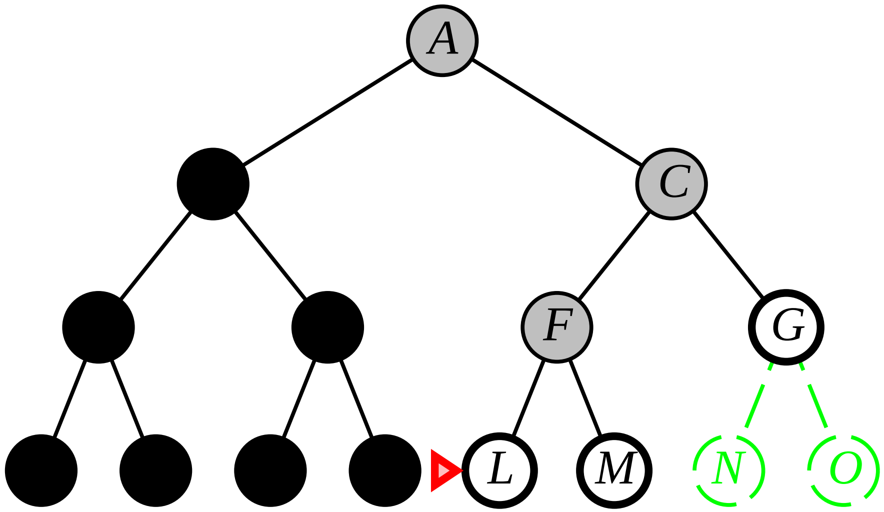
Implementation: $fringe =$ LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

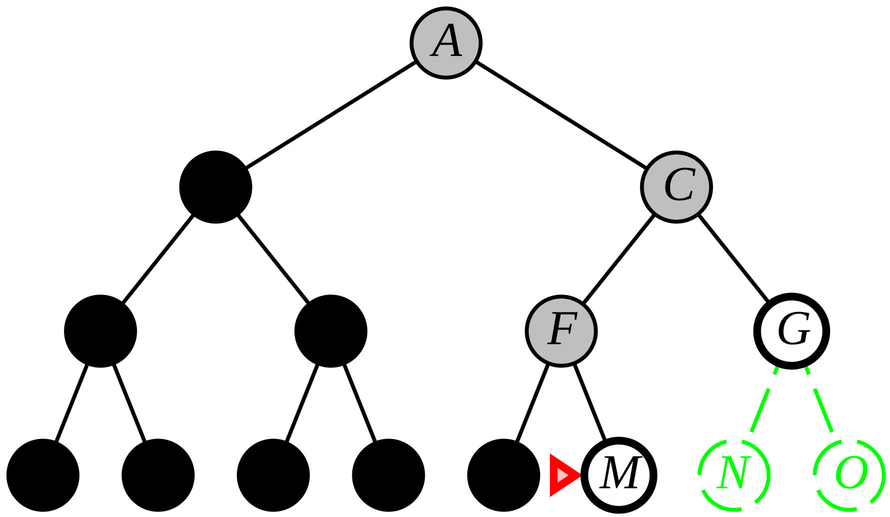
Implementation: $fringe =$ LIFO queue, i.e., put successors at front



Depth-First Search

Strategy: Expand deepest unexpanded node

Implementation: $fringe$ = LIFO queue, i.e., put successors at front



Properties of Depth-First Search

Complete?

Time complexity?

Space complexity?

Optimal?

Properties of Depth-First Search

Complete? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path $\Rightarrow$ complete in finite spaces

Time complexity?

Space complexity?

Optimal?

Properties of Depth-First Search

Complete? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path $\Rightarrow$ complete in finite spaces

Time complexity? $O(b^m)$: terrible if m is much larger than d but if solutions are dense, may be much faster than breadth-first

Space complexity?

Optimal?

Properties of Depth-First Search

Complete? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path $\Rightarrow$ complete in finite spaces

Time complexity? $O(b^m)$: terrible if m is much larger than d but if solutions are dense, may be much faster than breadth-first

Space complexity? $O(bm)$, i.e., linear space!

Optimal?

Properties of Depth-First Search

Complete? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path $\Rightarrow$ complete in finite spaces

Time complexity? $O(b^m)$: terrible if m is much larger than d but if solutions are dense, may be much faster than breadth-first

Space complexity? $O(bm)$, 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

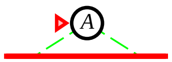
```
function Depth-Limited-Search (problem, limit) return soln/fail/cutoff
   return Recursive-DLS(Make-Node(Initial-State(problem)), problem, limit)
end function

function Recursive-DLS (node, problem, limit) return soln/fail/cutoff
   cutoff-occurred := false;
   if (Goal-State(problem, State(node))) then return 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 != fail) then return result;
        end for
        if (cutoff-occurred) then return cutoff; else return fail;
end function
```

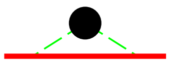
Iterative Deepening Search

```
function Iterative-Deepening-Search (problem) return soln
    for limit from 0 to MAX-INT do
       result := Depth-Limited-Search(problem, limit)
       if (result != cutoff) then return result
    end for
end function
```

Iterative Deepening Search

Limit = 0



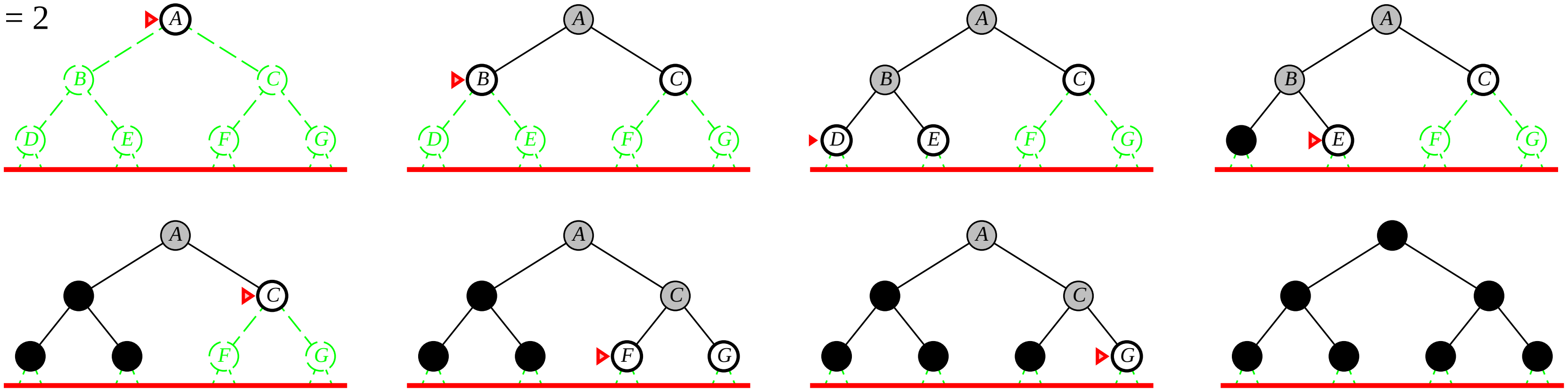


Iterative Deepening Search

Limit = 1

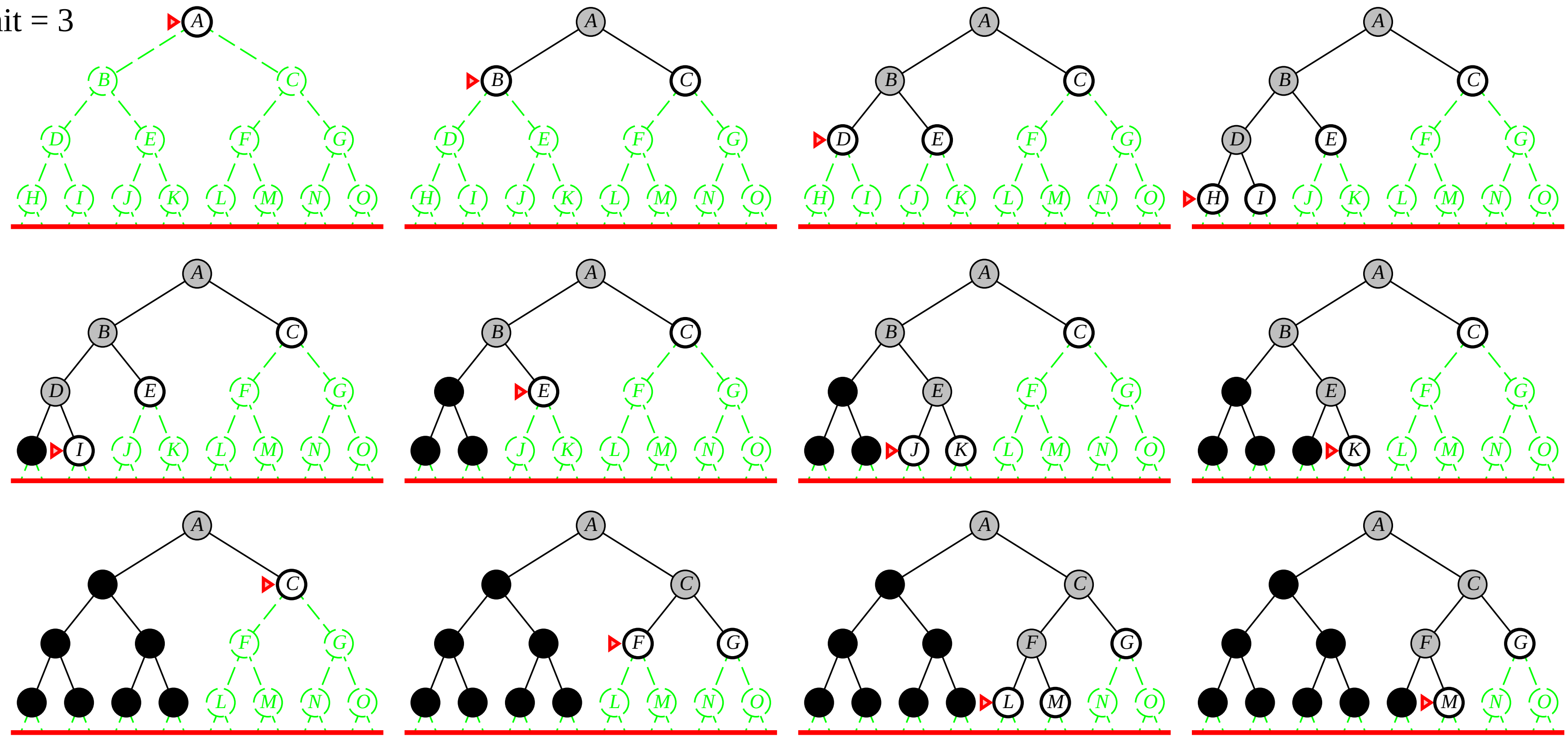
Iterative Deepening Search

Limit = 2



Iterative Deepening Search

Limit = 3



Properties of Iterative Deepening Search

Complete?

Time complexity?

Space complexity?

Optimal?

Properties of Iterative Deepening Search

Complete? Yes

Time complexity?

Space complexity?

Optimal?

Properties of Iterative Deepening Search

Complete? Yes

Time complexity? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space complexity?

Optimal?

Properties of Iterative Deepening Search

Complete? Yes

Time complexity? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space complexity? $O(bd)$

Optimal?

Properties of Iterative Deepening Search

Complete? Yes

Time complexity? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space complexity? $O(bd)$

Optimal? Only if step costs are all identical

Properties of Iterative Deepening Search

Complete? Yes

Time complexity? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space complexity? $O(bd)$

Optimal? Only if step costs are all identical

Numerical comparison between Iterative Deepening and Breadth First, with $b = 10$, $d = 5$, and solution at "far right" of search tree:

$$
\begin{aligned}
N(\text{ID}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
N(\text{BF}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
\end{aligned}
$$

Iterative deepening search is actually faster than breadth-first search!

Properties of Iterative Deepening Search

Complete? Yes

Time complexity? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space complexity? $O(bd)$

Optimal? Only if step costs are all identical

Numerical comparison between Iterative Deepening and Breadth First, with $b = 10$, $d = 5$, and solution at "far right" of search tree:

$$
\begin{aligned}
N(\text{ID}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
N(\text{BF}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
\end{aligned}
$$

Iterative deepening search is actually faster than breadth-first search!

It does better because other nodes at depth d are not expanded

Summary of Algorithms

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

b, branching factor d, depth of shallowest solution l, depth limit

m, depth of search tree C^*, cost of optimal solution

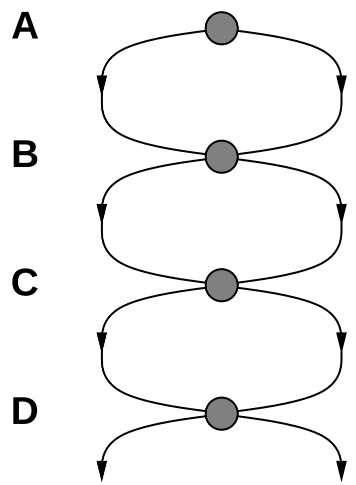
^{a} if b is finite

^{b} is step costs $> \epsilon$ for some $\epsilon > 0$

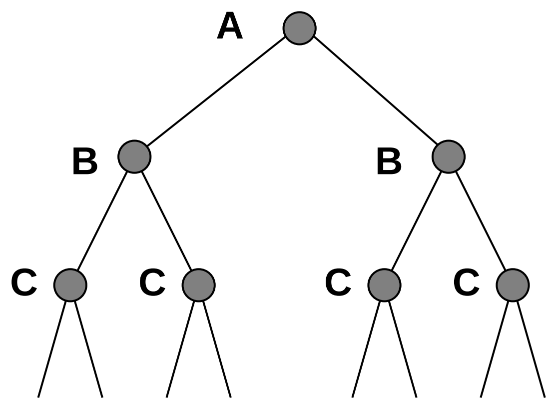
^{c} is all step costs are the same

Repeated States

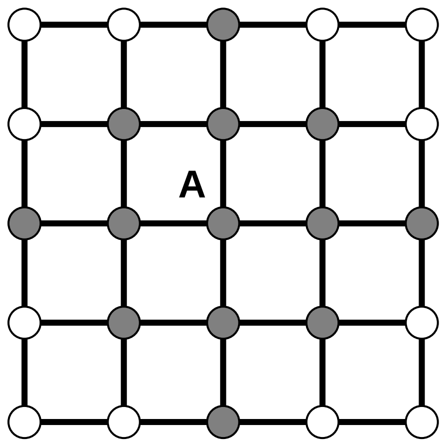
Failure to detect repeated states can turn a linear problem into an exponential one!



(a)



(b)



(c)