Solving Problems by Searching

Chapter 3

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

Problem Solving

- A problem can be defined formally by 4 components:-
  - **Initial state** :- describes possible situations from where problem solving begins.
  - **A set of actions** :- It is formulated using a **successor function (SF)** that describes the possible actions. It is represented by SF(x) where x is a state.

Initial state + SF $\longrightarrow$ **state space**

Problem solving …..

- **State space** :- It defines all the possible configurations of the relevant objects associated with the problem.
- **Goal Test** :- represents acceptable solutions.
- **Path Cost Function** :- A path in the state space is a sequence of states connected by a sequence of actions. A path cost function assigns a numeric cost to each path.

Problem-Solving Agents

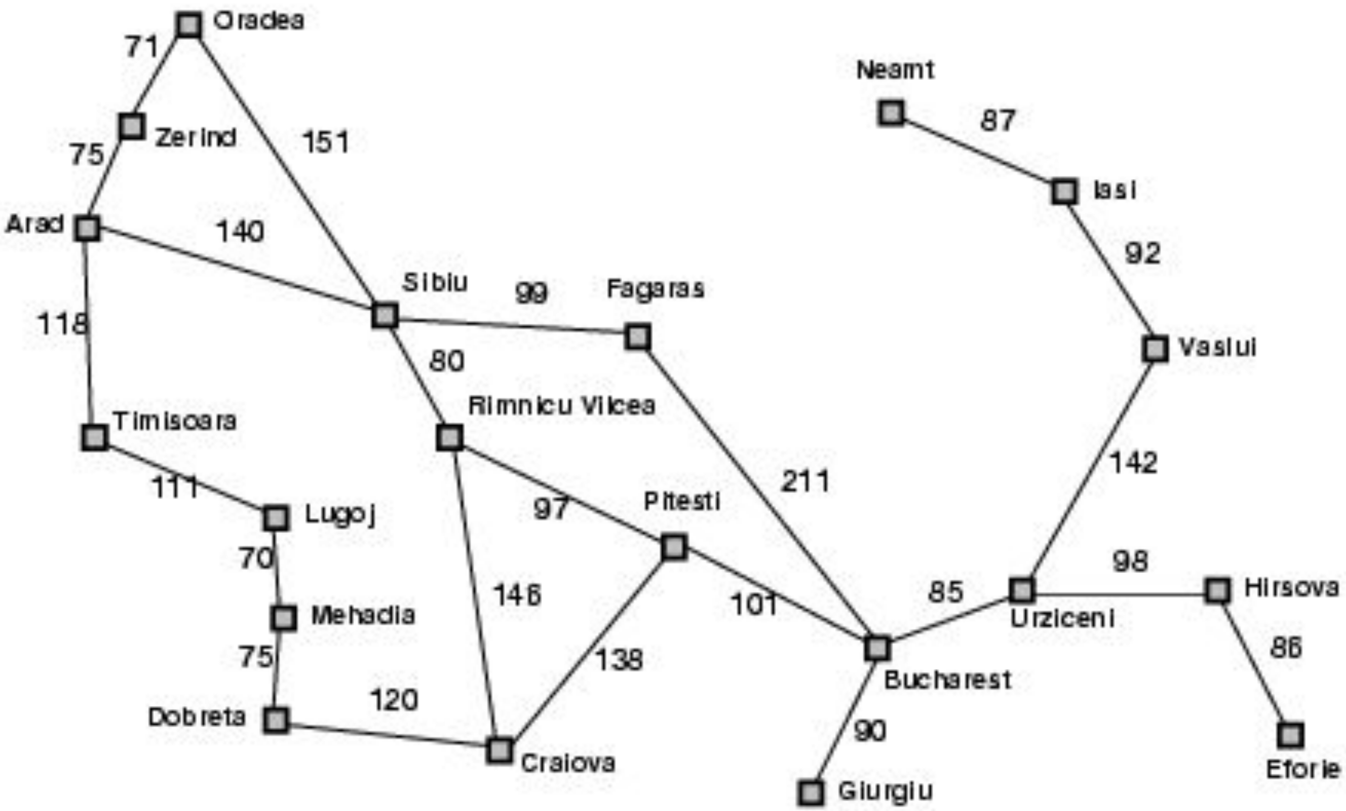
```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH( problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Example: Romania

- On holiday in Romania; currently in Arad.
- Formulate goal:
  - To be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Romania with step costs in km

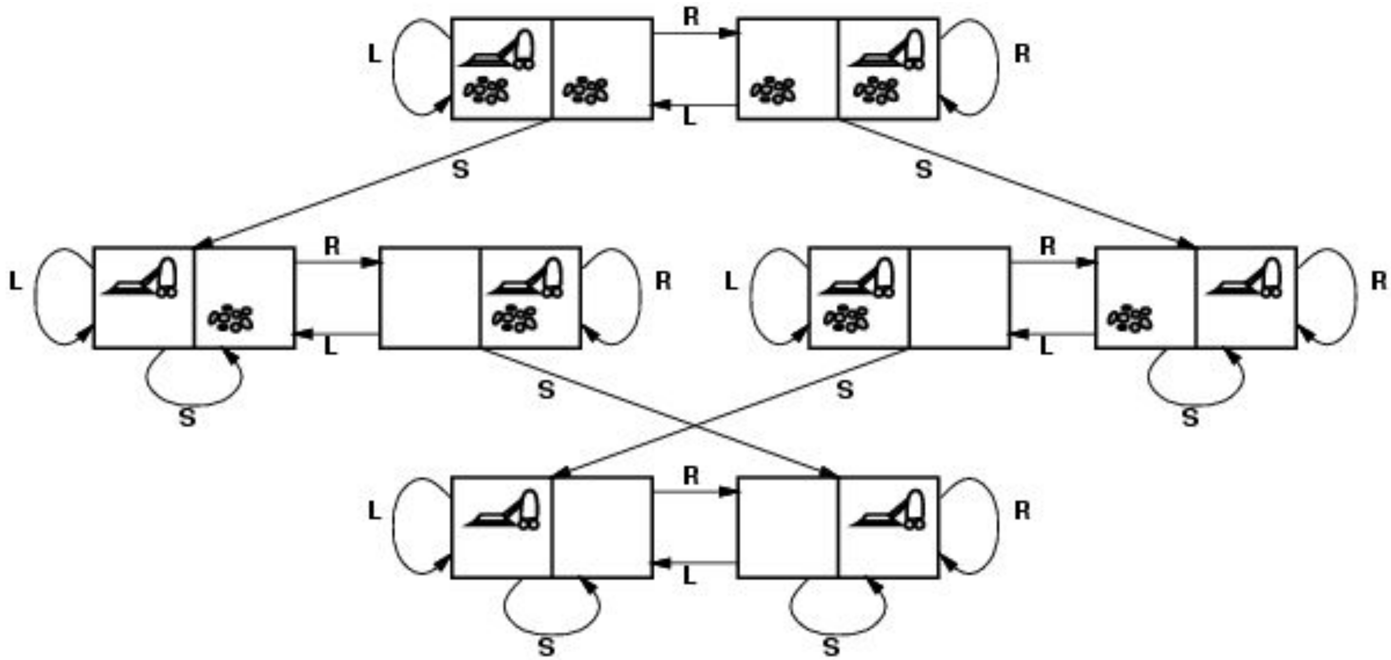


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Selecting a State Space

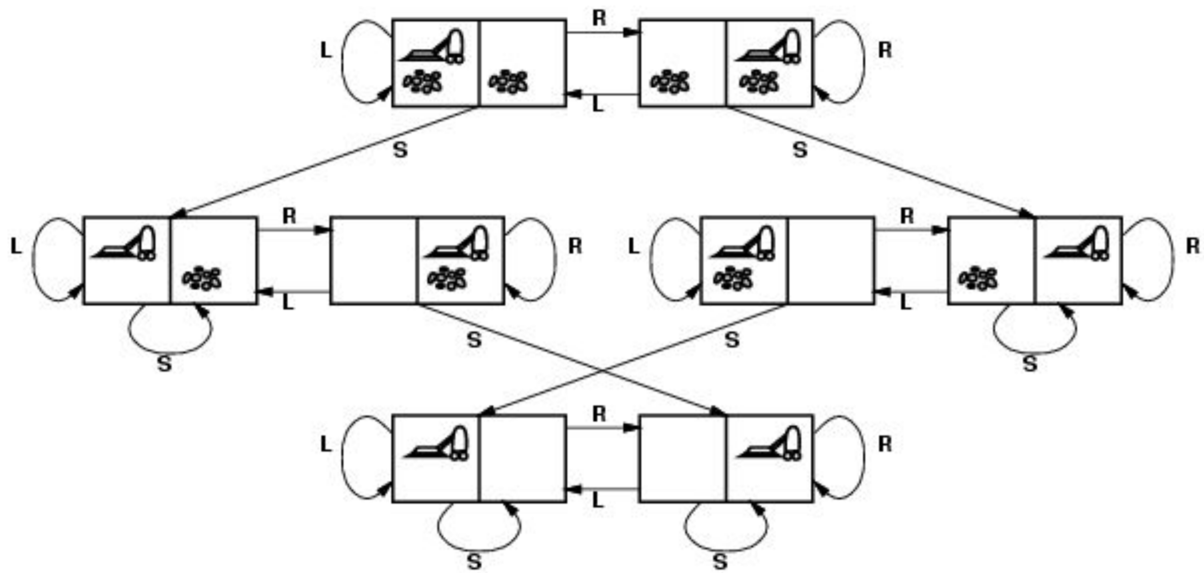
- Real world is absurdly complex
  ☐ state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad ☐ Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad‟ must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

Vacuum World State Space Graph



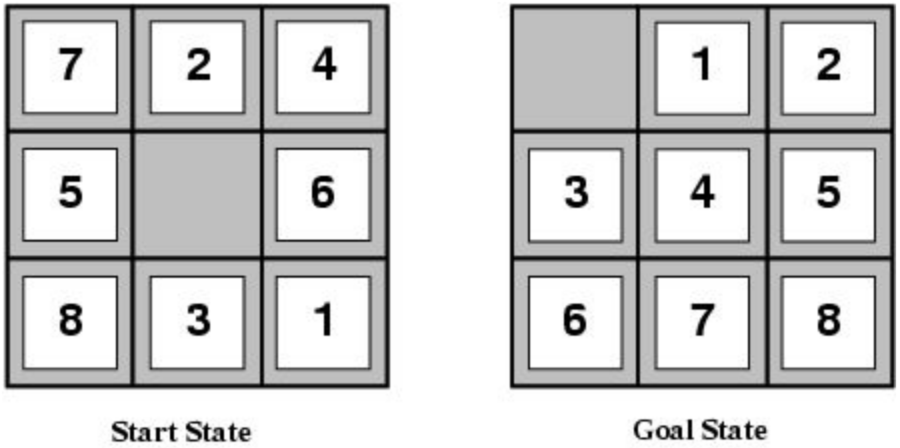
- states?
- actions?
- goal test?
- path cost?

Vacuum World State Space Graph



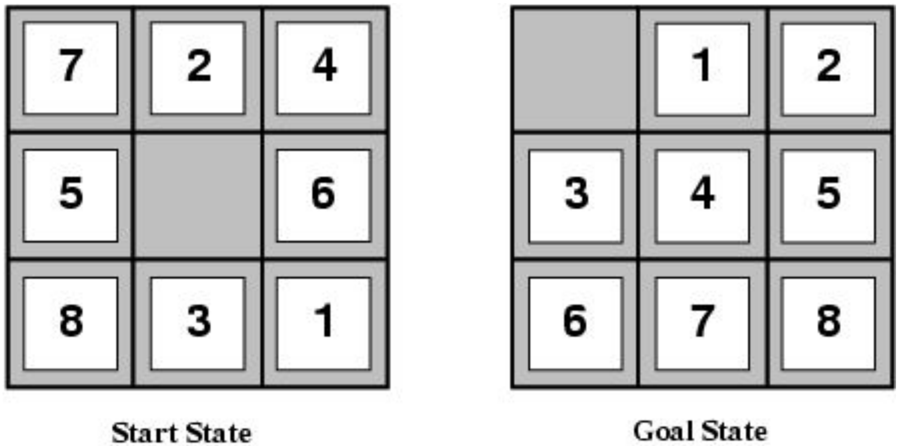
- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



- states?
- actions?
- goal test?
- path cost?

Example: The 8-puzzle



- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

Basic AI Problem Solving Techniques

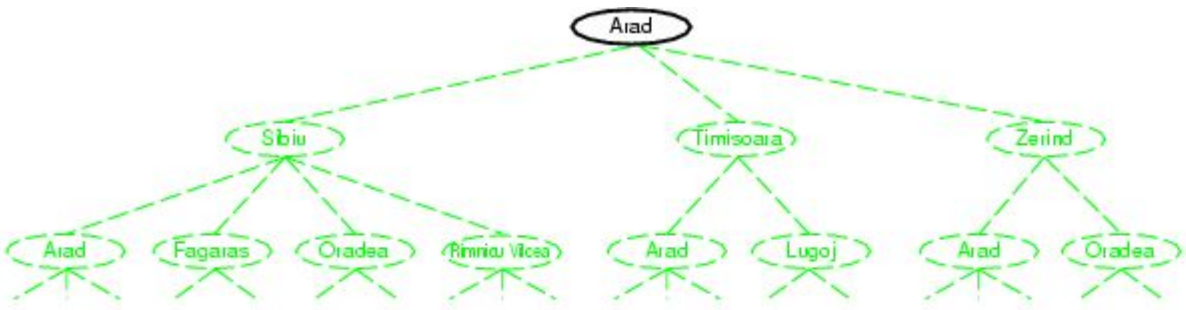
- Problem solving by Searching
- Problem solving by Reasoning / Inference
- Problem solving by Matching

Tree Search Algorithms

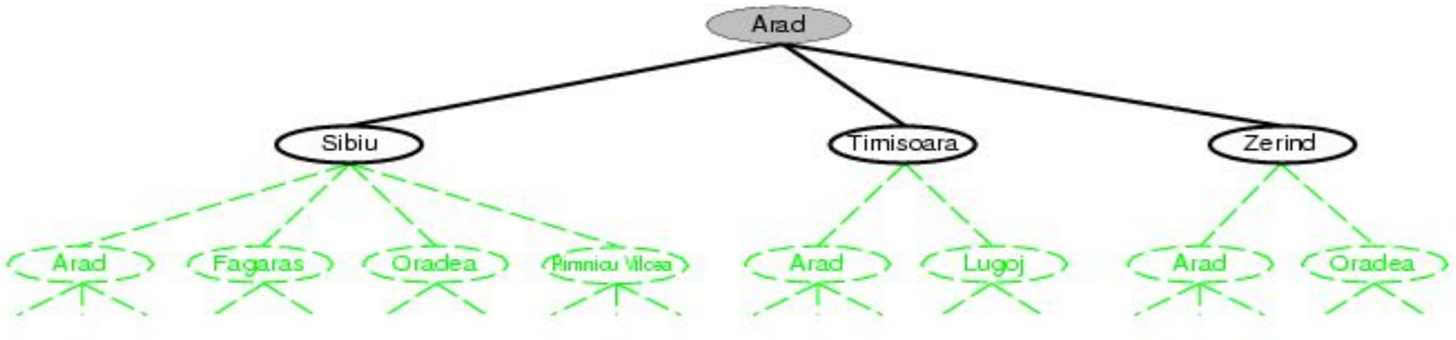
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

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
    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

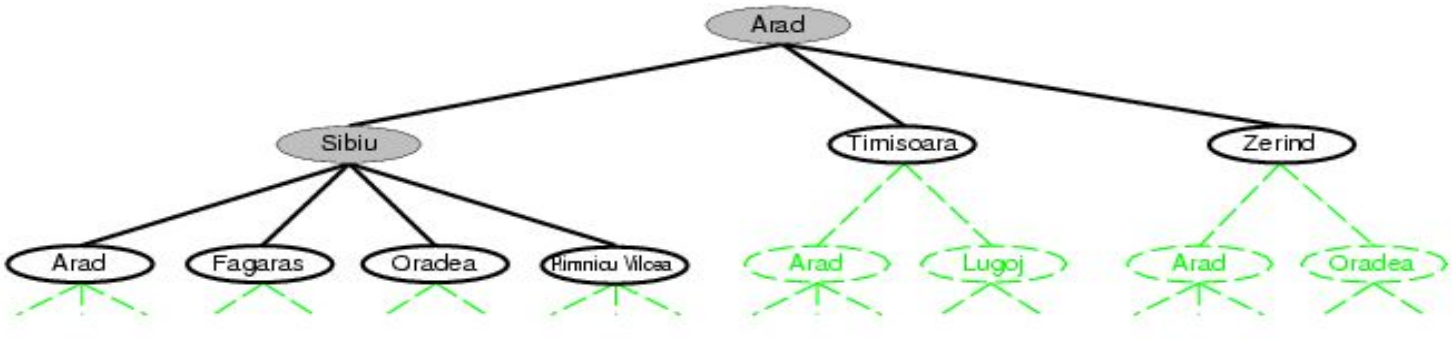
Tree search example



Tree search example



Tree search example



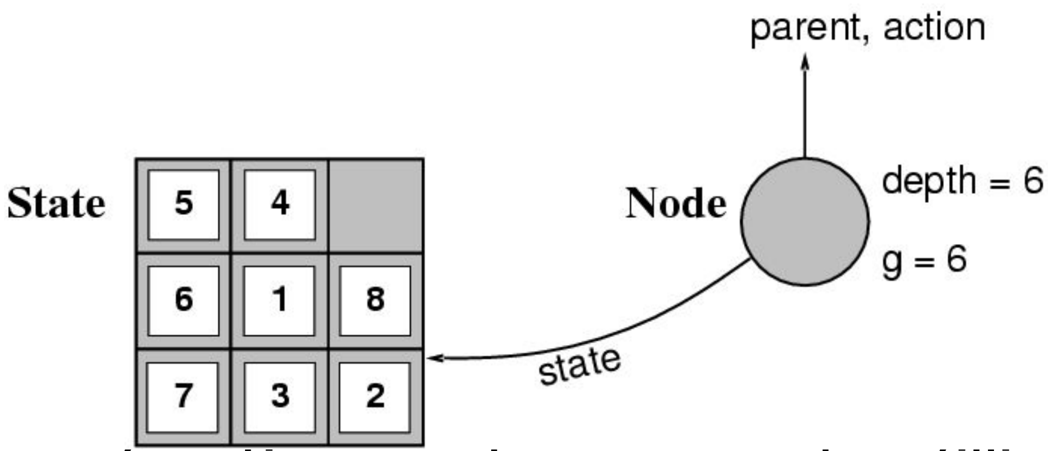
Implementation: General Tree Search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    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)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

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 includes state, parent node, action, path cost *g(x)*, depth



- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

Evaluating Search strategies

- Performance of search 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 ∞)

Search Strategies

- They are of 2 types:-
  - **Uninformed search / Blind search**
    - Blind search has no additional information about the states beyond that provided in the problem definition. All they can do is generate successor and distinguish a goal state from a non-goal state.
  - **Informed search / Heuristic search**
    - Informed search identifies whether one non-goal state is "more promising" than another one.

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

- It is a strategy in which the root node is expanded first, then all the successors of the root node are expanded, then their successors. At a given depth in the search tree all the nodes are expanded before many node at the next level is expanded.
- A FIFO queue is used i.e, new successors join the tail of the queue.

Breadth-first search algorithm

- Create a variable NODE-LIST (FIFO queue) and set it to initial state.
- Until a goal state is found or the NODE-LIST is empty,

  Do

  A. If NODE-LIST is empty then quit.

     else remove the first element frm NODE-LIST and call it V(visited)

Breadth-first search algorithm

B. For each rule (from the rule set) whose L.H.S matches with the state described in V

Do

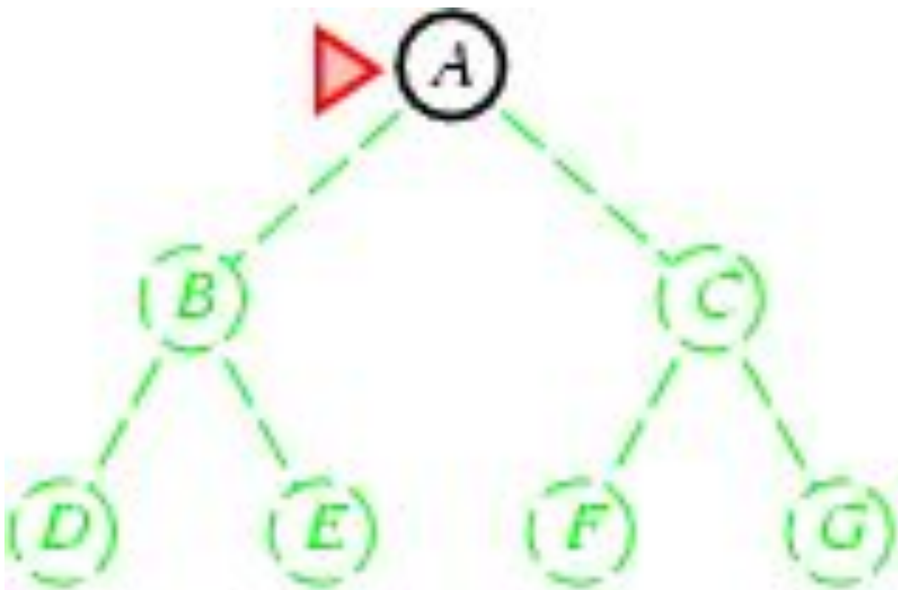
I. Apply the rule to generate a new state.

II. If the new state is a goal state then quit and return the state.

III. Otherwise add the new state to the end of the NODE-LIST.

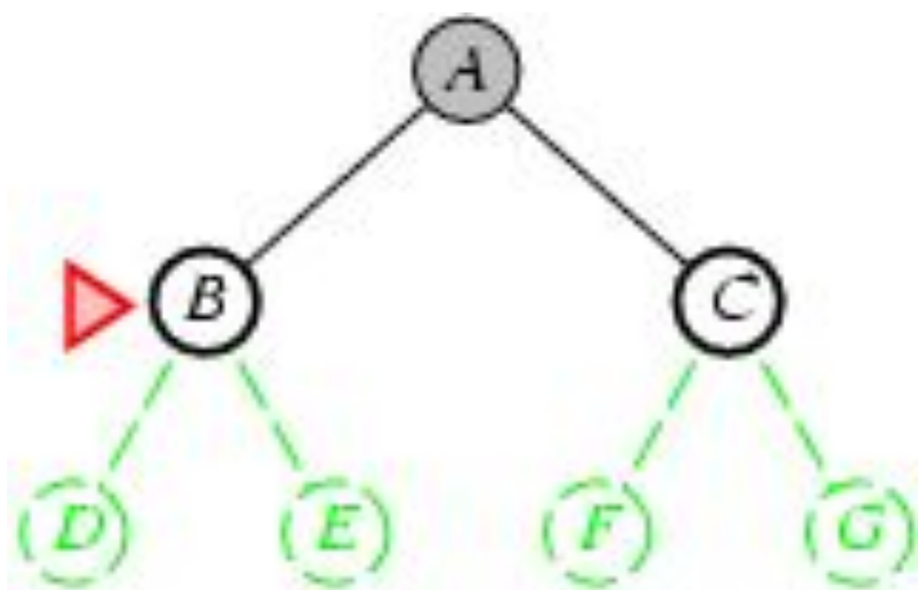
Breadth-first search

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



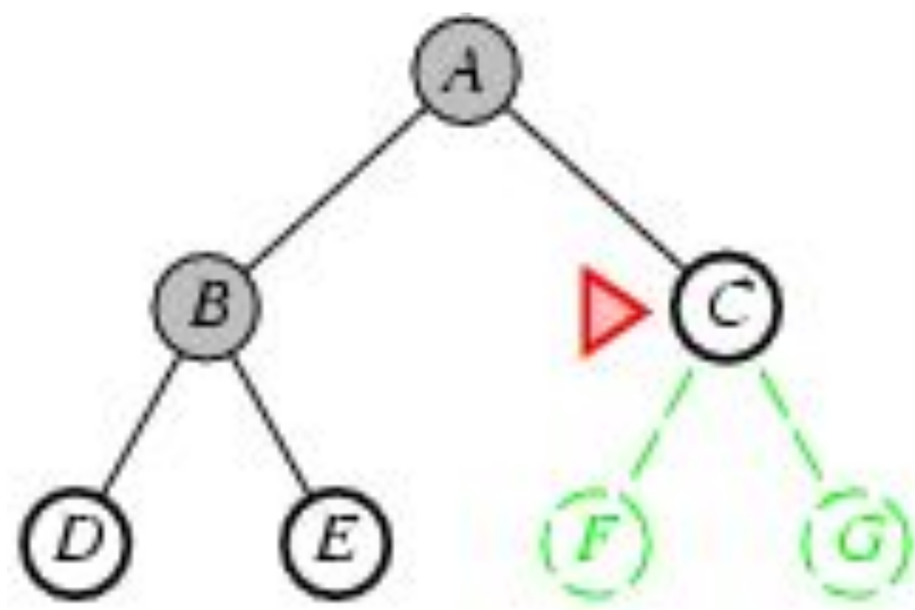
Breadth-first search

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



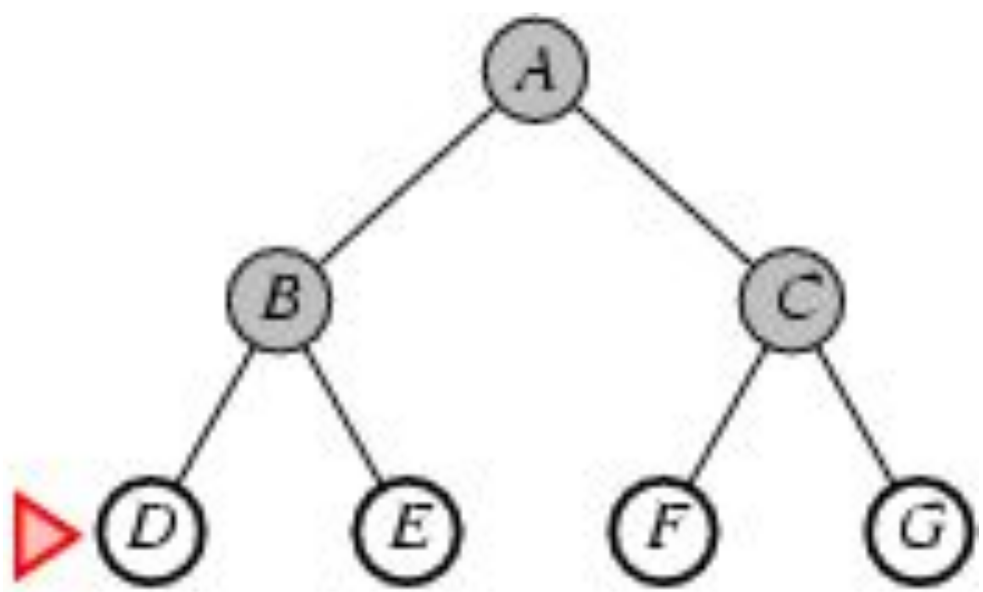
Breadth-first search

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



Breadth-first search

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



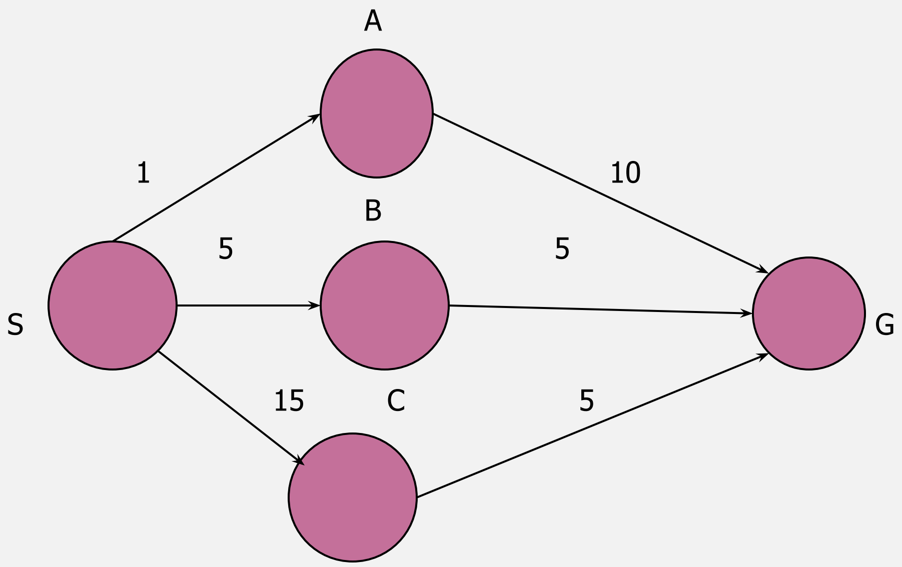
Properties of breadth-first search

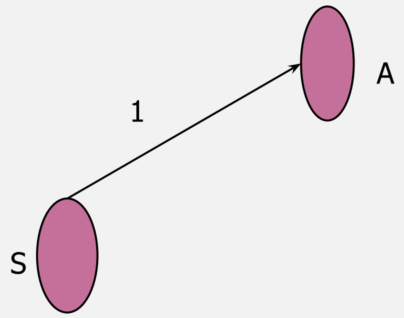
- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d$ = $O(b^d)$
- Space? $O(b^d)$ (keeps every node in memory)
- Optimal? Yes (if path cost is a non-decreasing function of depth i.e path cost = 1 per step)
- Space is the bigger problem (more than time)

Uniform-cost search

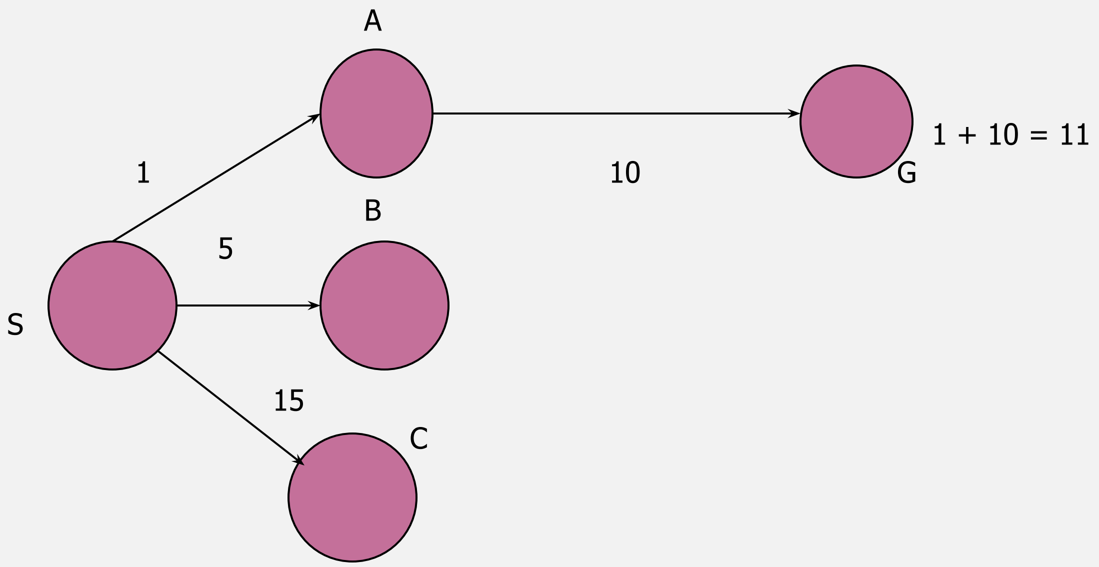
- Breadth- first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node. **Uniform – cost search expands the node n with the lowest path cost**. **Uniform-cost search doesn’t care about the number of steps a path has, but only about their total cost.**
- Drawback :-
  - It will get stuck in an infinite loop if it ever expands a node that has a zero-cost action leading back to the same state. (for example a noop action) . So, completeness is guaranteed if cost of every step is $>= \epsilon$, then it is optimal also.
  - Worst case time and space complexity is very high .

Uniform-cost search

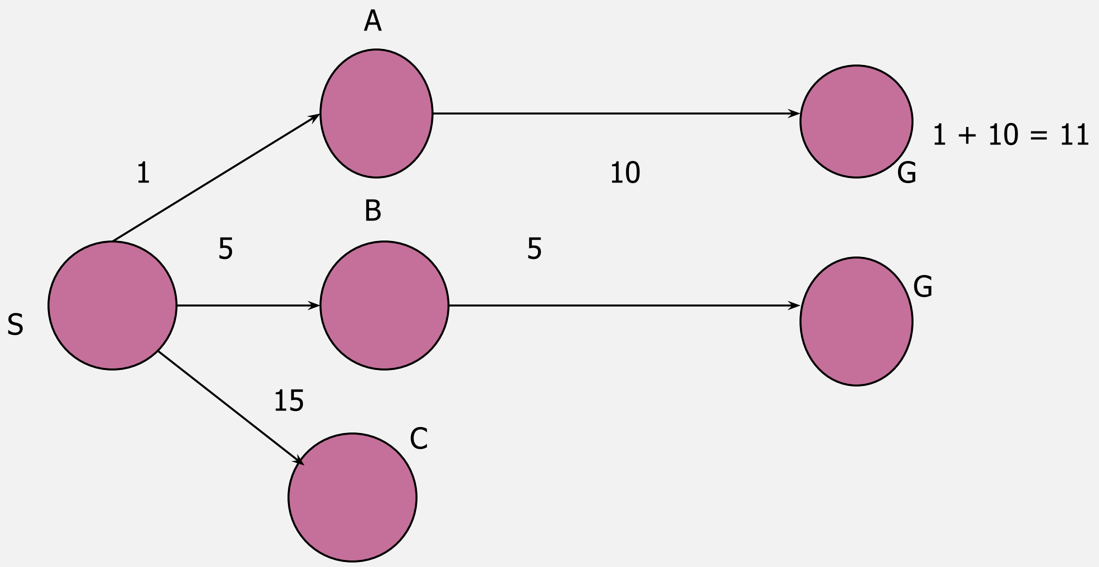




Uniform-cost search



Uniform-cost search



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(DFS)

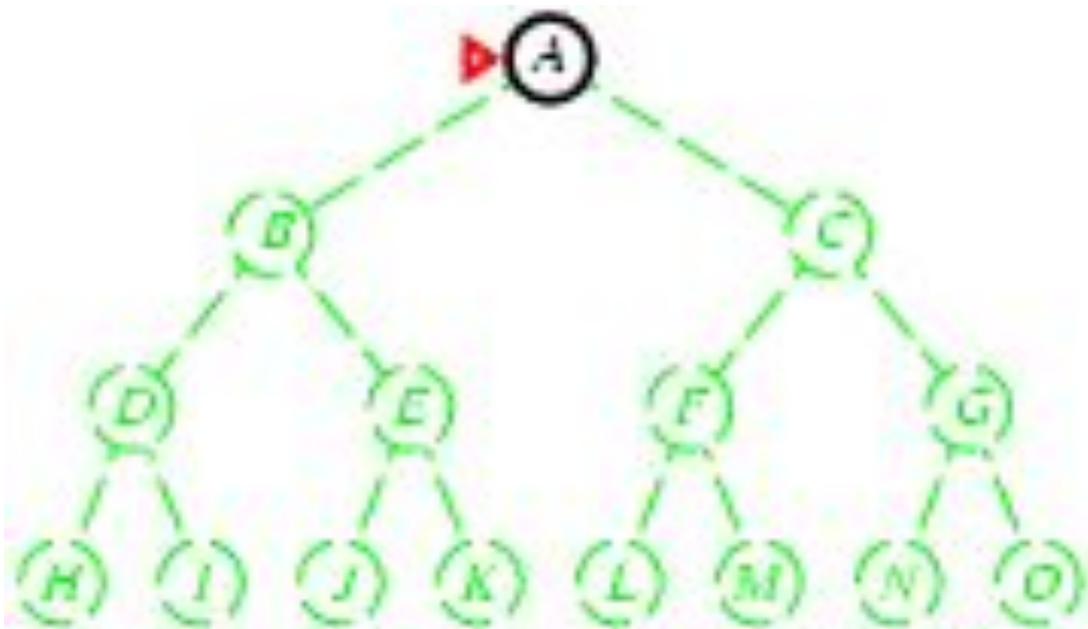
- It is a strategy that expands the deepest node in the search tree.
- Here a stack is used.

Depth-first search(DFS)

1. Form a one element stack consisting of the root node.
2. Until the stack is empty or the goal node is found out, repeat the following:-
   1. Remove the first element from the stack. If it is the goal node announce success, return.
   2. If not, add the first element's children if any, to the top of the stack.
3. If the goal node is not found announce failure.

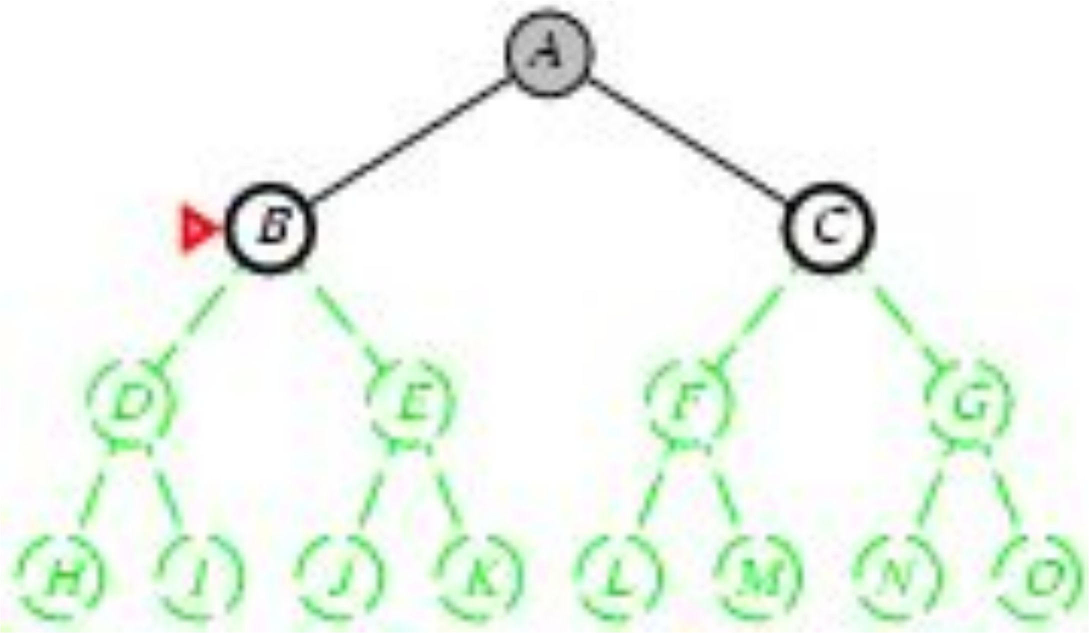
Depth-first search

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



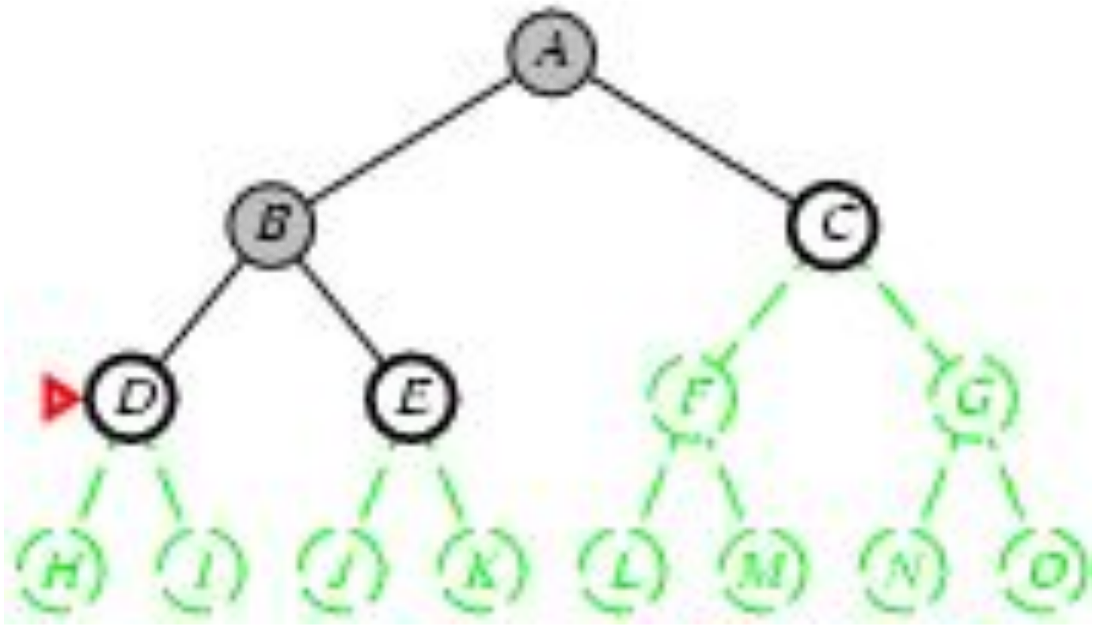
Depth-first search

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



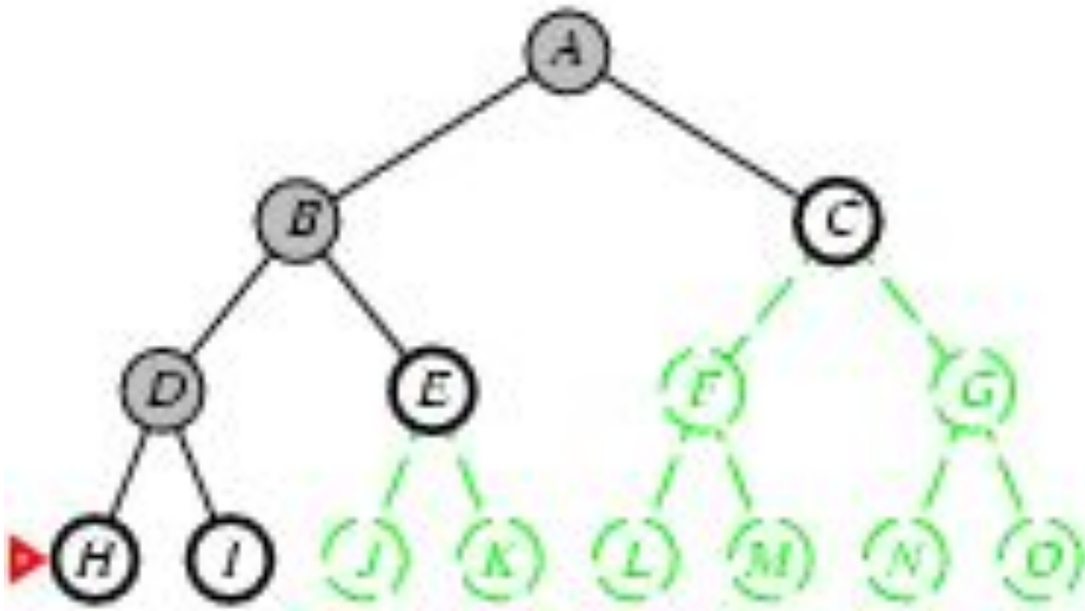
Depth-first search

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



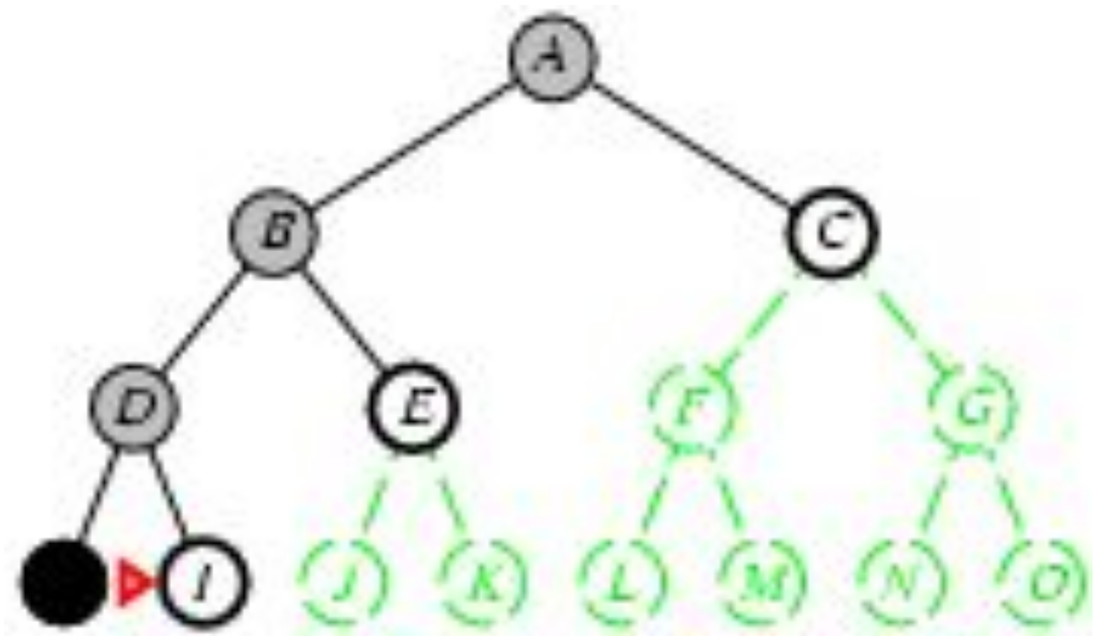
Depth-first search

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



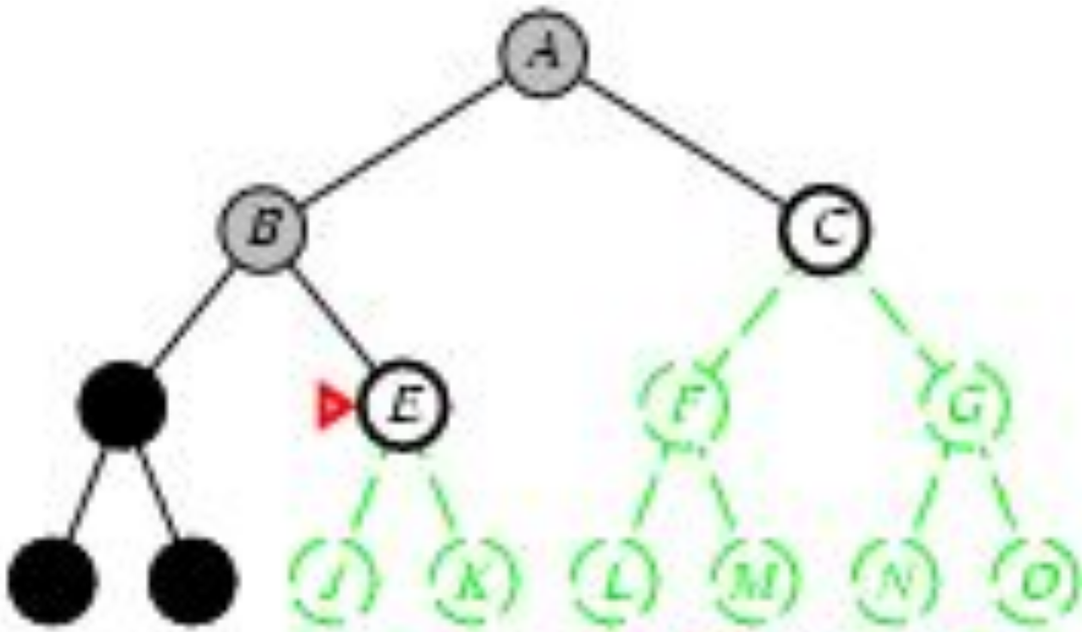
Depth-first search

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



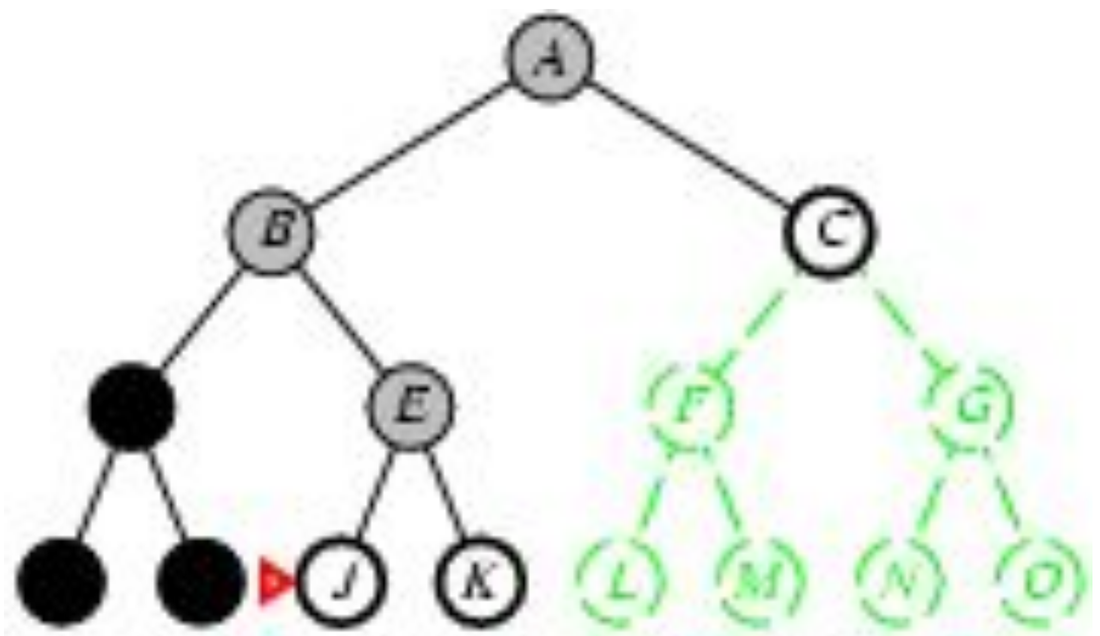
Depth-first search

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



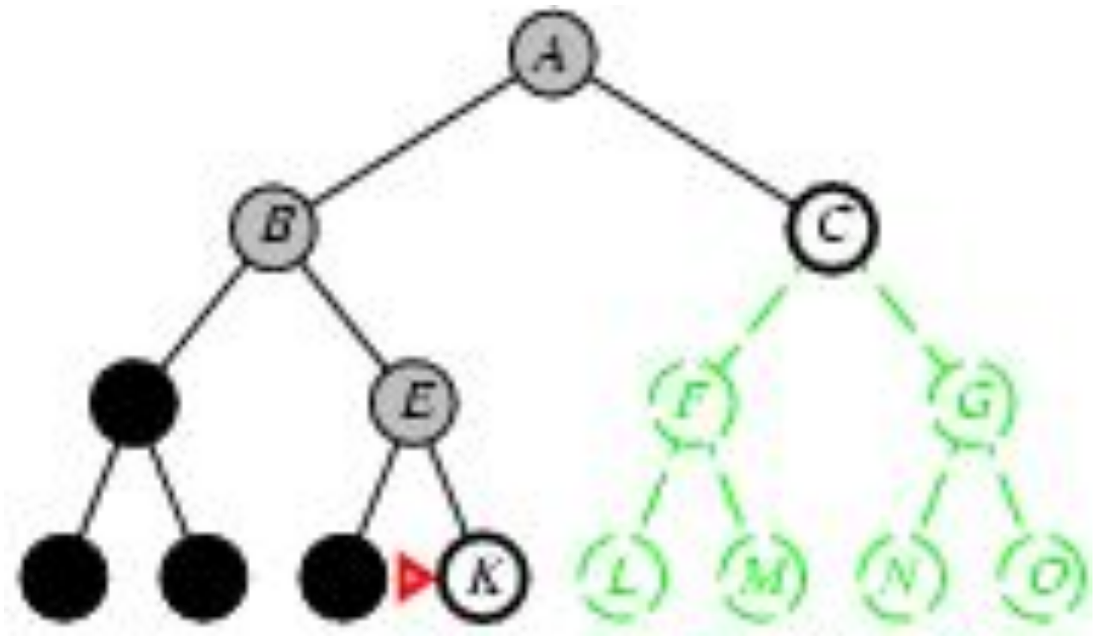
Depth-first search

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



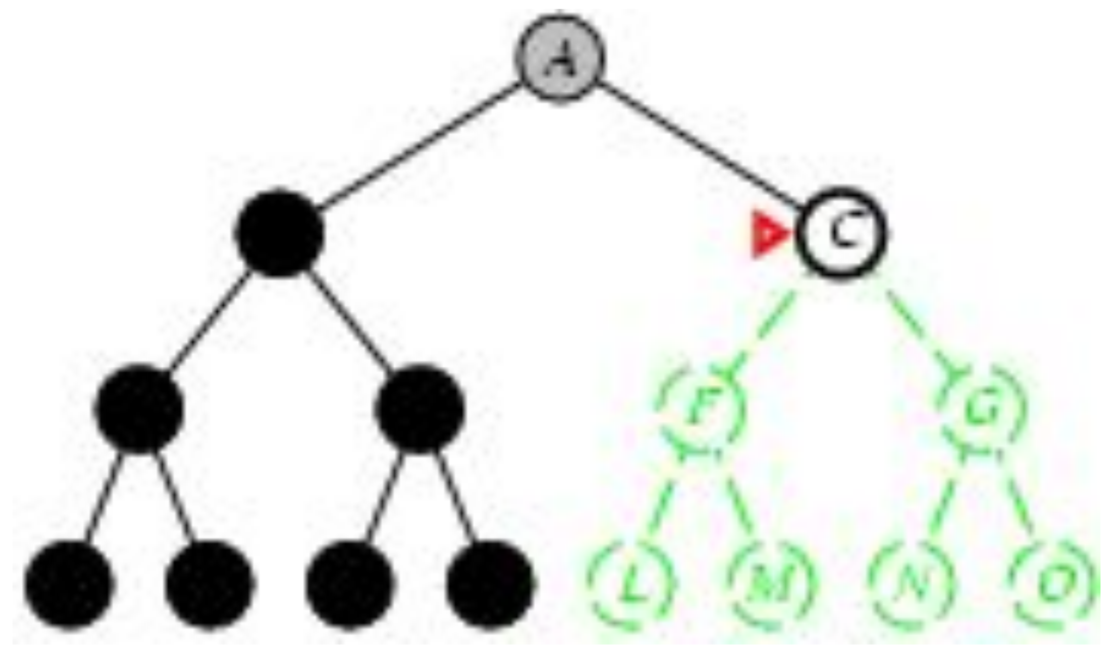
Depth-first search

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



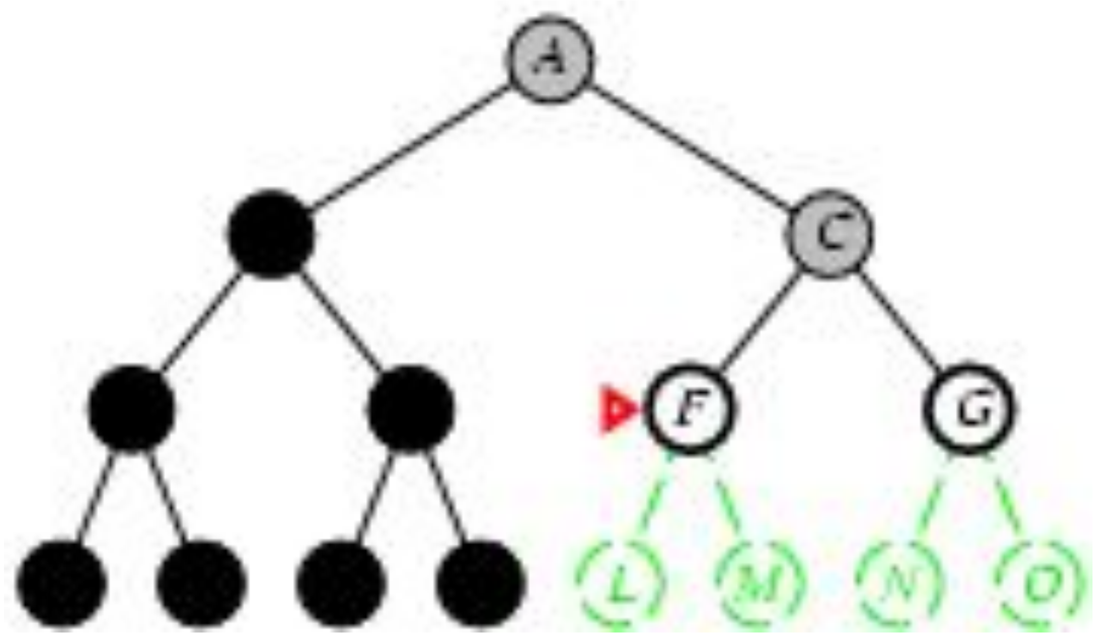
Depth-first search

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



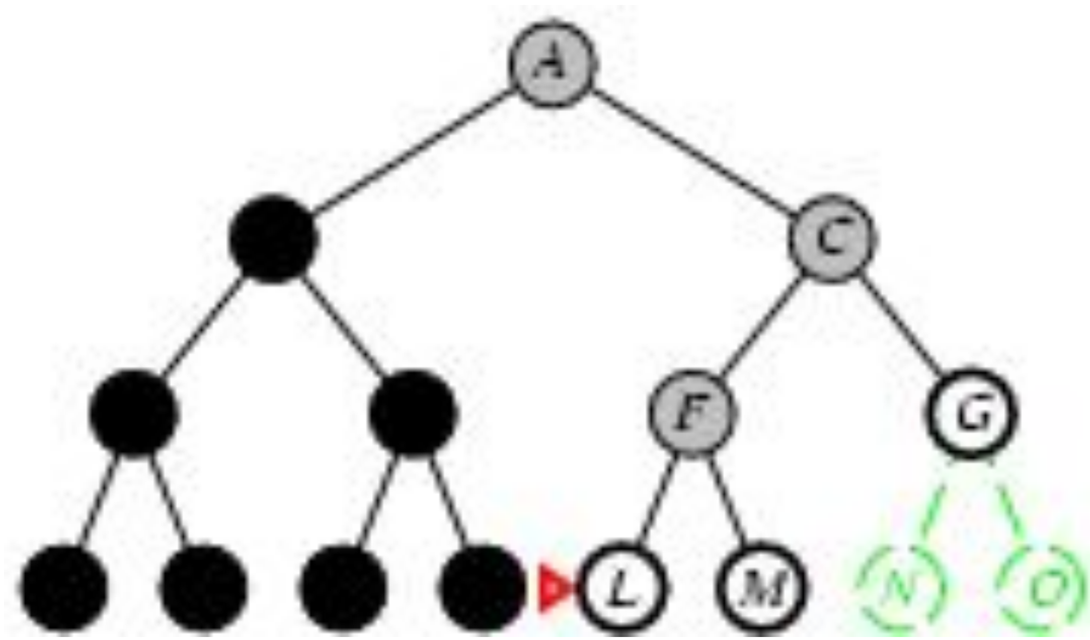
Depth-first search

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



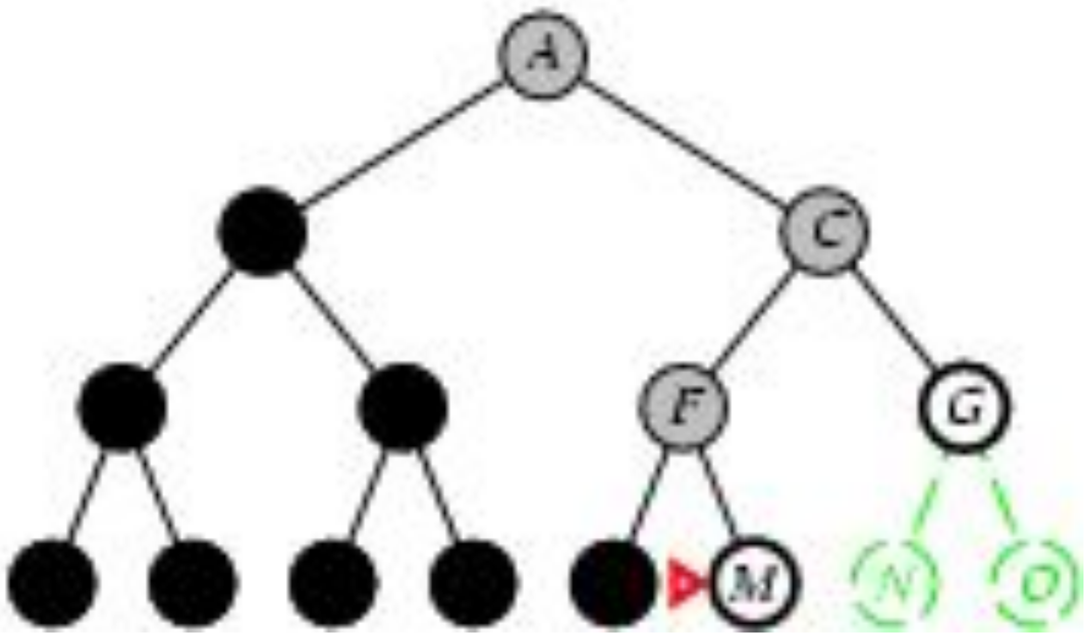
Depth-first search

- 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
  - We may modify the algorithm to avoid repeated states along the path
    DFS is complete in finite spaces.
- Time? The number of nodes generated is of the order of $O(b^m)$. It is terrible if m is much larger.
  - but if solutions are dense, may be much faster than breadth-first
- Space? It only stores the unexpanded nodes. So it requires 1 +b + b +b ….. m times = $O(bm)$, i.e., linear space!
- Optimal? No. If it makes a wrong choice, it may go down a very long path and finds a solution, where as there may be a better solution at a higher level in the tree.

Depth-limited search

- The problem with DFS is that the search can go down an infinite branch and thus never return. Depth limited search avoids this problem by avoiding imposing a depth limit l which effectively terminates the search at that depth. That is , nodes at depth l are treated as if they have no successors.
- The choice of depth parameter l is an important factor. If l is too deep , it is wasteful in terms of both time and space. But if l < d (the depth at which solution exists) i,e the shallowest goal is beyond the depth limit, then this algorithm will never reach a goal state.

Depth-limited search

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

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

Recursive implementation:

```
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
```

Properties of depth- limited search

- Complete? If l >= d , then it is.
- Time? $O(b^{l})$.
- Space? $O(bl)$, i.e., linear space.
- Optimal? No.

Iterative deepening search

- The problem with depth-limited search is deciding on a suitable depth parameter, which is not always easy.
- To overcome this problem there is another search called iterative deepening search.
- This search method simply tries all possible depth limits; first 0, then 1, then 2 etc. until a solution is found.
- Iterative deepening combines the benefits of DFS and BFS. It may appear wasteful as it is expanding nodes multiple times. But the overhead is small in comparison to the growth of an exponential search tree.

Iterative deepening search

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

    inputs: *problem*, a problem

    for *depth* $\leftarrow$ 0 **to** ∞ **do**

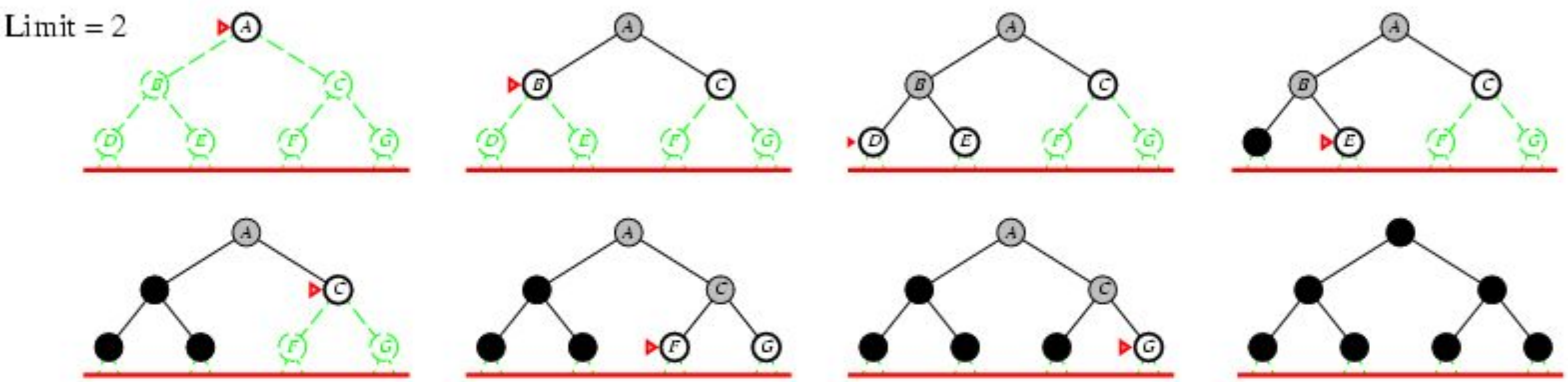
        result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)

        if *result* $\neq$ cutoff **then return** *result*

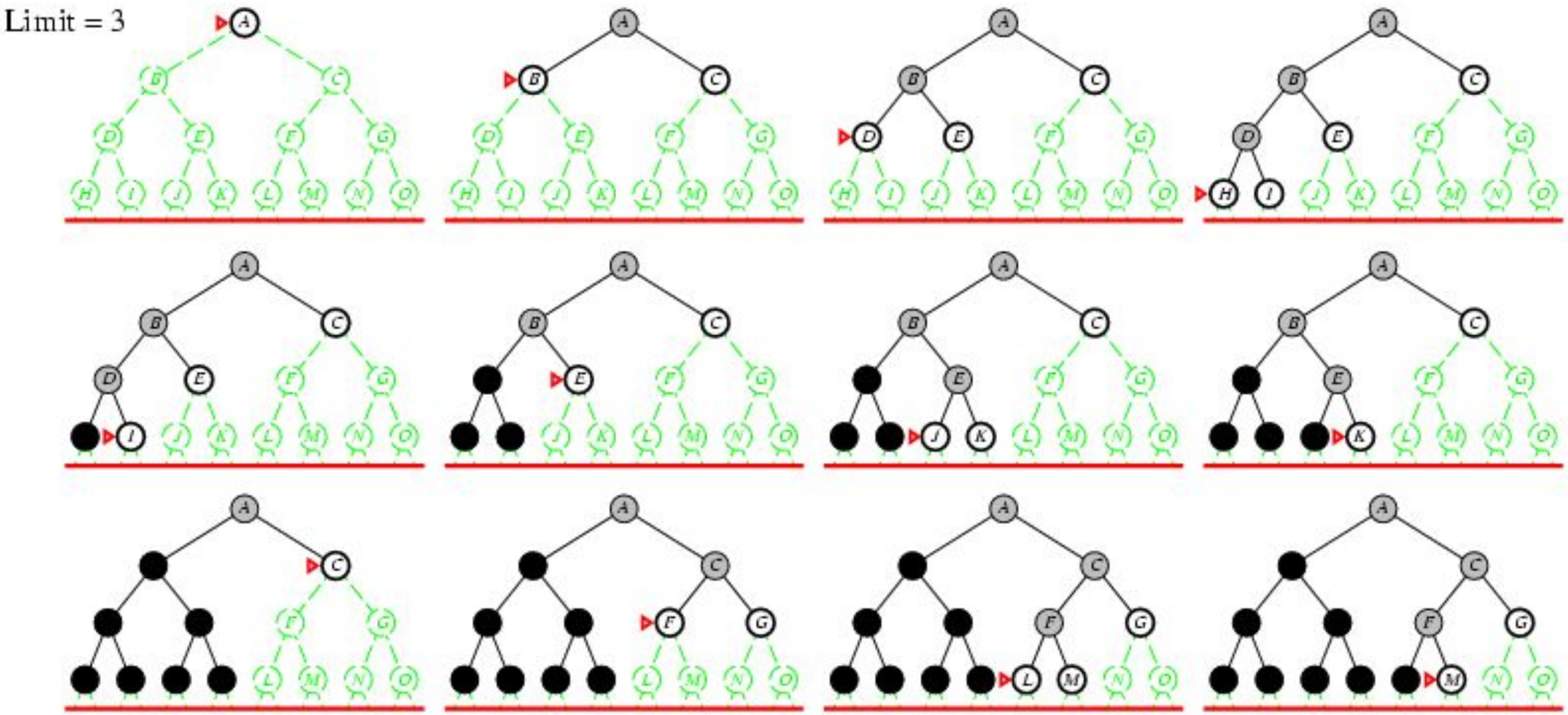
Iterative deepening search l =0

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*:

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

as the nodes at the bottom level d are expanded once, the nodes at (d-1) are expanded twice, those at (d – 3) are expanded three times and so on back to the root.

Iterative deepening search

- 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%

we can see that compared to the overall number of expansions , the total is not substantially increased.

Properties of iterative deepening search

- Complete? Yes , like BFS it is complete when b is finite.
- Time? $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? O(bd) , like DFS memory requirement is linear.
- Optimal? Yes, if like BFS path cost is a non-decreasing function of the depth of the node, it is optimal.

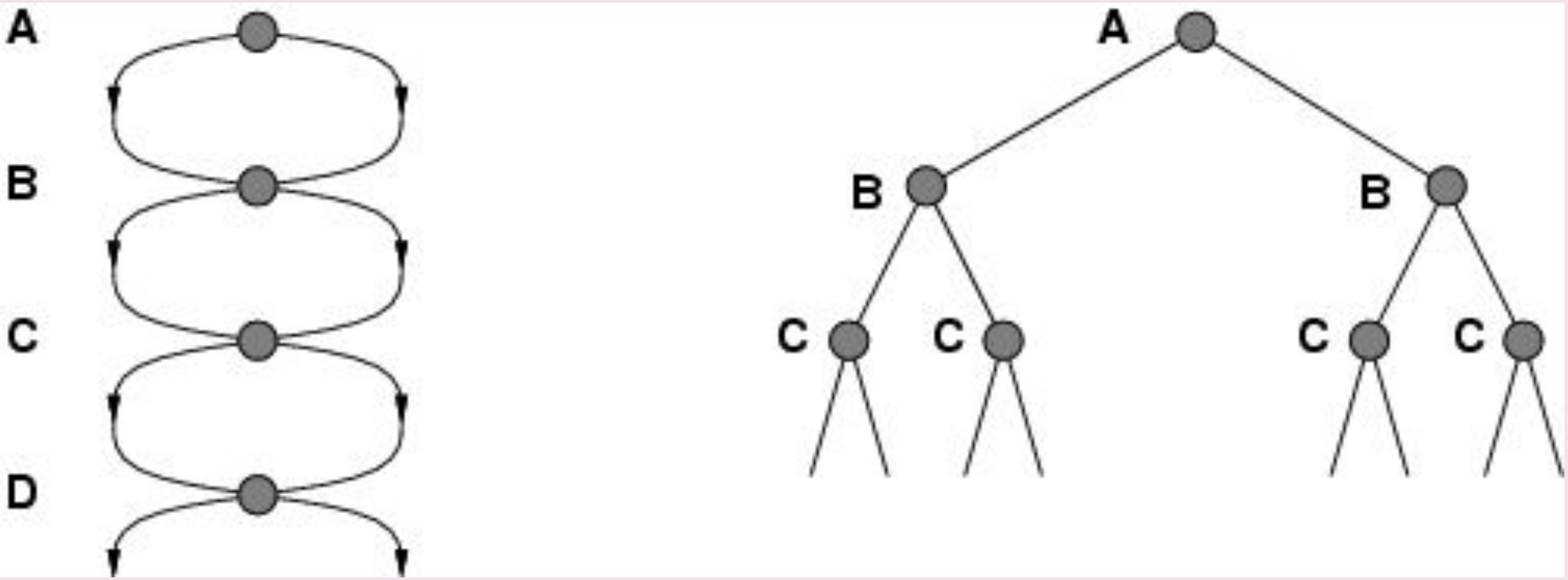
  In general iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

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!



Repeated states

- In most searching algorithm, the state space generates an exponentially larger search tree. This occurs mainly because of repeated nodes. If we can avoid the generation of repeated nodes we can limit the number of nodes that are created and stop the expansion of repeated nodes.

Repeated states

- In most searching algorithm, the state space generates an exponentially larger search tree. This occurs mainly because of repeated nodes. If we can avoid the generation of repeated nodes we can limit the number of nodes that are created and stop the expansion of repeated nodes. There are three methods having increasing order of computational overhead to control the generation of repeated nodes:-

Repeated states

1. Don't generate a node that is the same as the parent node.
2. Don't create paths with cycles in them. To do this we can check each ancestor node and refuse to create a state that is the same as this set of nodes.
3. Don't generate any state that is the same as any state generated before. This requires that every state is kept in memory. (space complexity is $O(b^d)$)

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