# Uninformed Search

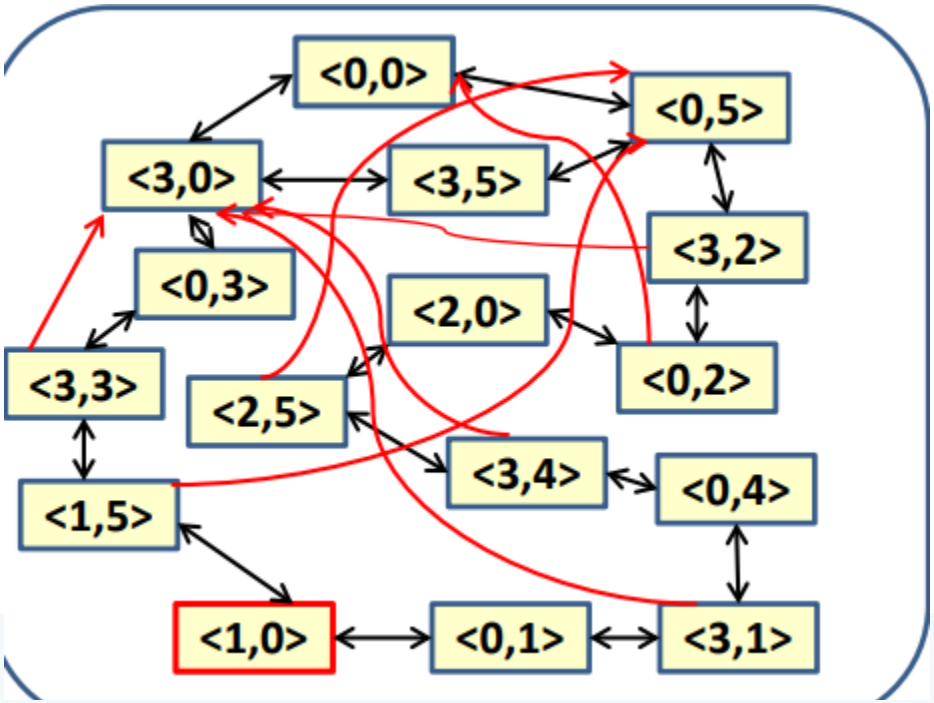
# AUTOMATED PROBLEM SOLVING BY SEARCH

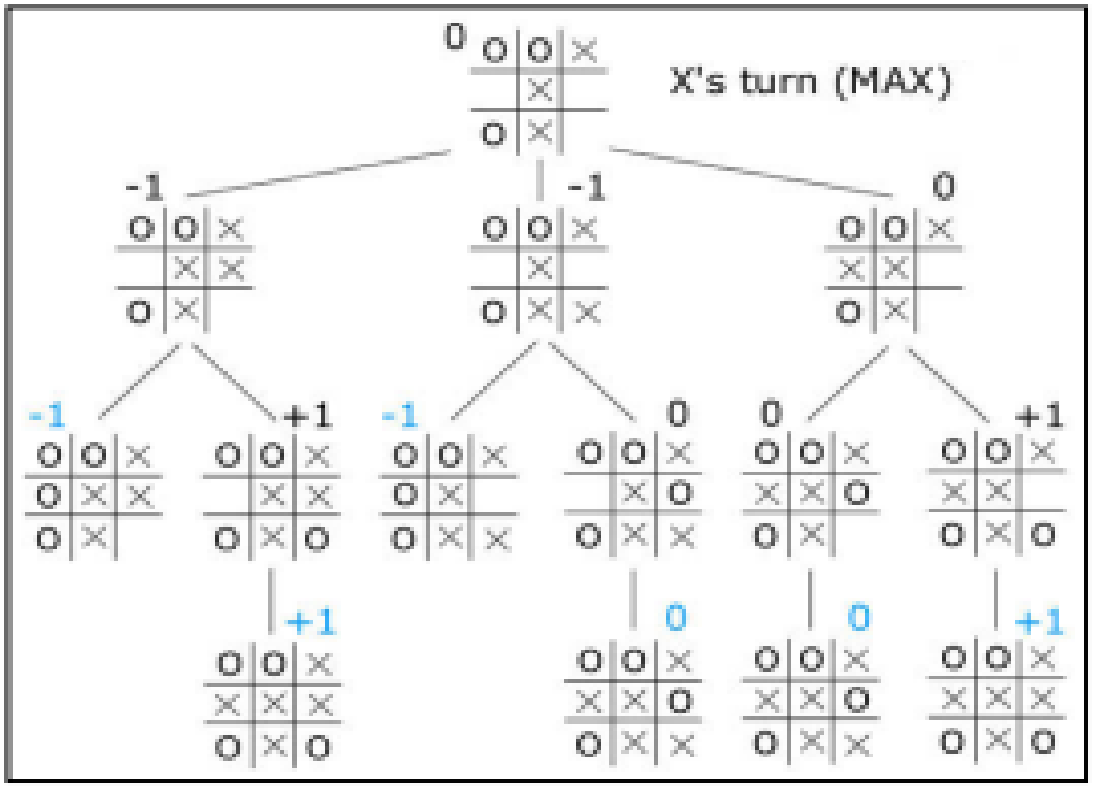
- Problem Formulation by AI Search Methods consists of the following key concepts
  - Configuration or State
  - Constraints or Definitions of Valid Configurations
  - Rules for Change of State and their Outcomes
  - Initial or Start Configurations
  - Goal Satisfying Configurations
  - An Implicit State or Configuration Space
  - Valid Solutions from Start to Goal in the State Space
  - General Algorithms which SEARCH for Solutions in this State Space

- **ISSUES**
  - **Size of the Implicit Space, Capturing Domain Knowledge, Intelligent Algorithms that work in reasonable time and Memory, Handling Incompleteness and Uncertainty**

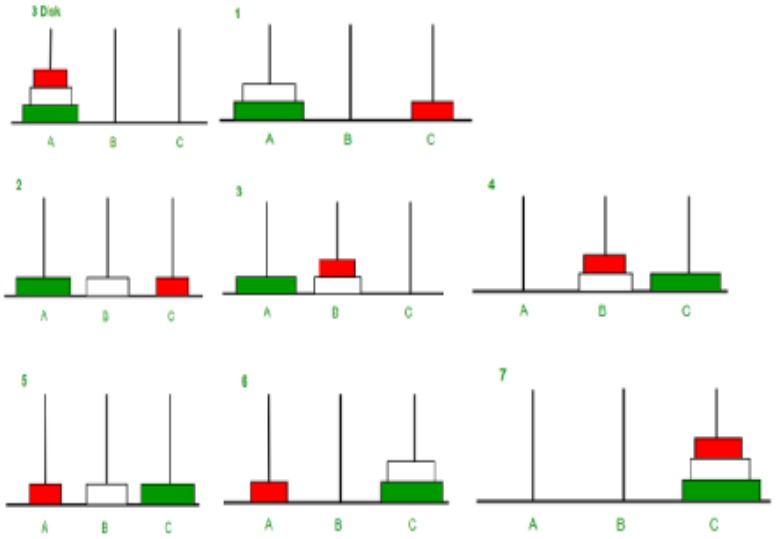
# TWO JUG PROBLEM

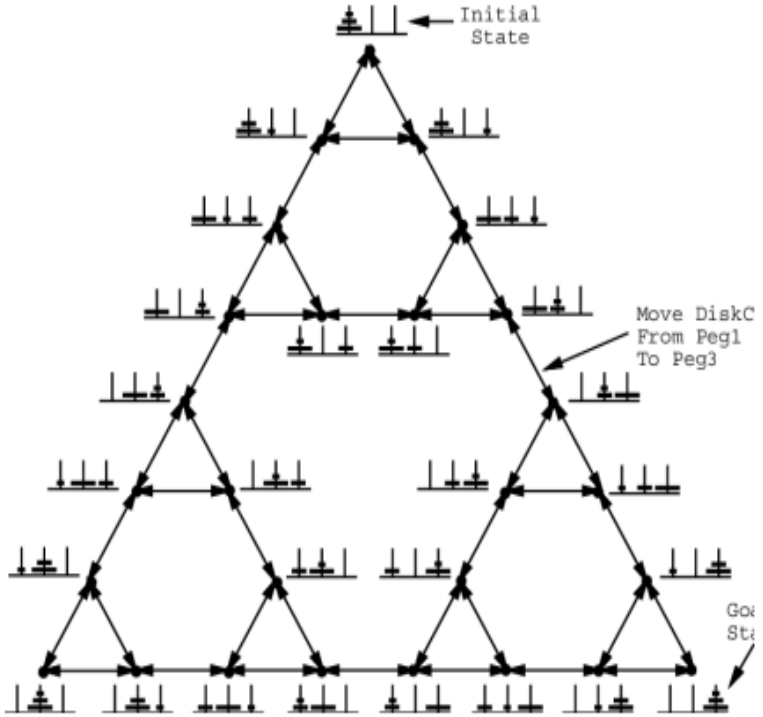
- There is a large bucket B full of water and Two (02) jugs, J1 of volume 3 litre and J2 of volume 5 litre. You are allowed to fill up any empty jug from the bucket, pour all water back to the bucket from a jug or pour from one jug to another. The goal is to have jug J1 with exactly one (01) litre of water
- State Definition: <J1, J2>
- Rules:
  - Fill (J1): <J1, J2> to <3,J2>
  - Fill (J2): <J1, J2> to <J1, 5>
  - Empty (J1), Empty (J2): Similarly defined
  - Pour (J1, J2): <J1, J2> to <X,Y>, where
    - X = 0 and Y = J1 + J2 if J1+J2 ≤ 5,
    - Y = 5 and X = (J1+ J2) – 5, if J1+J2 > 5
  - Pour (J2, J2): Similarly defined
- Start: <0,0>, Goal: <1,0>




X's turn (MAX)
Game

# 3 DISK, 3 PEG TOWER of HANOI STATE SPACE



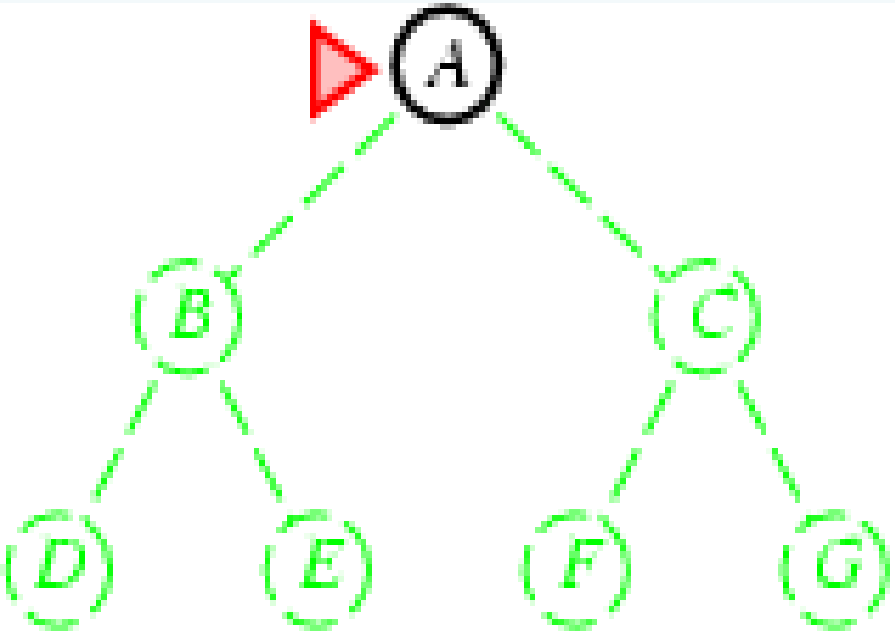
# Searching

- The SEARCH TREE represents possible action sequences starting at the initial state i.e. the root node.
- Expand an ***unexpanded*** node
  - test whether this is a goal state.
- The current state is expanded by applying each legal action to the current state, thereby **generating a new set of states.**
- Now we choose which of these states considered further to expand.
- Follow one option now and putting the others aside for later, in case the first choice does not lead to a solution, Repeat.

# Uninformed Search Strategies

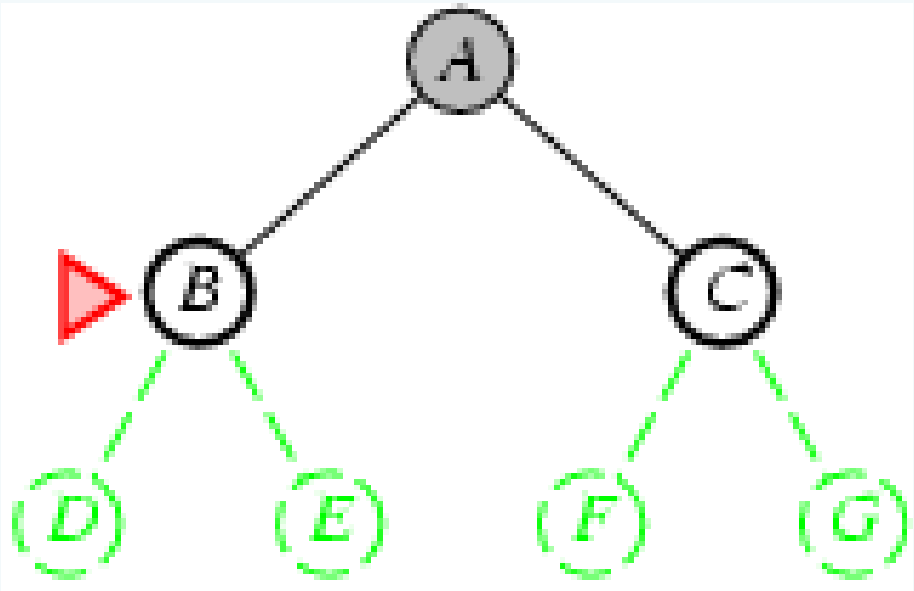
- Uninformed search strategies use only the information available in the problem definition
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

# Breadth-first search



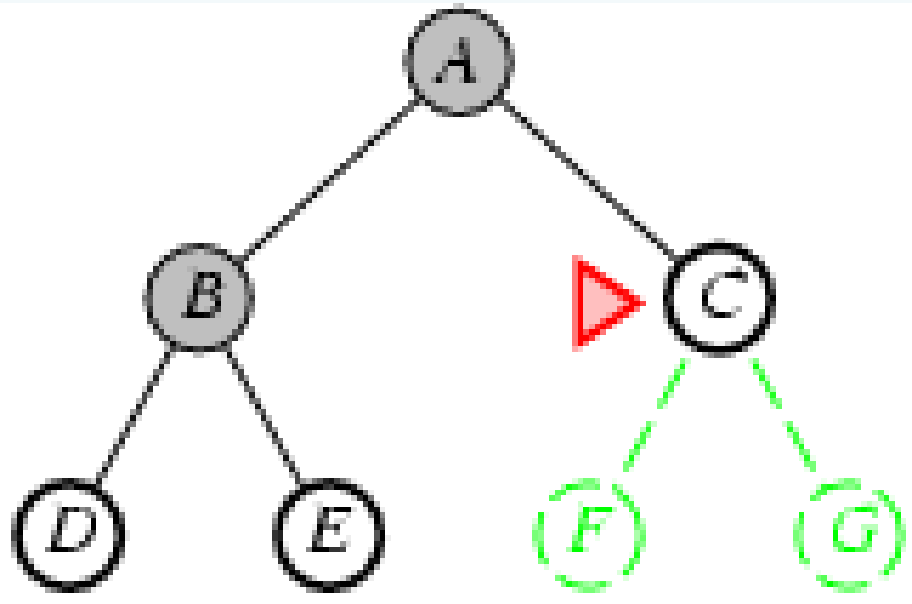
# Breadth-first search

- Expand shallowest unexpanded node

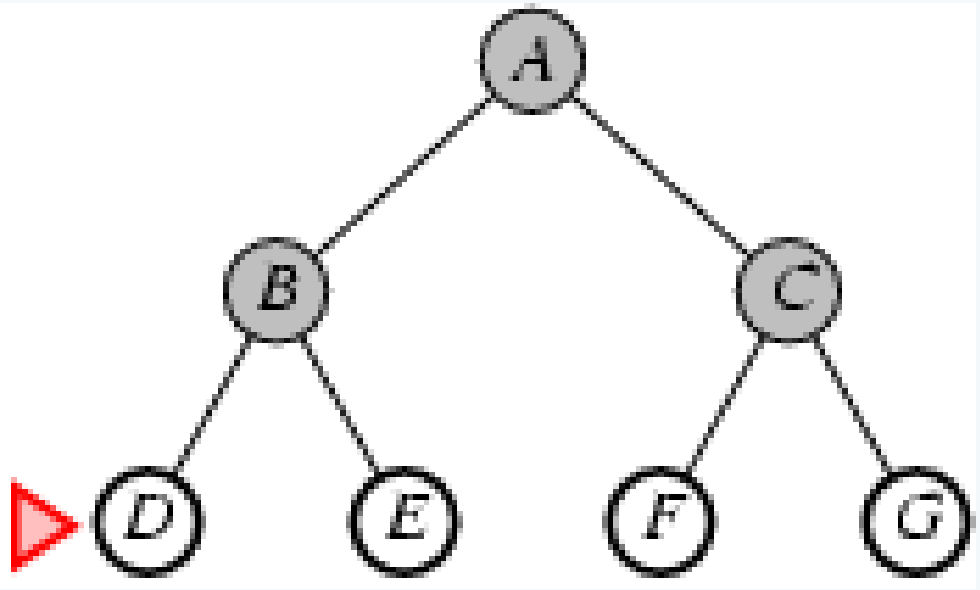



- Nodes that have been expanded are shaded;
- Nodes that have been generated but not yet expanded are outlined in bold;
- Nodes that have not yet been generated are shown in faint dashed lines.

# Breadth-first search



# Breadth-first search



# Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

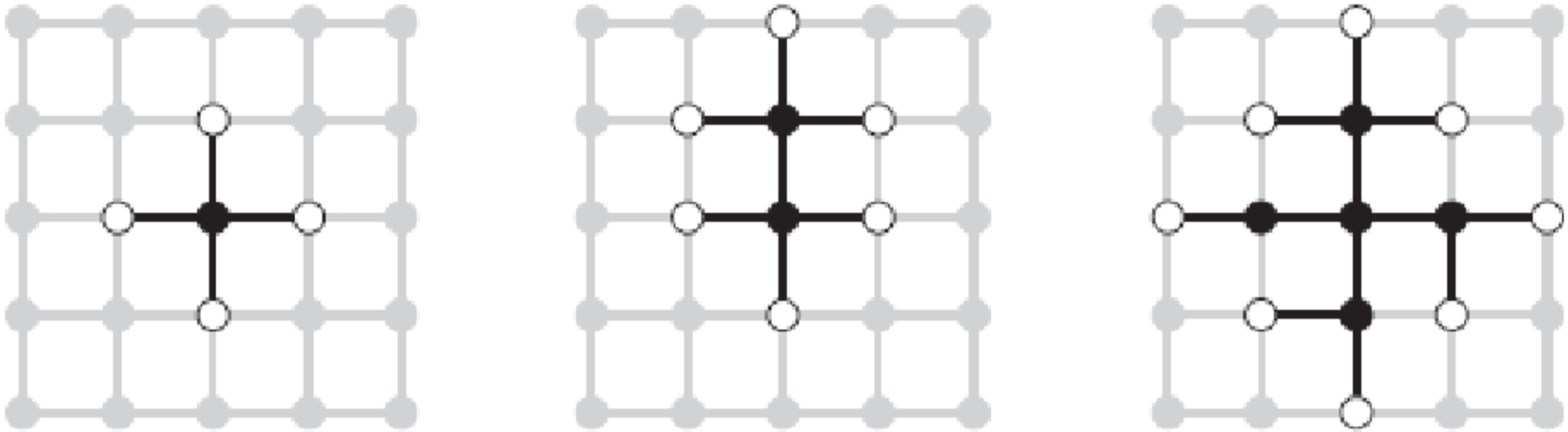
- In some cases, redundant paths are unavoidable, however, the actions are reversible, such as route-finding problems and sliding-block puzzles.
- In some cases actions are irreversible.
- The set of all leaf nodes available for expansion at any given point is called the **frontier.**

The way to avoid exploring redundant paths is to remember where one has been.

Augment the TREE-SEARCH algorithm with a data structure called the **explored set (**also known as the **closed list),** which remembers every expanded node.

Newly generated nodes that match previously generated nodes—ones in the explored set—can be discarded instead of being added to the frontier.

In Graph Search Algorithm the frontier separates the state-space graph into the explored region and the unexplored region.



The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is *systematically* examining the states in the state space, one by one, until it finds a solution.

# Infrastructure for search algorithms

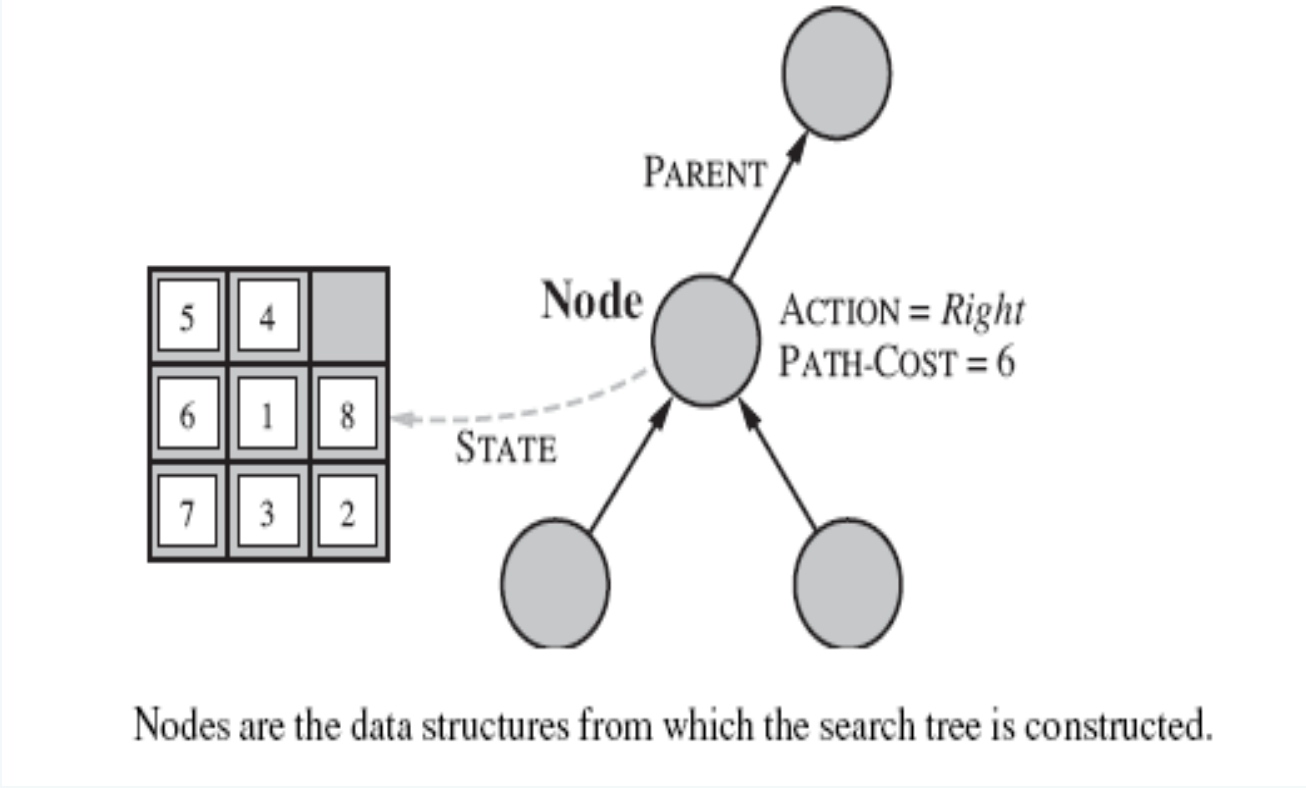
For each node *n* of the tree, we have a structure that contains four components:

• n.STATE: the state in the state space to which the node corresponds;

• n.PARENT: the node in the search tree that generated this node;

• n.ACTION: the action that was applied to the parent to generate the node;

• n.PATH-COST: the cost denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.




Nodes are the data structures from which the search tree is constructed.

1. Each Node has a parent, a state, and various bookkeeping fields.
2. Arrows point from child to parent.
3.A node is a bookkeeping data structure used to represent the search tree.
4. A state corresponds to a configuration of the world. Thus, nodes are on particular paths, as defined by PARENT pointers, whereas states are not.
5. Furthermore, two different nodes can contain the same world state if that state is generated via two different search paths.

Nodes are to be put somewhere.
The appropriate data structure is a **queue.**
**The** operations on a queue are:
• EMPTY?(queue) returns true only if there are no more elements in the queue.
• POP(queue) removes the first element of the queue and returns it.
• INSERT(element, queue) inserts an element and returns the resulting queue.

Queues are characterized by the *order in which they store the inserted nodes.*

- In AI, the graph is often represented *implicitly by the initial state, actions, and transition model and is frequently infinite.*

- Complexity is expressed in terms of : *b*, the **branching factor or maximum number of successors of any node;** ***d*, the depth of the shallowest goal** node (i.e., the number of steps along the path from the root); and *m*, **the maximum length of any path in the state space.**

- Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory.

For the most part, we describe time and space complexity for search on a tree; for a graph, the answer depends on how "redundant" the paths in the state space are.

For breadth-first graph search in particular, every node generated remains in memory.

There will be $O(b^{d-1})$ nodes in the explored set and $O(b^{d})$ nodes in the frontier, so the space complexity is $O(b^{d})$.

First, *the memory requirements are a bigger problem for breadth-first search than is the execution time.*

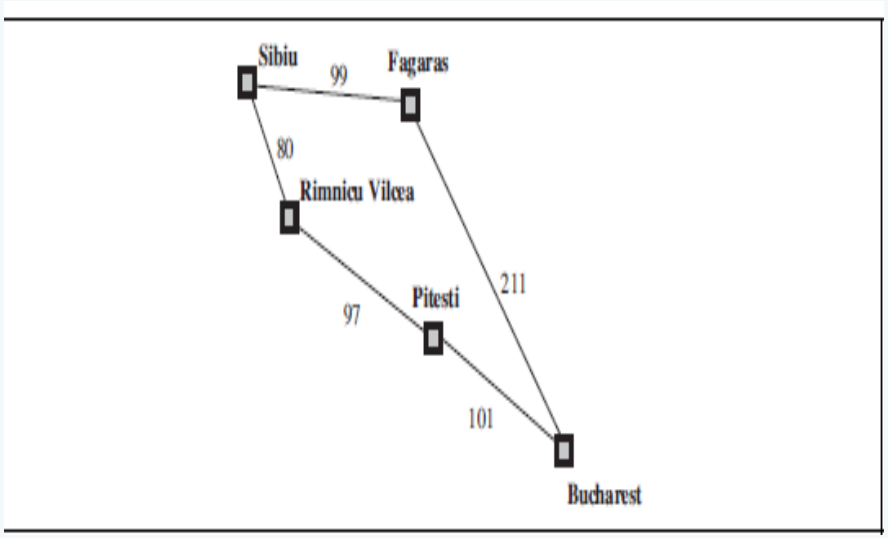
If your problem has a solution at a depth *d* with large value, then it will take huge time for breadth-first search (or indeed any uninformed search) to find it.

# 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? $O(b^d)$ (keeps every node in memory)
- Optimal? Yes, when all step costs are equal, breadth-first search is optimal because it always expands the *shallowest unexpanded node.*
- Space is the bigger problem (more than time).
- In general, *exponential-complexity search problems cannot be solved by uninformed methods*

# Uniform-cost search

- UCS traversing via different edges might not have the same cost. The goal is to find a path where the cumulative sum of costs is the least.
- Whenever uniform-cost search selects a node *n* for expansion, the optimal path to that node has been found.
- Step costs are nonnegative, paths never get shorter as nodes are added.
- Uniform-cost search expands nodes in order of their optimal path cost.
- Uniform-cost search does not care about the *number of steps a path has, but only about* their total cost.




•The problem is to get from **Sibiu** to **Bucharest**. The successors of **Sibiu** are **Rimnicu Vilcea** and **Fagaras**, with costs 80 and 99, respectively.

- The least-cost node, **Rimnicu Vilcea**, expanded next, adding **Pitesti** with cost 80 +97=177.

- The least-cost node is now **Fagaras**, so it is expanded, adding **Bucharest** with cost 99+211=310.

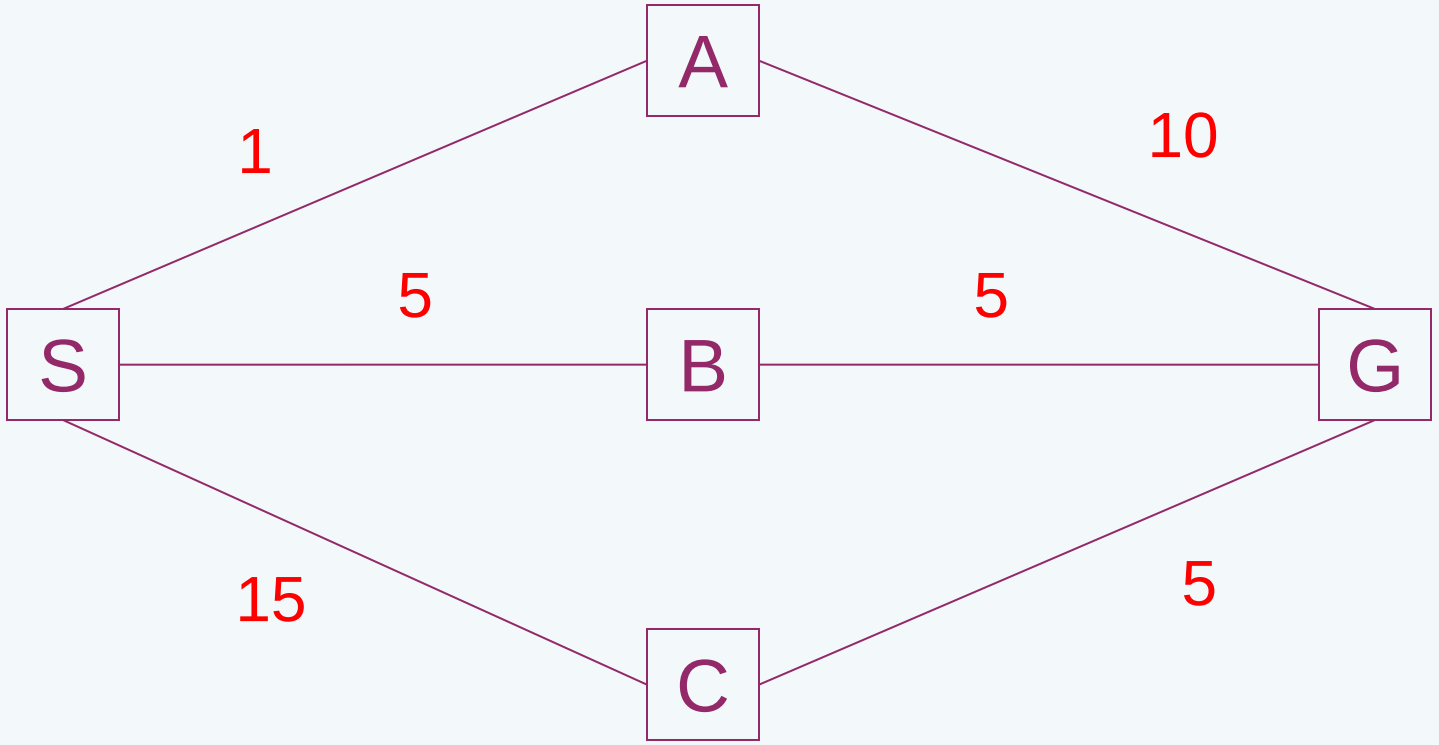
- Now a goal node has been generated, but uniform-cost search keeps going, choosing **Pitesti** for expansion and adding a second path to Bucharest with cost 80+97+101= 278.

•Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. **Bucharest**, now with g-cost 278, is selected for expansion and the solution is returned.

- Uniform-cost search will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions—for example, a sequence of NoOp actions.
- Completeness is guaranteed provided the cost of every step exceeds some small positive constant ε.
- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of *b* and *d*.
- Say, C be the cost of the optimal solution, and assume that every action costs at least ε.
- The algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C/\varepsilon \rfloor})$
- When all step costs are equal, $O(b^{1+\lfloor C/\varepsilon \rfloor})$ is just $b^{d+1}$

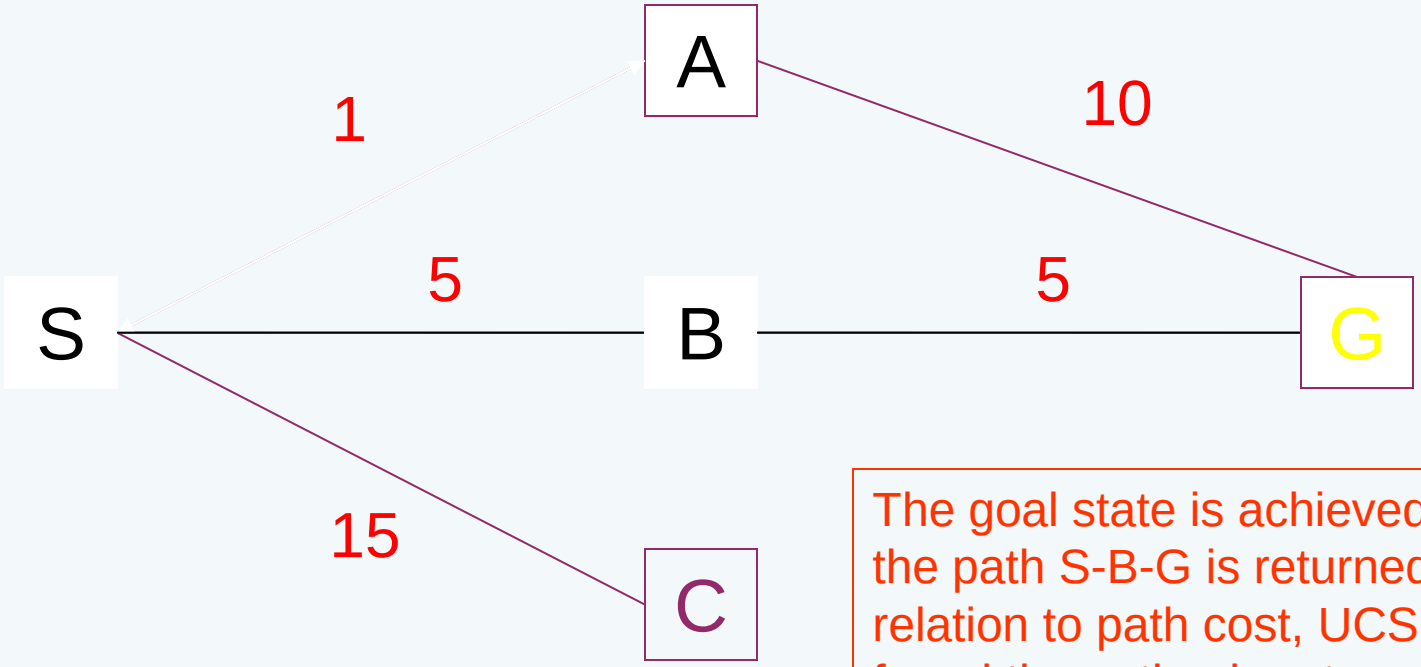
- When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

Consider the following problem…

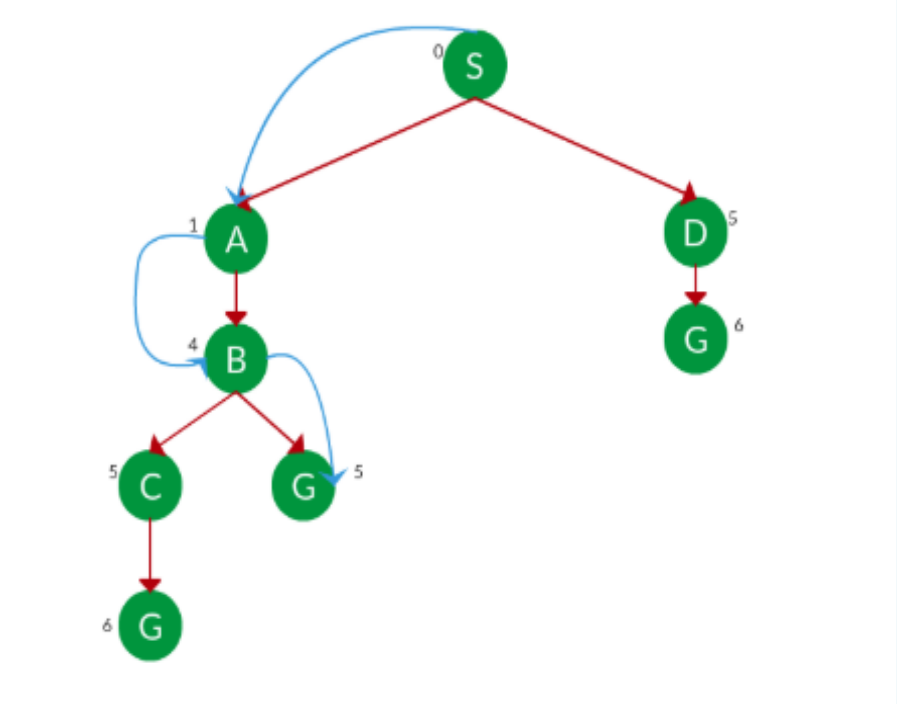



We wish to find the shortest route from node S to node
In terms of path cost, we can clearly see that the route *SBG* is the cheapest route.
However, if we let breadth-first search loose on the problem it will find the non-optimal path *SAG*, assuming that A is the first node to be expanded at level 1.

We start with our initial state and expand it…




The goal state is achieved and the path S-B-G is returned. In relation to path cost, UCS has found the optimal route.




Path-
Cost (C) =
ε =
Time complexity =

# Uniform-Cost Search

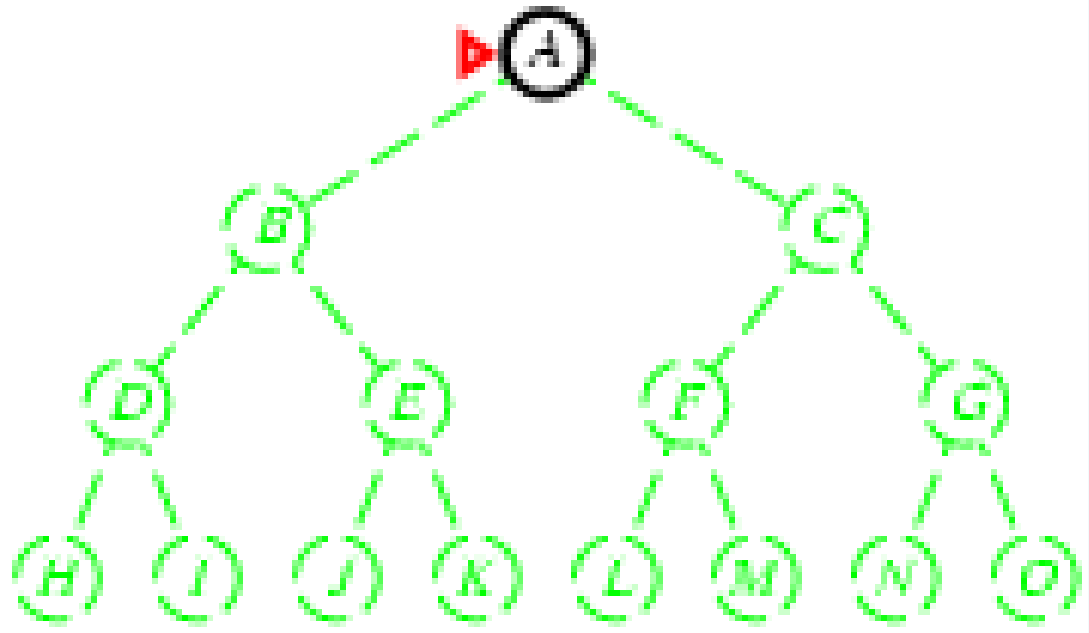
- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost $\geq \varepsilon$
- Time? $O(b^{\lceil (C/\varepsilon) \rceil})$ where $C$ is the cost of the optimal solution
- Space? $O(b^{\lceil (C/\varepsilon) \rceil})$
- Optimal? Yes – given the condition of completeness – you always expand the node with lowest cost
  - $O(b^{\lceil (C/\varepsilon) \rceil})$ can be greater than $O^{b/d}$

# Uniform-Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

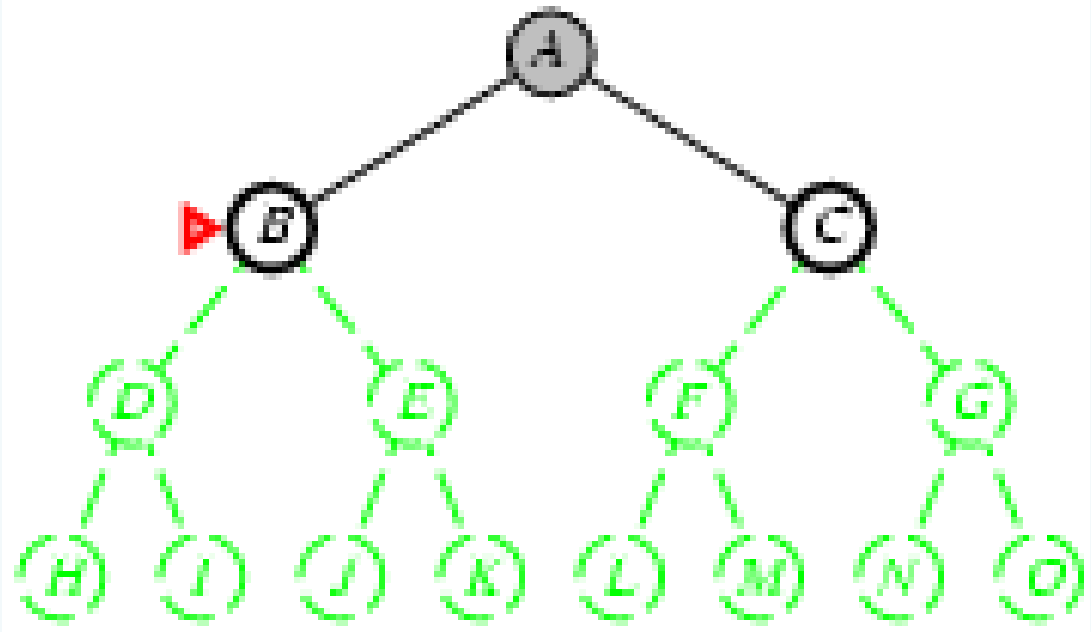
# Depth-first search

- Expand deepest unexpanded node



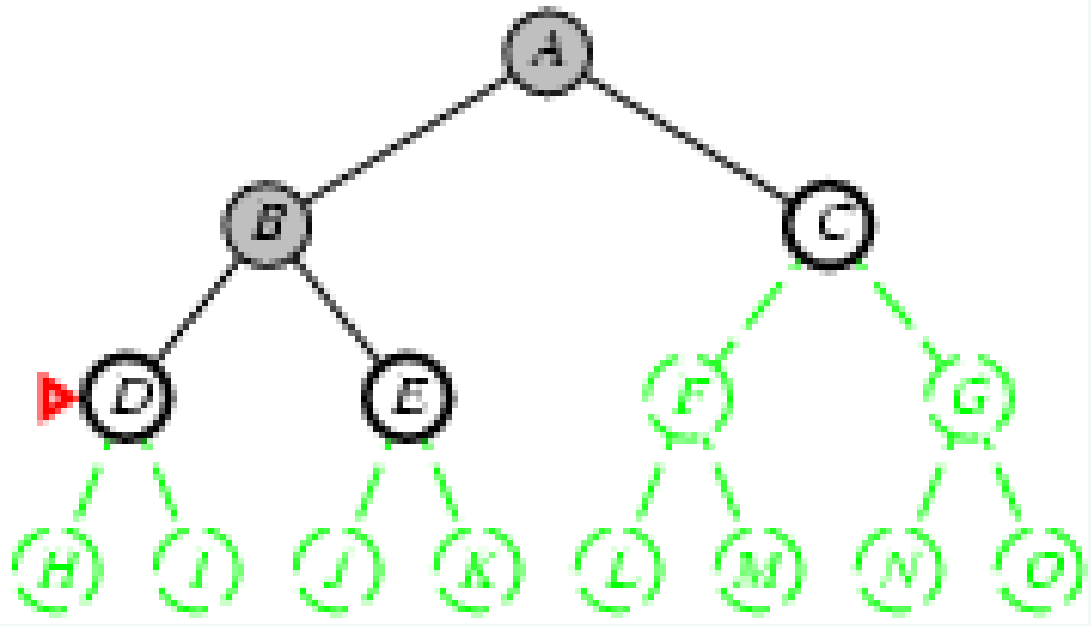
# Depth-first search

- Expand deepest unexpanded node



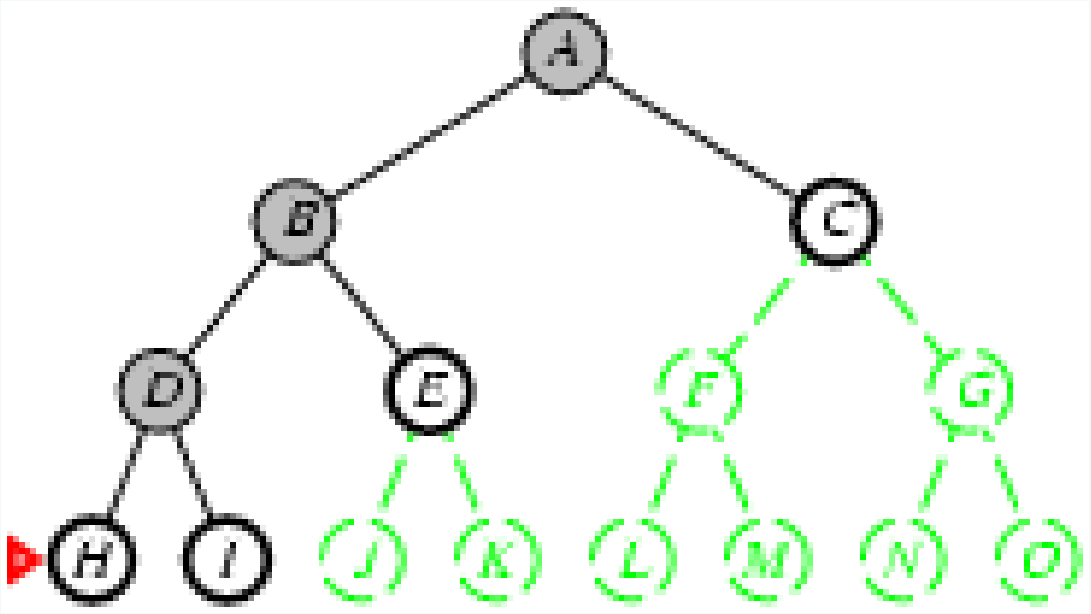
# Depth-first search

- Expand deepest unexpanded node



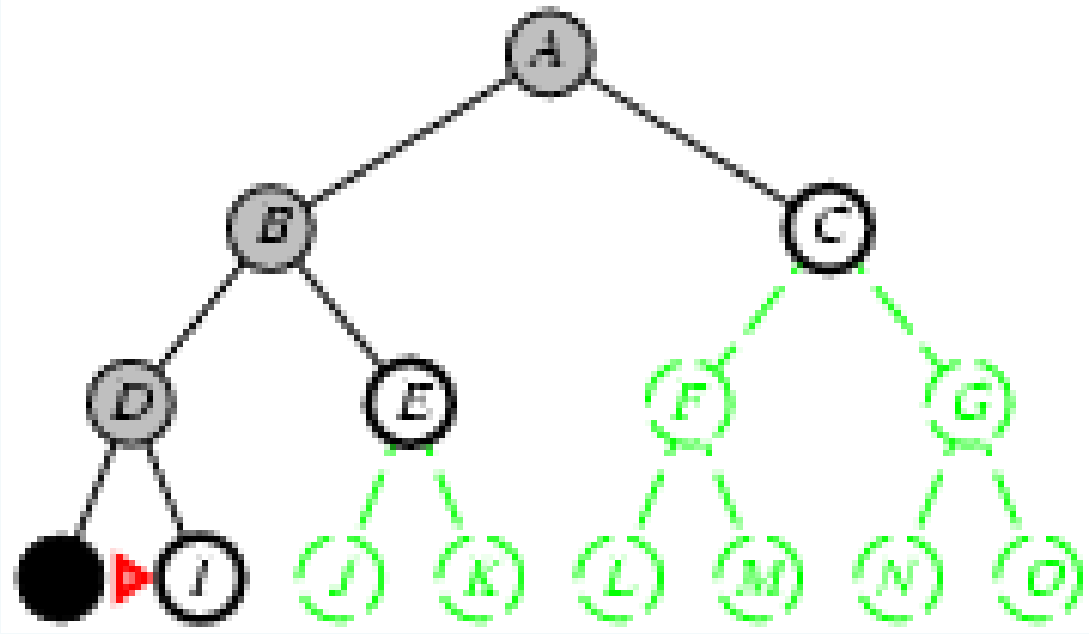
# Depth-first search

- Expand deepest unexpanded node



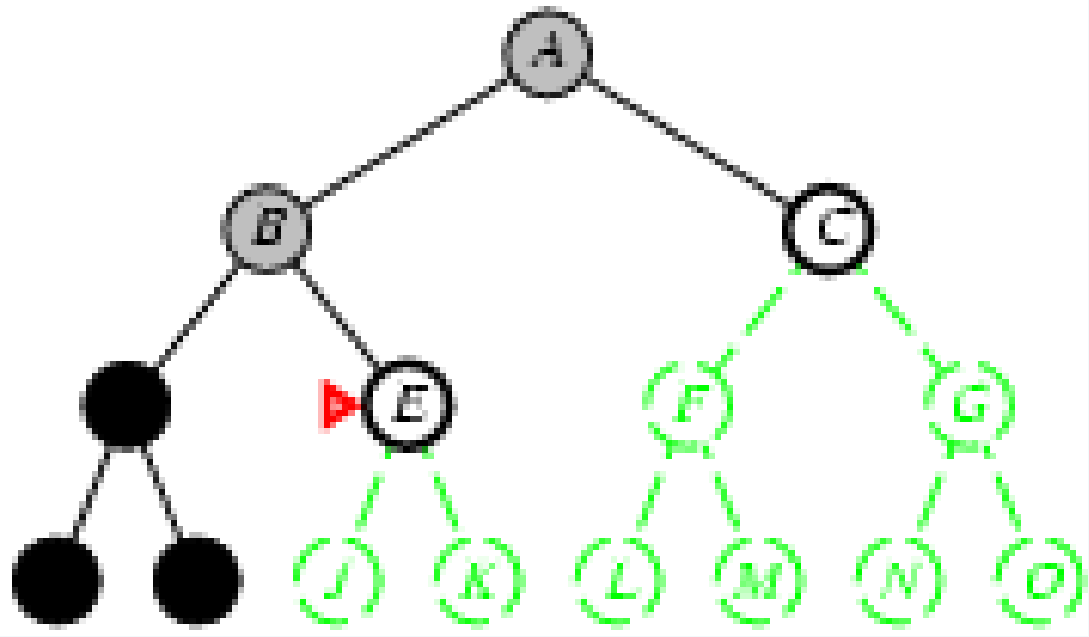
# Depth-first search

- Expand deepest unexpanded node



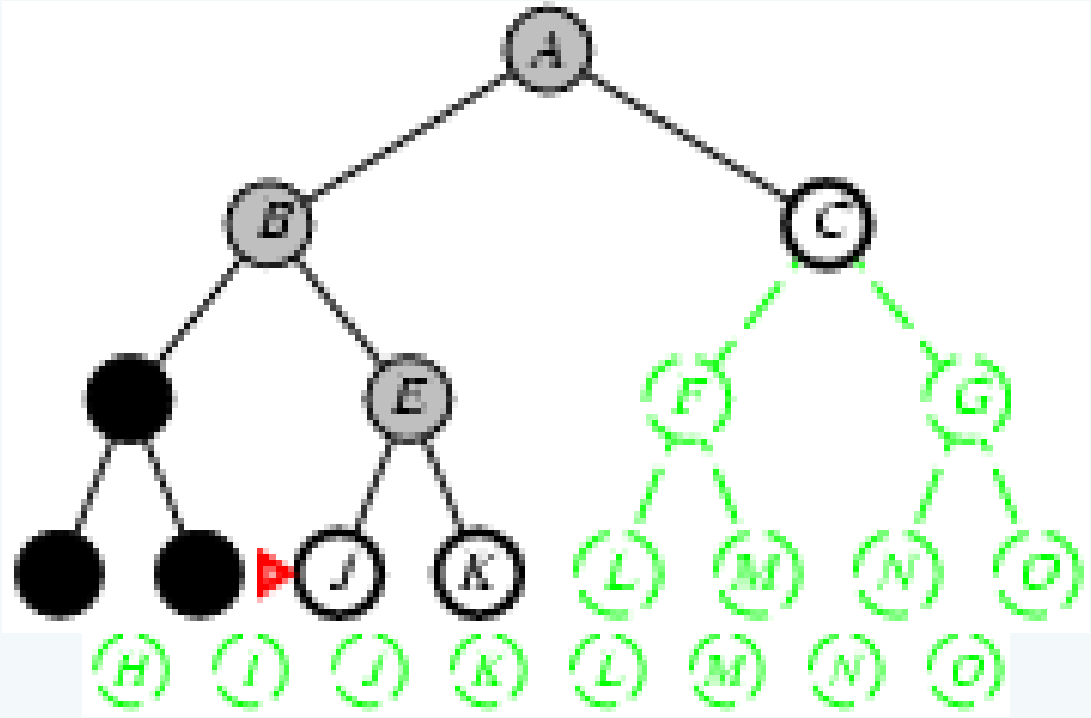
# Depth-first search

- Expand deepest unexpanded node



# Depth-first search

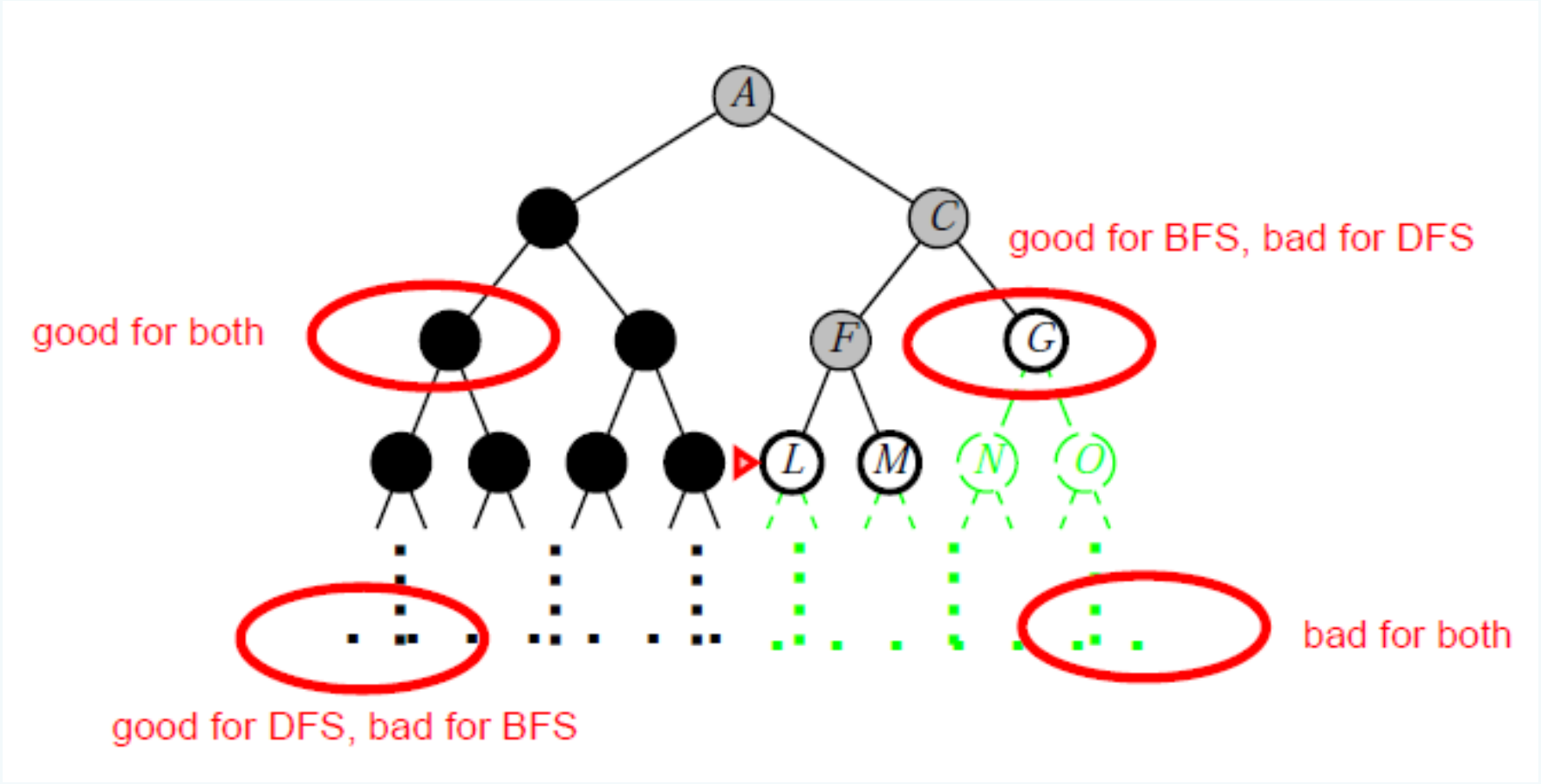
- Expand deepest unexpanded node



# Properties of Depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - 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

# BFS or DFS


A
C
F
G
L
M
N
O
good for both
good for BFS, bad for DFS
bad for both
good for DFS, bad for BFS

# Depth-limited search

- Depth-first search with depth limit *l*, i.e., nodes at depth *l* have no successors, knowledge is available.

Depth-first search is a special case of depth-limited with *l* being infinite.

- Depth-limited search can solve the drawback of the infinite path in the Depth-first search.
- In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

# Depth-Limited Search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

- Complete? NO. Why? ($l$ < d OR $l$ > d)
- Time? $O(b^l)$
- Space? $O(bl)$

# Iterative deepening search

Iterative deepening search gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.

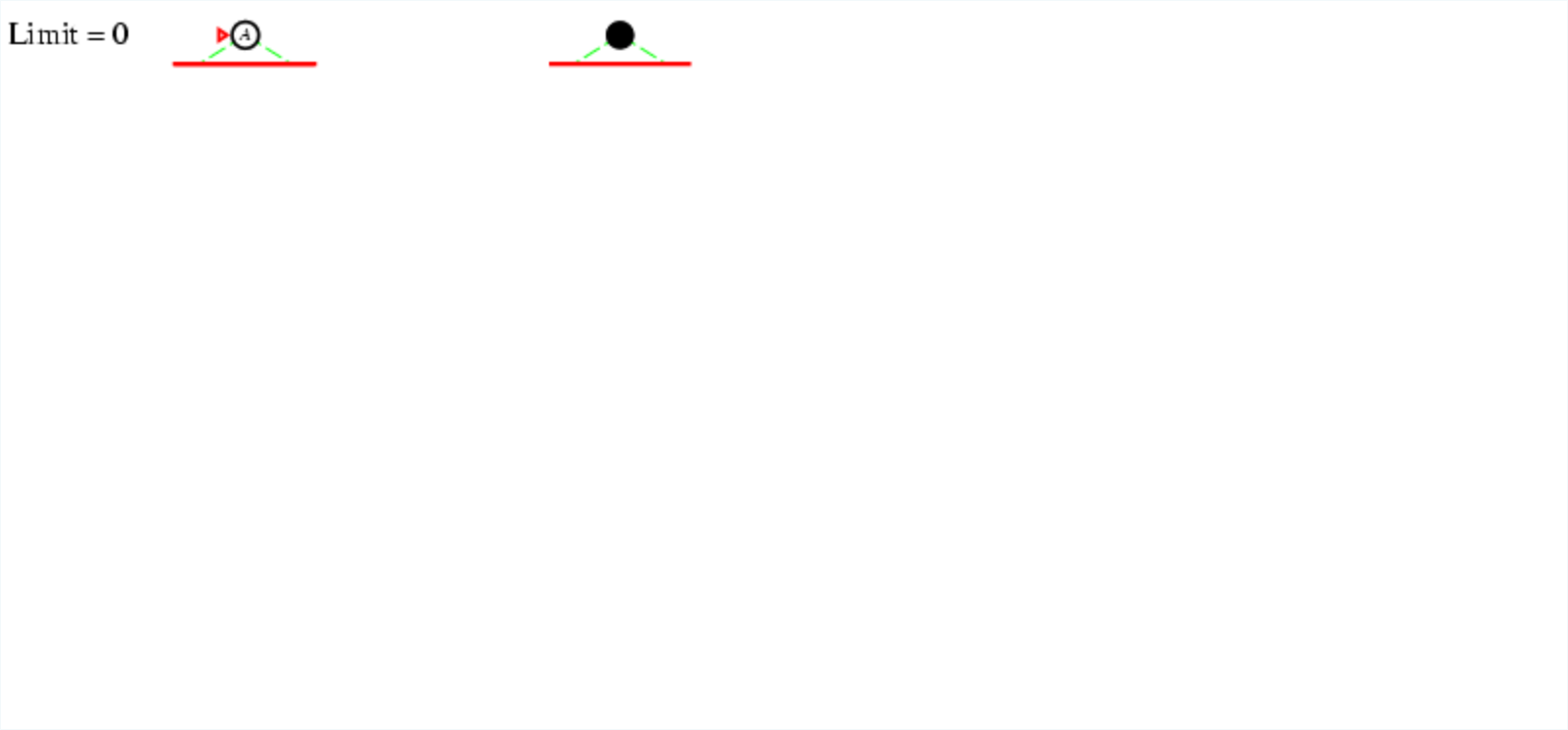
Iterative deepening combines the benefits of depth-first and breadth-first search.

Like depth-first search, its memory requirements are modest: O(bd)

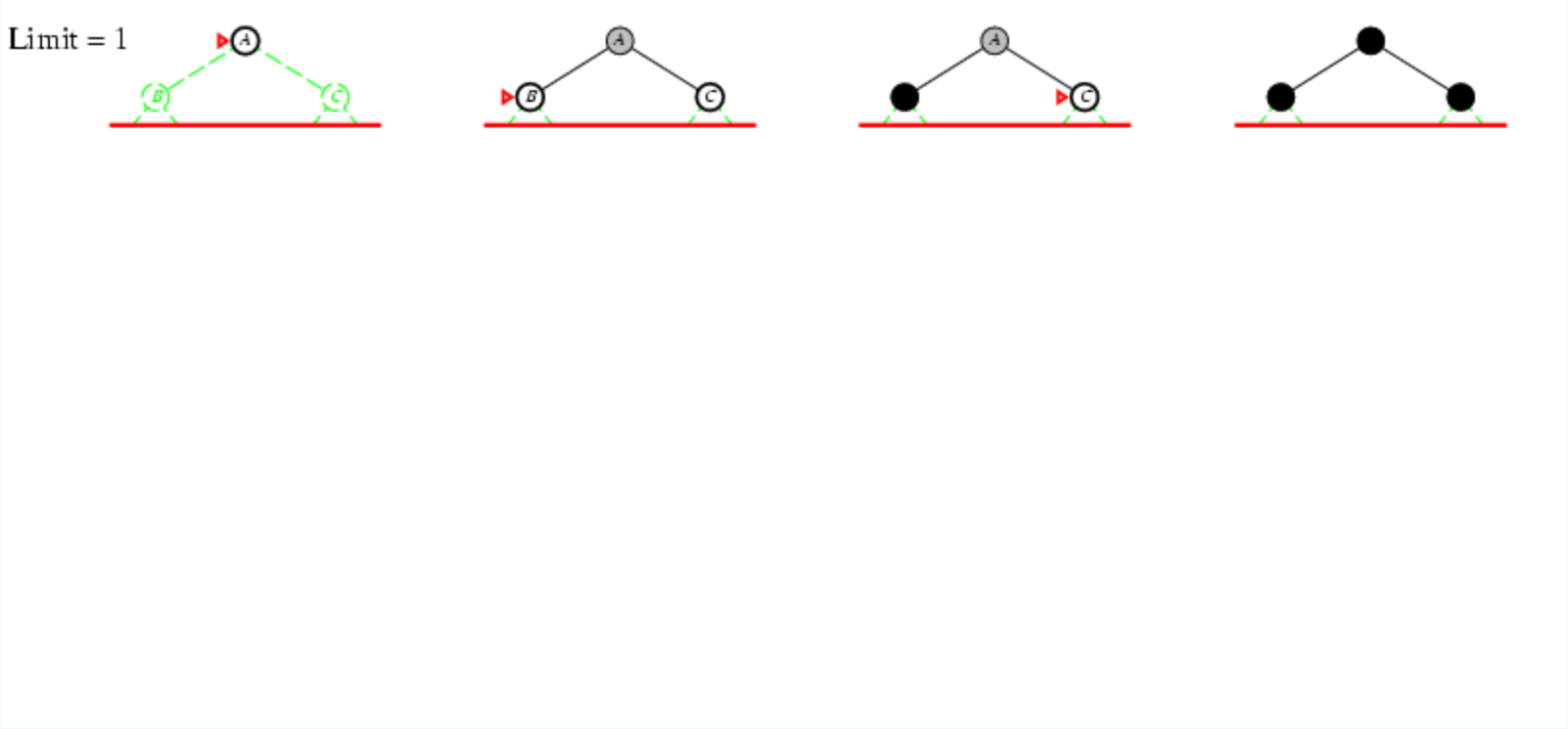
Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.

```
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
```

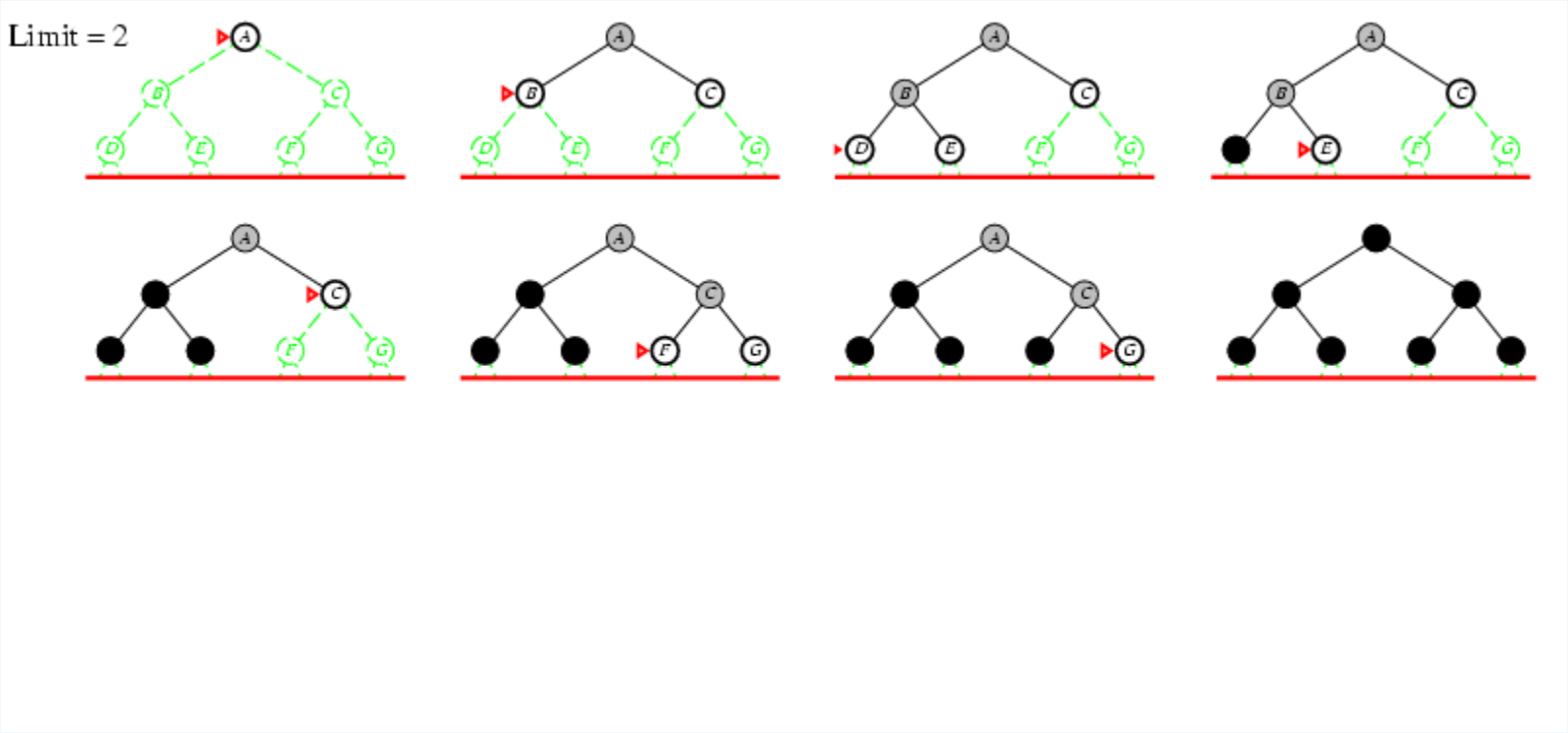
# Iterative deepening search $l = 0$



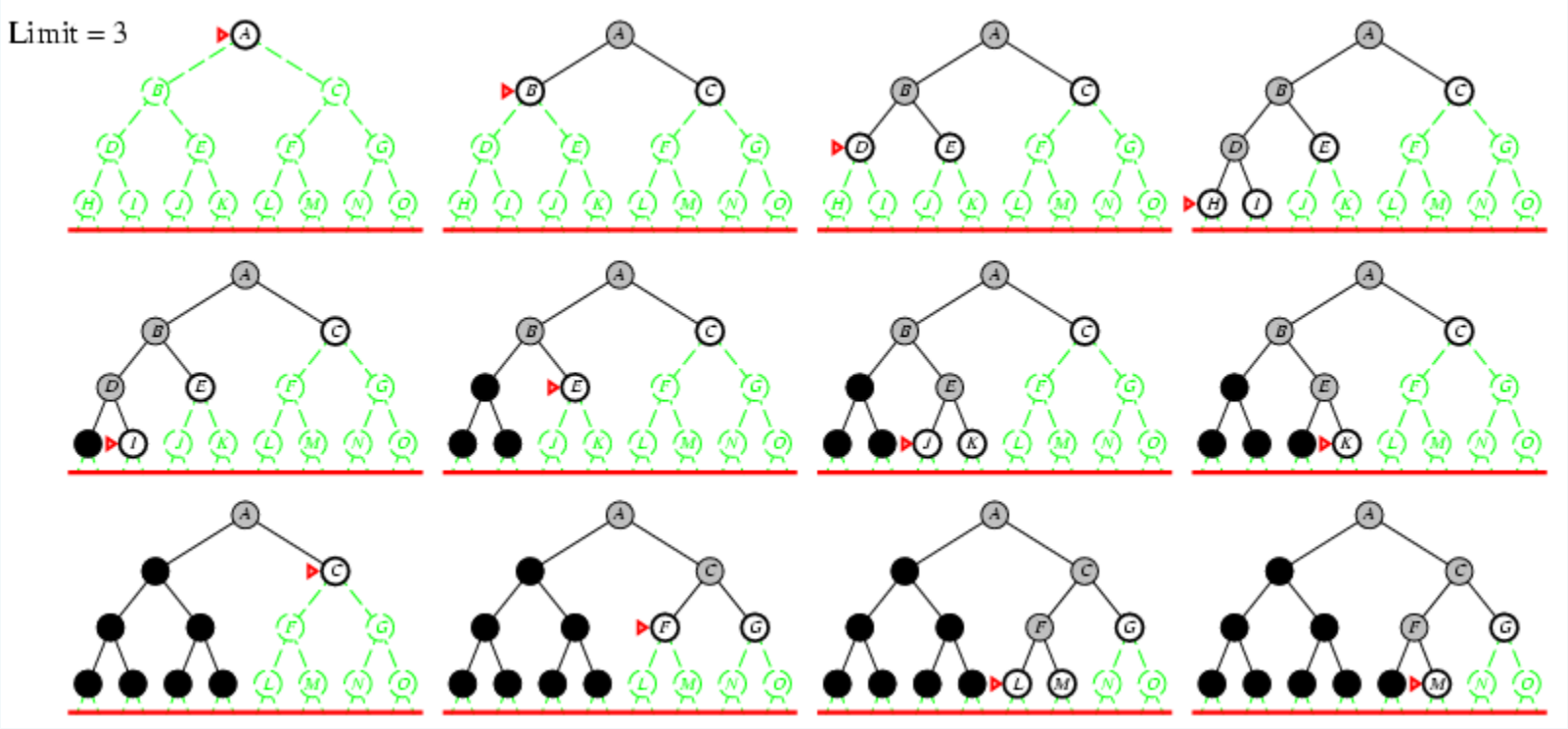
# Iterative deepening search $l = 1$



# 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*:

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

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

- For *b = 10, d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:
  - Overhead = (123,456 - 111,111)/111,111 = 11% more nodes than BFS

# 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 |