## Uninformed Search Algorithms

- These algorithms are given no information about the problem other than its definition.
- They are also called blind search algorithms. It means that the strategies have no additional information about states beyond that provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded.
- Although some of these algorithms can solve any solvable problem, none of them can do so efficiently.
- E.g., Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Depth First Search, Bidirectional Search.
- Key features of uninformed search algorithms:
  - Systematic exploration – uninformed search algorithms explore the search space systematically, either by expanding all children of a node (e.g. BFS) or by exploring as deep as possible in a single path before backtracking (e.g. DFS).
  - No heuristics – uninformed search algorithms do not use additional information, such as heuristics or cost estimates, to guide the search process.
  - Blind search – uninformed search algorithms do not consider the cost of reaching the goal or the likelihood of finding a solution, leading to a blind search process.
  - Simple to implement – uninformed search algorithms are often simple to implement and understand, making them a good starting point for more complex algorithms.
  - Inefficient in complex problems – uninformed search algorithms can be inefficient in complex problems with large search spaces, leading to an exponential increase in the number of states explored.

## Informed Search Algorithms

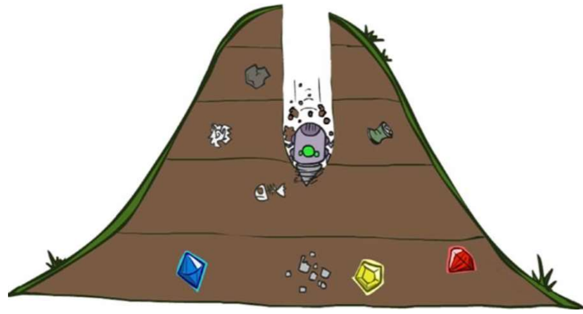
- These algorithms apply the strategies that know whether one non-goal state is “more promising” than another. They are also called heuristic search strategies.
- They can do quite well given some guidance on where to look for solutions.
- E.g., Hill Climbing, Best First Search, A* and AO* Algorithm, Constraint satisfaction, Means Ends Analysis, Minimax Search, Alpha-Beta Cut offs etc.
- Key features of informed search algorithms in AI:
  - Use of Heuristics – informed search algorithms use heuristics, or additional information, to guide the search process and prioritize which nodes to expand.
  - More efficient – informed search algorithms are designed to be more efficient than uninformed search algorithms, such as breadth-first search or depth-first search, by avoiding the exploration of unlikely paths and focusing on more promising ones.
  - Goal-directed – informed search algorithms are goal-directed, meaning that they are designed to find a solution to a specific problem.
  - Cost-based – informed search algorithms often use cost-based estimates to evaluate nodes, such as the estimated cost to reach the goal or the cost of a particular path.
  - Prioritization – informed search algorithms prioritize which nodes to expand based on the additional information available, often leading to more efficient problem-solving.
  - Optimality – informed search algorithms may guarantee an optimal solution if the heuristics used are admissible (never overestimating the actual cost) and consistent (the estimated cost is a lower bound on the actual cost).

# Depth First Search (DFS)

**Source:**

1. Stuart Russell & Peter Norvig, "Artificial Intelligence : A Modern Approach", Pearson Education, 2nd Edition.
2. Elaine Rich and Kevin Knight, "Artificial Intelligence" Tata McGraw Hill.

## Depth First Search (DFS) (1)



- Depth-first search (DFS) always expands the deepest node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.
- DFS uses a stack (LIFO queue) for frontiers. A LIFO sequence means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.
- Advantages:
  – DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
  – It takes less time to reach to the goal node than Breadth-First Search (BFS) algorithm (if it traverses in the right path).
- Disadvantages:
  – There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
  – DFS algorithm goes for deep down searching and sometimes it may go to the infinite loop.

## DFS (2)

- **DFS on a binary tree:**

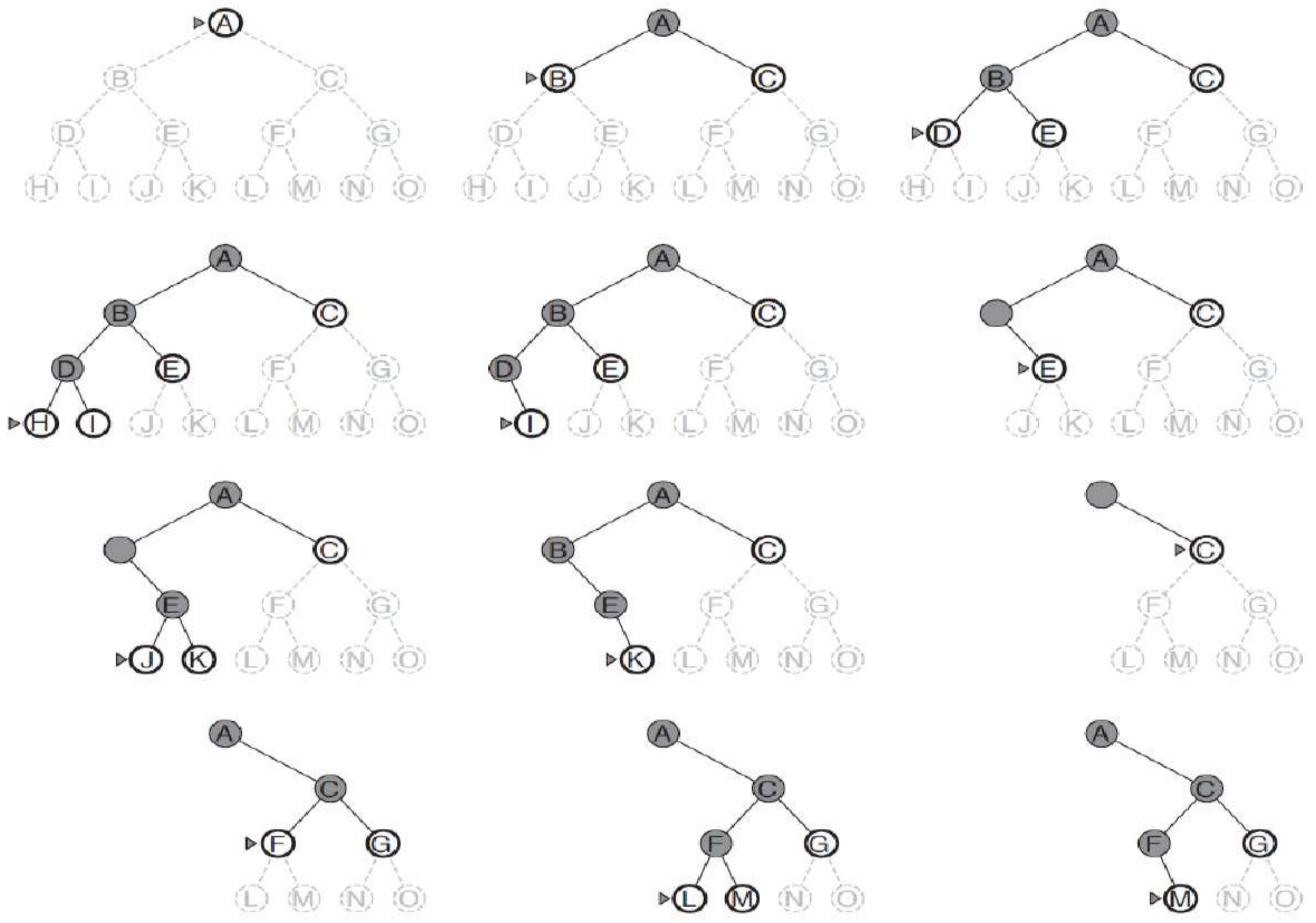
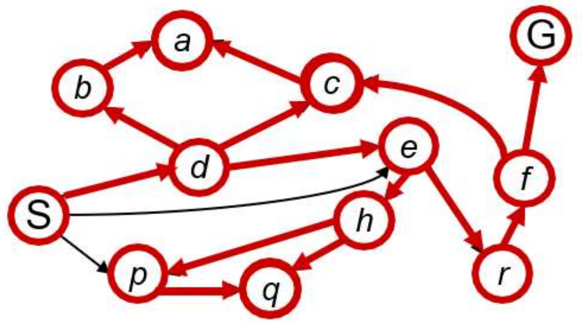
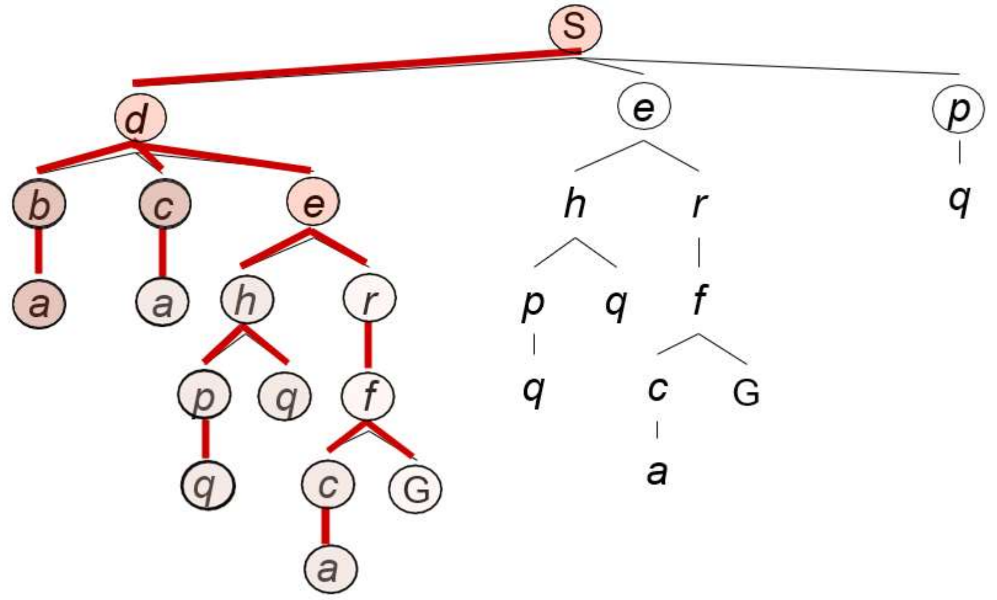


**Figure 3.16** Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and $M$ is the only goal node.

## DFS (3)

- The basic steps of DFS on a graph:
  1) Start with any unvisited node, visit it, and consider it as the current node.
  2) Search for an unvisited neighbour of the current node, visit it, and update it as the current node.
  3) If all the neighbours of the current node are visited, then backtrack to its predecessor (or parent) and update that predecessor as the new current node.
  4) Repeat steps (2) and (3) until all nodes in a graph are visited.
  5) If there are still unvisited nodes present in a graph, repeat steps (1) to (4).





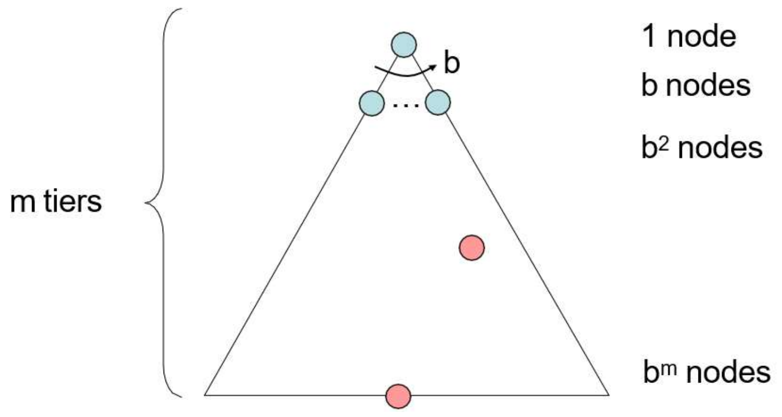
## DFS (4)

- **DFS Pseudocode:**

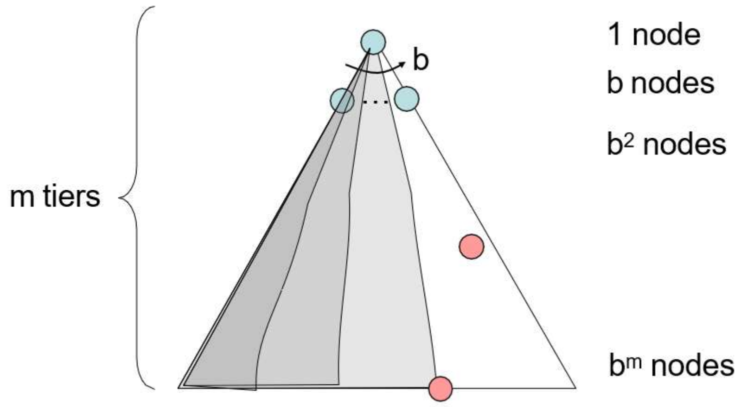
```
procedure DFS(G,v):
    for $v \in V$:
        $explored$[v] $\leftarrow$ false
    end for
    for all v $\in$ V do
        if not $explored$[v]:
            DFS-visit(v)
end procedure

procedure DFS-visit(v):
    $explored[v] \leftarrow true$
    for $u \in adj(v)$:
        if not $explored$[u]:
            DFS-visit(u)
        end if
    end for
end procedure
```

# DFS (5)

- **DFS Properties:**
- Cartoon of search tree has:
  - b is the branching factor
  - m is the maximum depth of any node, and this can be much larger than d (Shallowest solution depth)
  - solutions at various depths
  - Number of nodes in entire tree = $b^0 + b^1 + b^2 + .... b^m = O(b^m)$



- Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. If depth m is finite, it takes time $O(b^m)$.
- Space Complexity: DFS algorithm needs to store only single path from the root node, hence, space complexity of DFS is equivalent to the size of the frontier (fringe), which is only O(bm). (only has siblings on path to root).
- Completeness: DFS is complete within finite state space as it will expand every node within a limited search tree.
- Optimal: DFS does not provide always an optimal solution as it finds the "leftmost" solution, regardless of depth or cost.

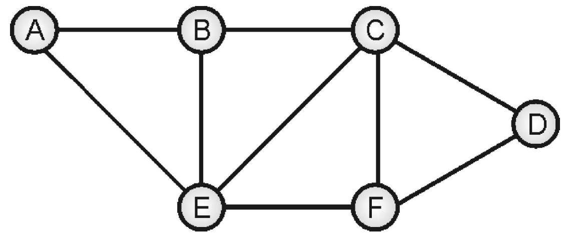


## DFS (6)

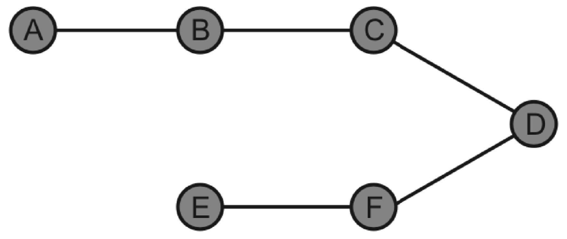
- A variant of depth-first search called backtracking search uses still less memory.
- In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next.
- In this way, only O(m) memory is needed rather than O(bm).
- Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by modifying the current state description directly rather than copying it first. This reduces the memory requirements to just one state description and O(m) actions.
- For this to work, we must be able to undo each modification when we go back to generate the next successor.
- For problems with large state descriptions, such as robotic assembly, these techniques are critical to success.

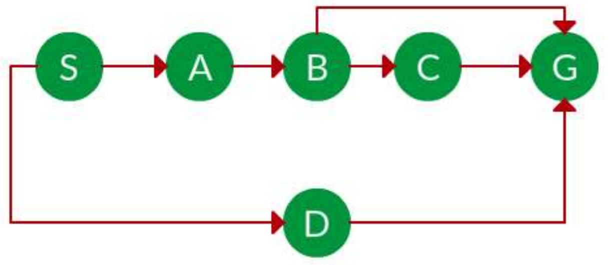
## DFS (7)

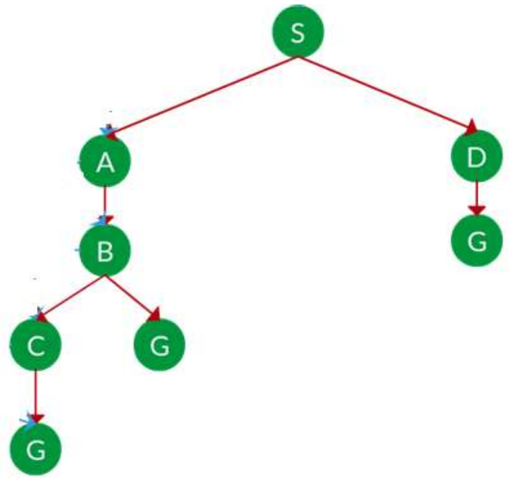
**Examples:**



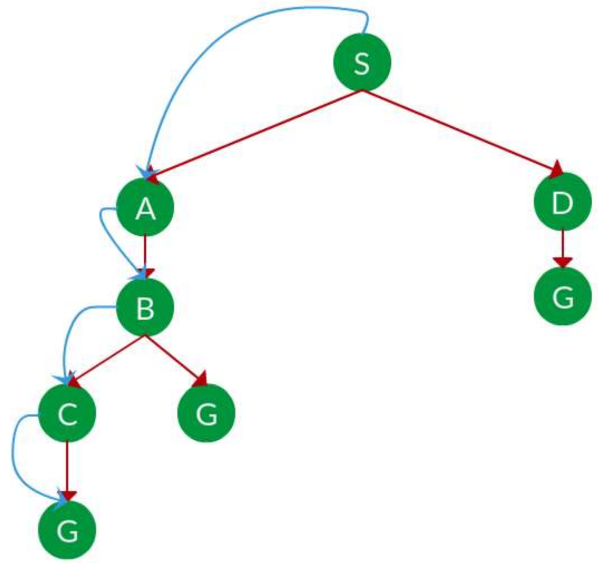
**The LIFO order of visited nodes is A, B, C, D, F, E.**







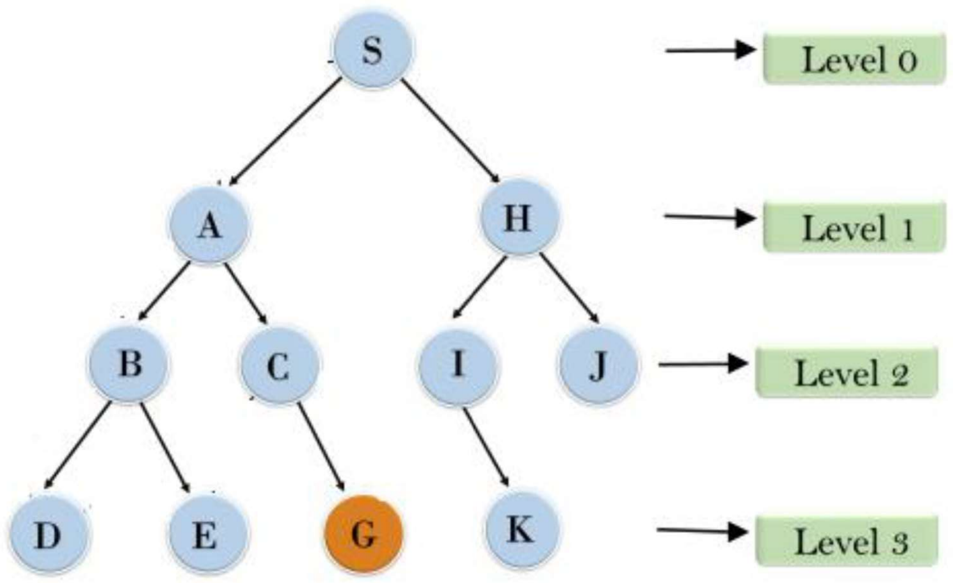
DFS traverses the tree “deepest node first”, it would always pick the deeper branch until it reaches the solution

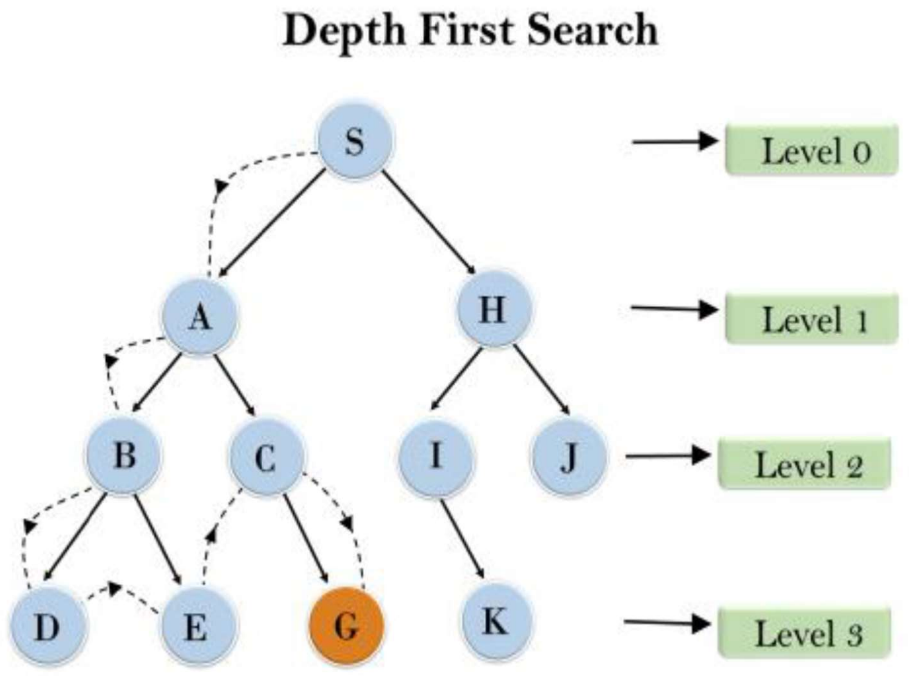


**The path to Goal state G is: S, A, B, C, G.**

## DFS (8)

**Example:**





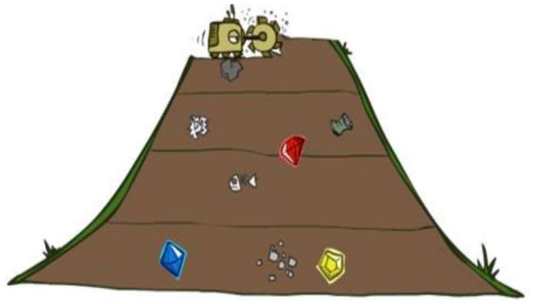
**The path to Goal state G is: S, A, B, D, E, C, G.**

## Breadth First Search (BFS)

**Source:**

1. Stuart Russell & Peter Norvig, "Artificial Intelligence : A Modern Approach", Pearson Education, 2nd Edition.
2. Elaine Rich and Kevin Knight, "Artificial Intelligence" Tata McGraw Hill.

## Breadth First Search (BFS) (1)



- Breadth-first search (BFS) expands all the nodes at a given depth in the search tree before any nodes at the next level are expanded.
- In BFS the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- In BFS the shallowest unexpanded node in the search tree is chosen for expansion.
- BFS uses FIFO queue for frontiers. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
- Advantages:
  - BFS will provide a solution if any solution exists.
  - If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.
- Disadvantages:
  - It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
  - BFS needs lots of time if the solution is far away from the root node.

## BFS (2)

```
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)
```

**Figure 3.11** Breadth-first search on a graph.

- **BFS on a binary tree:**

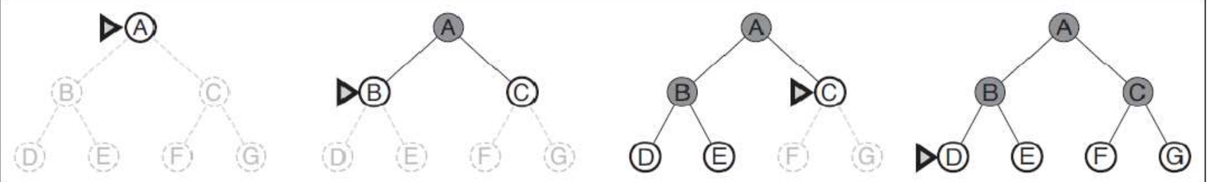
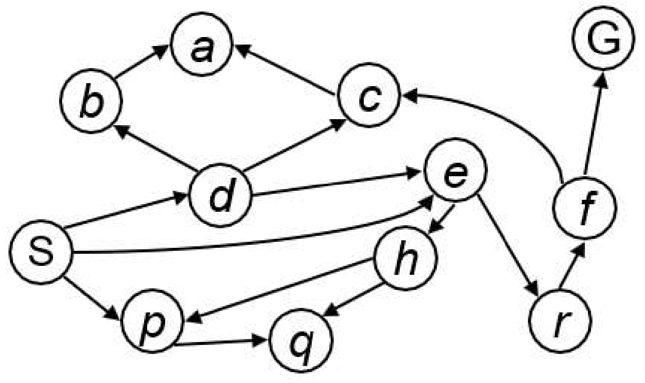
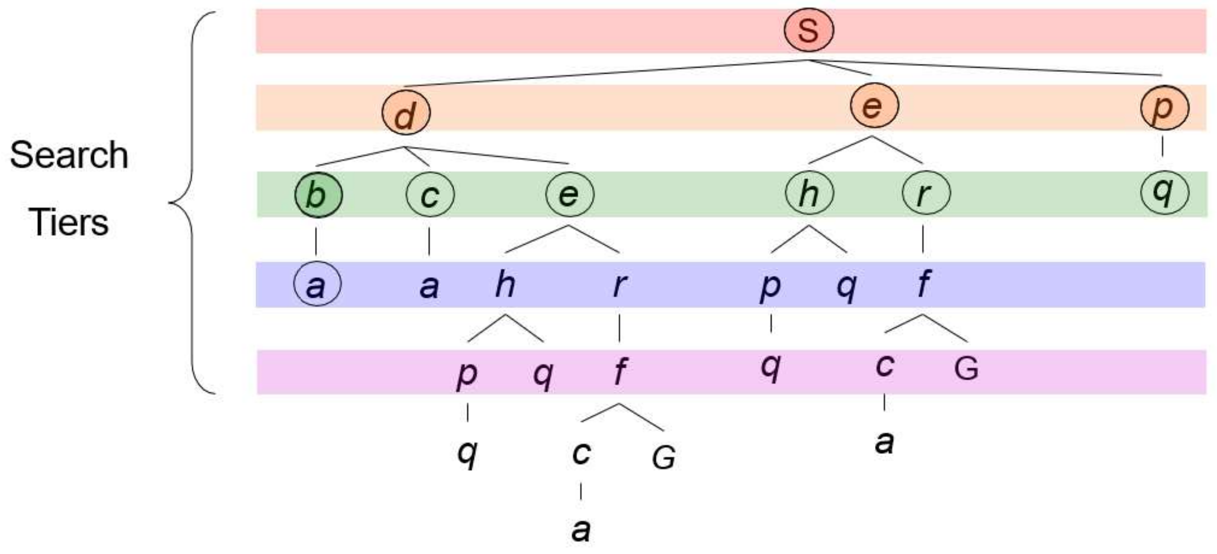


**Figure 3.12** Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

# BFS (3)

- The basic steps of BFS on a graph:
  1) Start with any unvisited node, visit it, and consider it as the root of a BFS tree Update its level as the current level.
  2) Search for all unvisited neighbours of the node in the current level, visit them one by one, and update the level of these newly visited neighbours as the new current level.
  3) Repeat step (2) until all nodes in a graph are visited.
  4) If there are still unvisited nodes present in a graph, repeat steps (1) to (4).



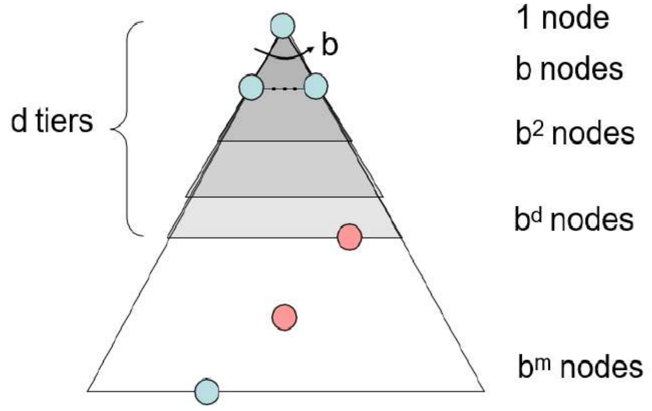


## BFS (4)

- **BFS Pseudocode:**

```
procedure BFS(G,s)

    for each vertex v $\in$ V[G] do
        $explored$[v] $\leftarrow$ false
        $d$[v] $\leftarrow$ $\infty$
    end for
    $explored$[s] $\leftarrow$ true
    $d$[s] $\leftarrow$ $0$
    $Q$:= a queue data structure, initialized with s
    while $Q \neq \phi$ do
        u $\leftarrow$ remove vertex from the front of $Q$
        for each v adjacent to u do
            if not $explored$[v] then
                $explored$[v] $\leftarrow$ true
                $d$[v] $\leftarrow$ $d$[u] + 1
                insert v to the end of $Q$
            end if
        end for
    end while

end procedure
```

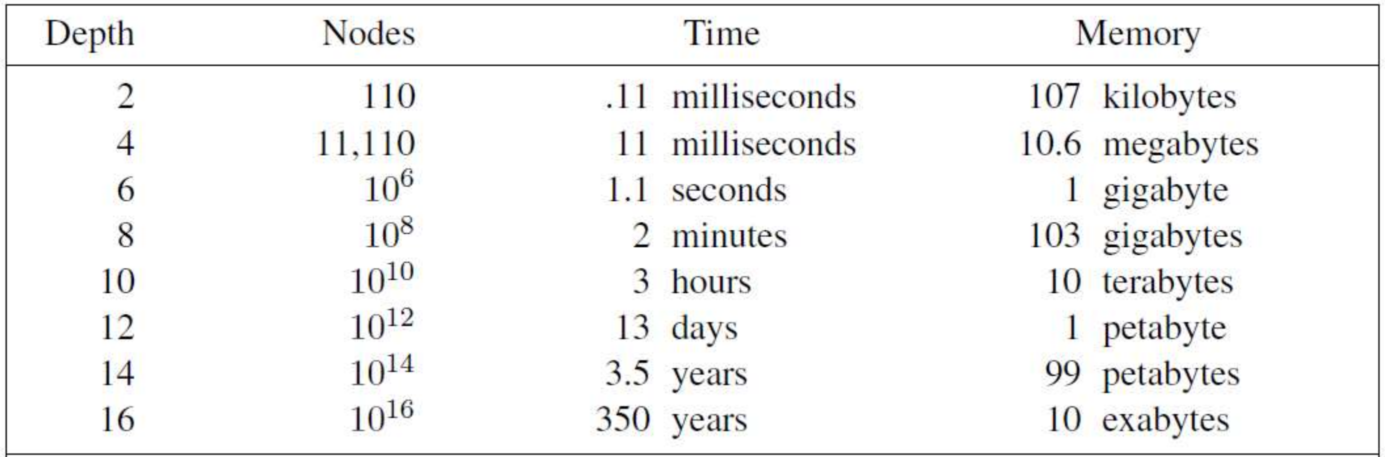
# BFS (5)



- **BFS Properties:**
- Cartoon of search tree has:
  - b is the branching factor
  - m is the maximum depth of any node, and m >> d (Shallowest solution depth)
  - Number of nodes in entire tree = $b^0 + b^1 + b^2 + \ldots. b^m = O(b^m)$
- Time Complexity:
  - BFS processes all nodes above shallowest solution at depth d. So, it takes time of $O(b^d)$.
  - If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $O(b^{d+1})$.
  - In the worst case, if the shallowest solution is the last node generated at depth m, then the total number of nodes generated is $O(b^m)$ and hence time complexity of BFS will be $O(b^m)$.
- Space Complexity:
  - BFS algorithm 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 $O(b^d)$, i.e., it is dominated by the size of the frontier.

## BFS (5)

- The memory requirements are a bigger problem for BFS than is the execution time. Time is still a major factor. In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.



| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 110 | .11 milliseconds | 107 kilobytes |
| 4 | 11,110 | 11 milliseconds | 10.6 megabytes |
| 6 | $10^6$ | 1.1 seconds | 1 gigabyte |
| 8 | $10^8$ | 2 minutes | 103 gigabytes |
| 10 | $10^{10}$ | 3 hours | 10 terabytes |
| 12 | $10^{12}$ | 13 days | 1 petabyte |
| 14 | $10^{14}$ | 3.5 years | 99 petabytes |
| 16 | $10^{16}$ | 350 years | 10 exabytes |

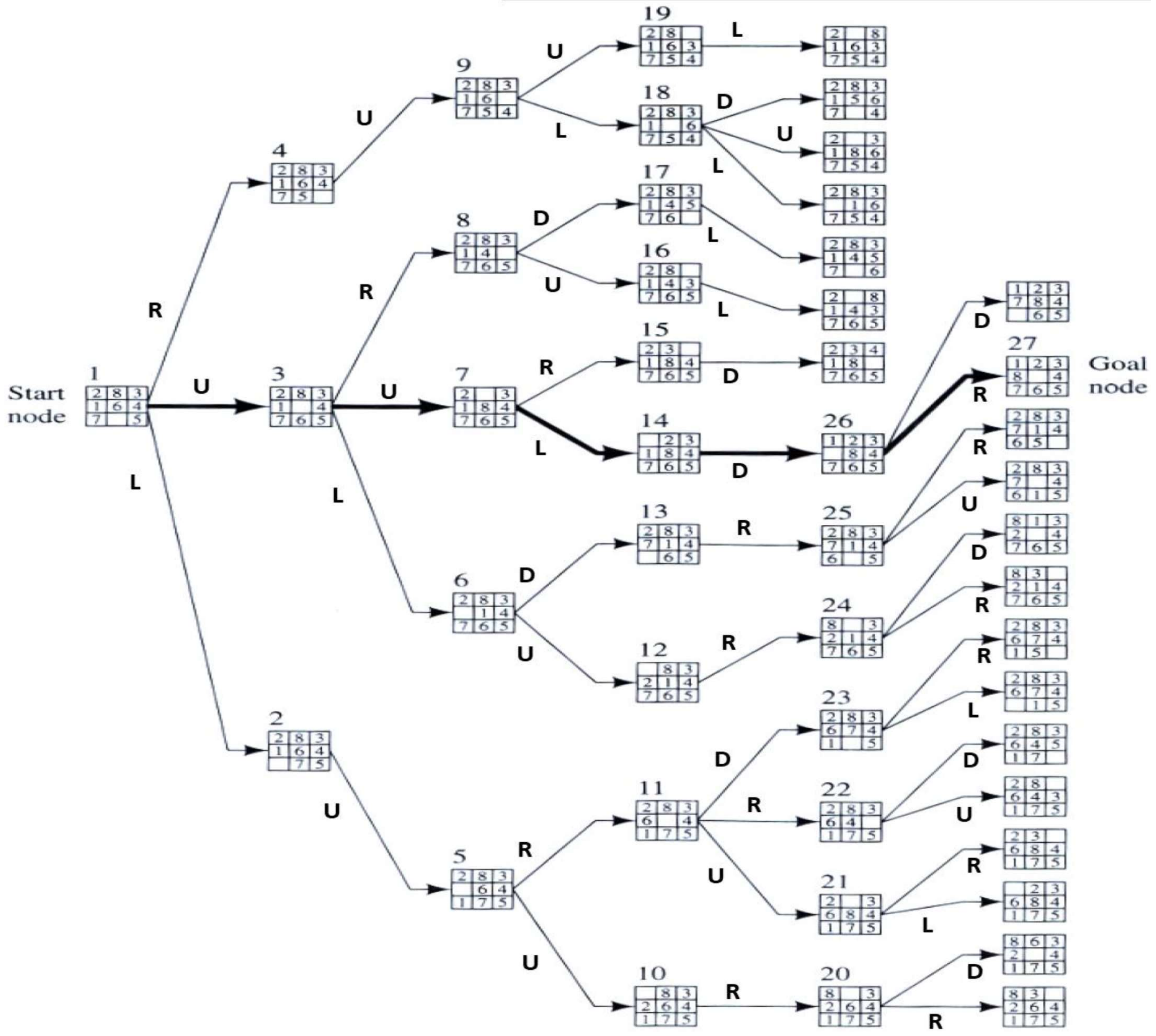
**Figure 3.13** Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

- Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

- Optimal: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

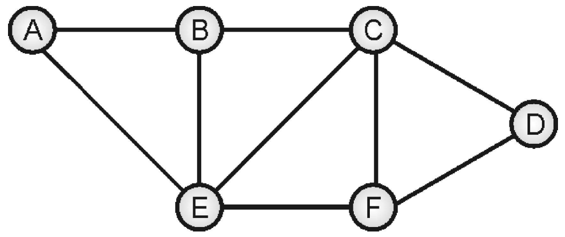
## BFS(6)

- **Example of BFS for 8-Puzzle Problem:**

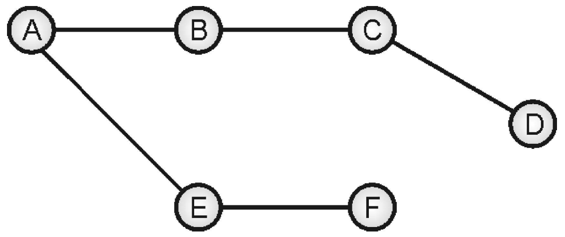


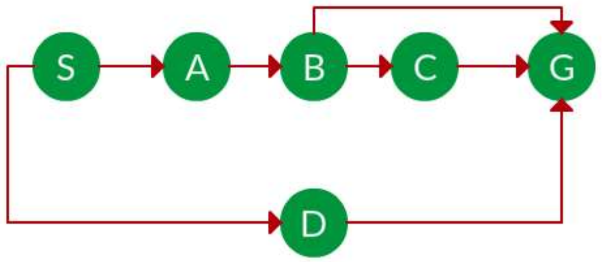
# BFS (7)

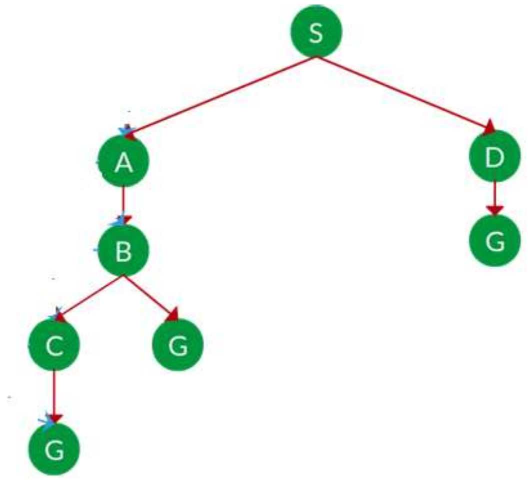
**Examples:**



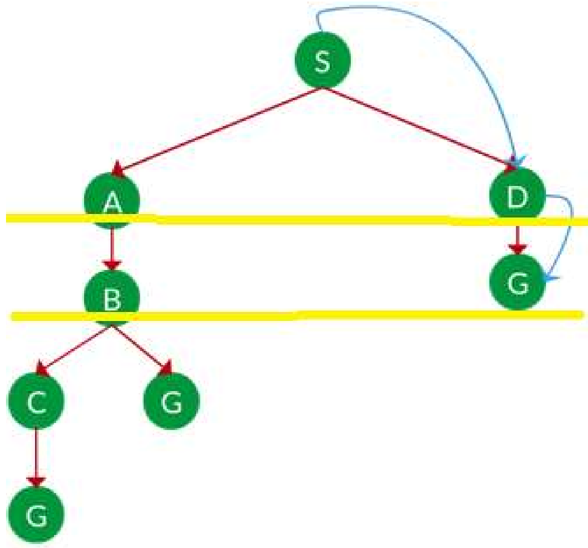
**The FIFO order of visited nodes is A, B, E, C, F, D.**







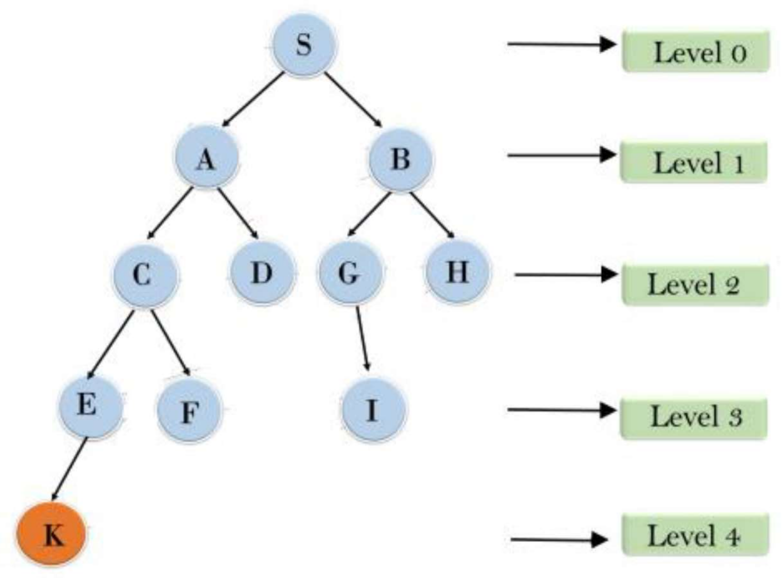
BFS traverses the tree “shallowest node first”, it would always pick the shallower branch until it reaches the solution.

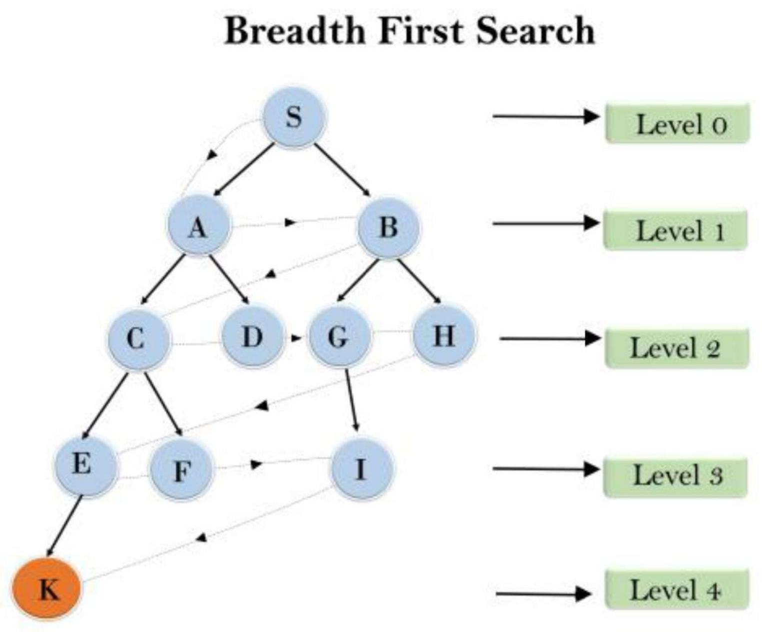


**The path to Goal state G is: S, D, G.**

## BFS(8)

- **Example:**





**The path to Goal state K is: S, A, B, C, D, G, H, E, F, I, K.**

## Depth Limited Search

**Source:**

1. Stuart Russell & Peter Norvig, "Artificial Intelligence : A Modern Approach", Pearson Education, 2nd Edition.

## Depth Limited Search (DLS) (1)

- The failure of DFS in infinite state spaces can be alleviated by supplying DFS with a predetermined depth limit $\ell$. That is, nodes at depth $\ell$ are treated as if they have no successors. This approach is called Depth Limited Search (DLS).
- The depth limit solves the infinite-path problem. DLS can be viewed as a special case of DFS.
- The diameter of the state space, gives us a better depth limit, which leads to a more efficient DLS. For most problems, however, we will not know a good depth limit until we have solved the problem.
- DLS can be terminated with two conditions of failure:
  - Standard failure value: It indicates that problem does not have any solution.
  - Cutoff failure value: It defines no solution for the problem within a given depth limit.
- DLS uses LIFO stack for frontiers as it is in DFS. A LIFO sequence means that the most recently generated i.e. the deepest unexpanded node is chosen for expansion.
- Advantages:
  - DLS is more efficient than DFS, using less time and memory.
  - If a solution exists, DFS guarantees that it will be found in a finite amount of time.
- Disadvantages:
  - DLS also has a disadvantage of incompleteness.
  - It may not be optimal if the problem has more than one solution.

## DLS (2)

```
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
```

**Figure 3.17** A recursive implementation of depth-limited tree search.

## DLS (3)

- **DLS Properties:**
- Cartoon of search tree has:
  - b is the branching factor
  - $\ell$ is the depth limit,
  - d is the shallowest solution depth
  - Number of nodes in entire tree = $b^0 + b^1 + b^2 + \ldots. b^\ell = O(b^\ell)$
  - Depth-first search can be viewed as a special case of depth-limited search with $\ell=\infty$

- Time Complexity: Time complexity of DLS will be equivalent to the node traversed by the algorithm. If depth limit is $\ell$, it takes time $O(b^\ell)$.

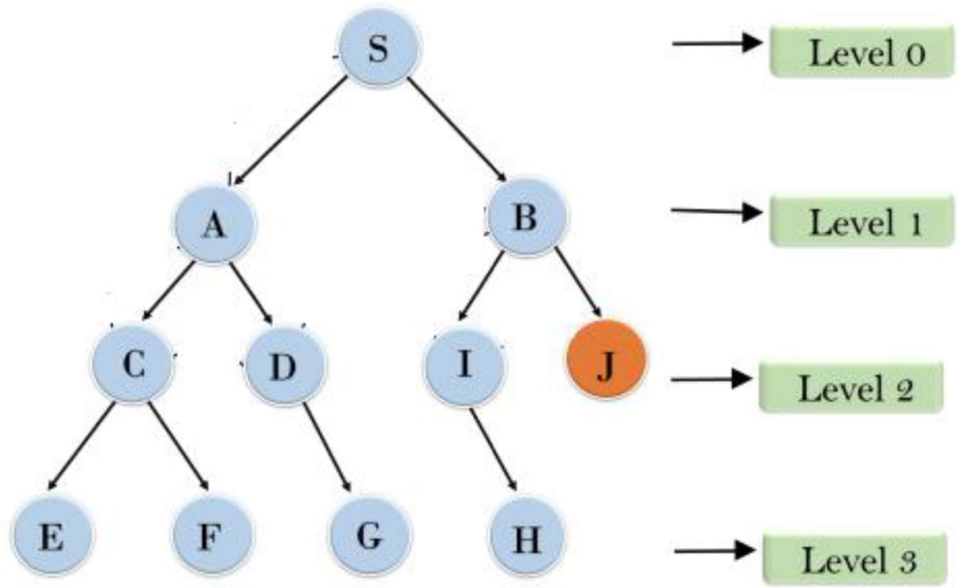
- Space Complexity: DLS algorithm needs to store only single path from the root node, hence, space complexity of DLS is equivalent to the size of the frontier (fringe), which is only $O(b\ell)$. (only has siblings on path to root).

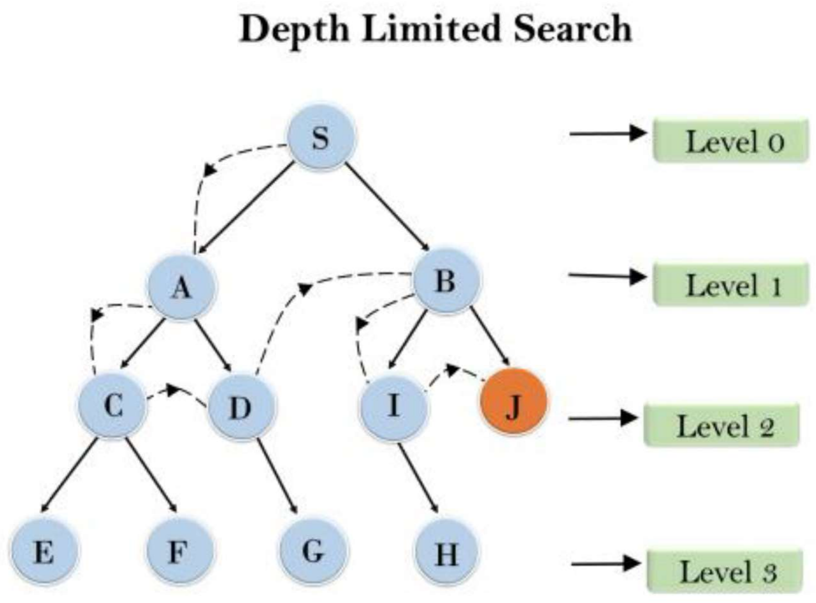
- Completeness: Unfortunately, it also introduces an additional source of incompleteness if we choose $\ell < d$, that is, the shallowest goal is beyond the depth limit. (This is likely when d is unknown.)

- Optimal: DLS will also be nonoptimal if we choose $\ell > d$.

## DLS (4)

- **Example:**





**The path to Goal state J is: S, A, C, D, B, I, J.**