Unit-II
Uninformed Search Strategies

-Dr. Radhika V. Kulkarni

Associate Professor, Dept. of Computer Engineering,

Vishwakarma Institute of Technology, Pune.

Sources:

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
3. Deepak Khemani, “A First Course in Artificial Intelligence”, McGraw Hill
4. Saroj Kaushik, “Artificial Intelligence”, Cengage Publication.

DISCLAIMER

This presentation is created as a reference material for the students of TY-CS, VIT (AY 2023-24 Sem-1).

It is restricted only for the internal use and any circulation is strictly prohibited.

Syllabus

Unit-II Uninformed Search Strategies

Uninformed Search Methods: Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Depth First Search, Bidirectional Search, Comparison of Uninformed search Strategies.

Search Algorithms

Sources:

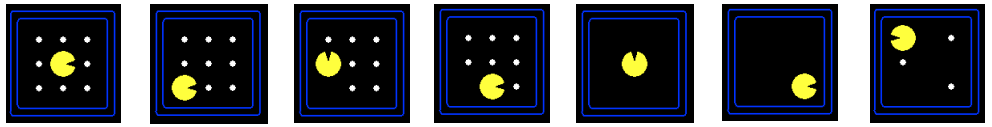
1. Stuart Russell & Peter Norvig, "Artificial Intelligence : A Modern Approach", Pearson Education, 2nd Edition.
2. Elaine Rich and Kevin Knight, "Artificial Intelligence" Tata McGraw Hill.

Problem-Solving

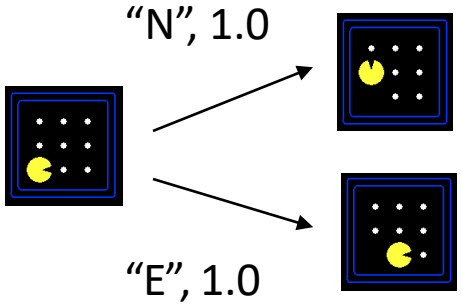
- A problem can be defined formally by five components:
  - The initial state that the agent starts in.
  - A description of the possible actions available to the agent.
  - A description of what each action does; the formal name for this is the transition model.
  - The goal test, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.
  - A path cost function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.
- A solution to a problem is an action sequence that leads from the initial state to a goal state.
  - Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.
- Execution: Once a solution is found, the actions it recommends can be carried out. This is called the execution phase. Thus, we have a simple "formulate, search, execute" design for the agent.
- Search: It is a process of looking for a sequence of actions that reaches the goal.
- The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions, and the nodes correspond to states in the state space of the problem.
- The set of all leaf nodes available for expansion at any given point is called the frontier.
- Search algorithms vary primarily according to how they choose which state to expand next—the so-called search strategy.

Search Problems

- A search problem consists of:
  - A state space



  - A successor function (with actions, costs)

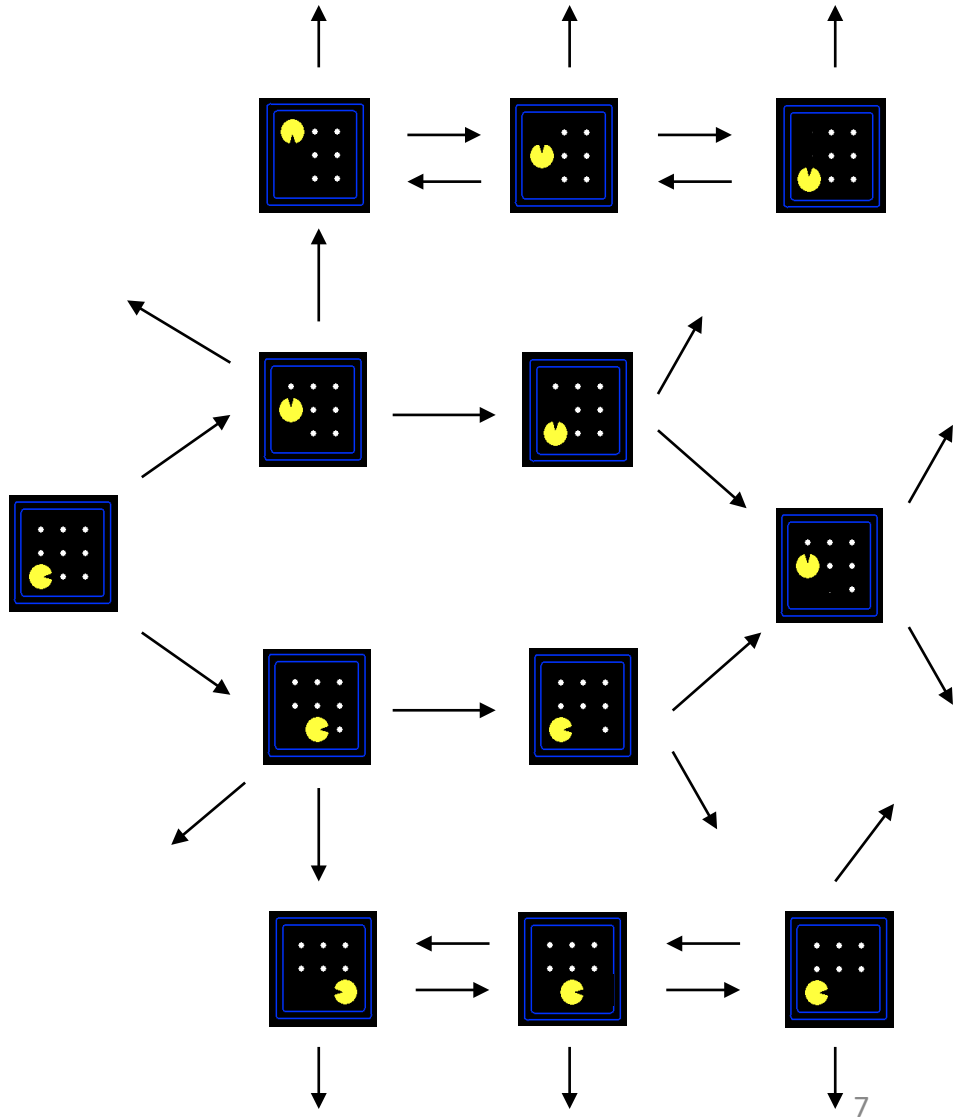


  - A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

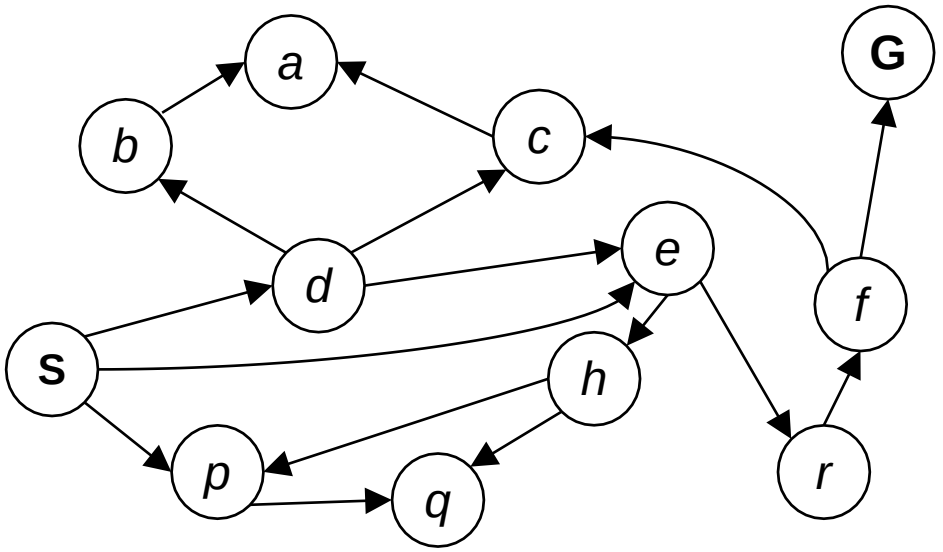
State Space Trees

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



Tiny state space graph for a tiny search problem

Search Algorithms

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

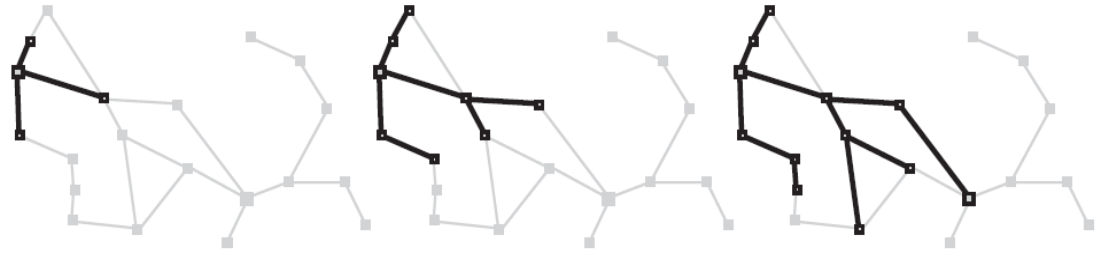


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

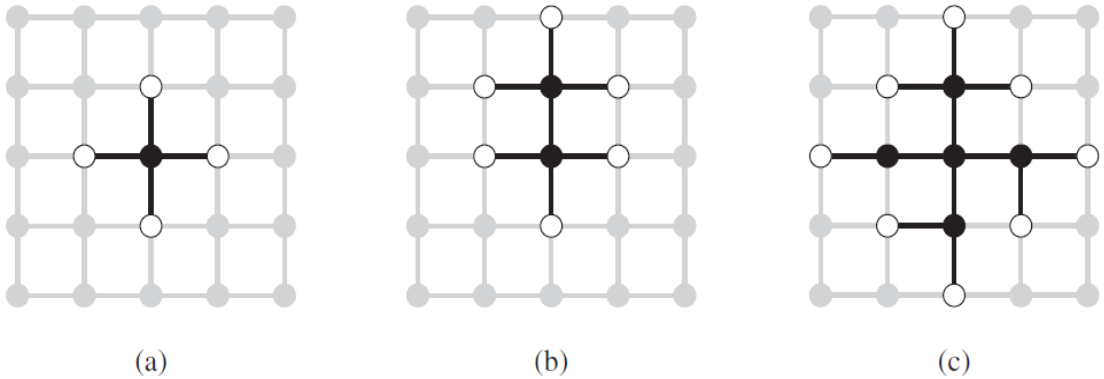


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

Infrastructure for Search Algorithms (1)

- Search algorithms require a data structure to keep track of the search tree that is being constructed.
- 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, traditionally denoted by *g(n)*, of the path from the initial state to the node, as indicated by the parent pointers.

- Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Infrastructure for Search Algorithms (2)

- The node data structure is depicted in Figure 3.10. The PARENT pointers string the nodes together into a tree structure. They allow the solution path to be extracted when a goal node is found; the SOLUTION function returns the sequence of actions obtained by following parent pointers back to the root.
- A node is a bookkeeping data structure used to represent the search tree.
- A state corresponds to a configuration of the world.
- Thus, nodes are on particular paths, as defined by PARENT pointers, whereas states are not.
- Furthermore, two different nodes can contain the same world state if that state is generated via two different search paths.

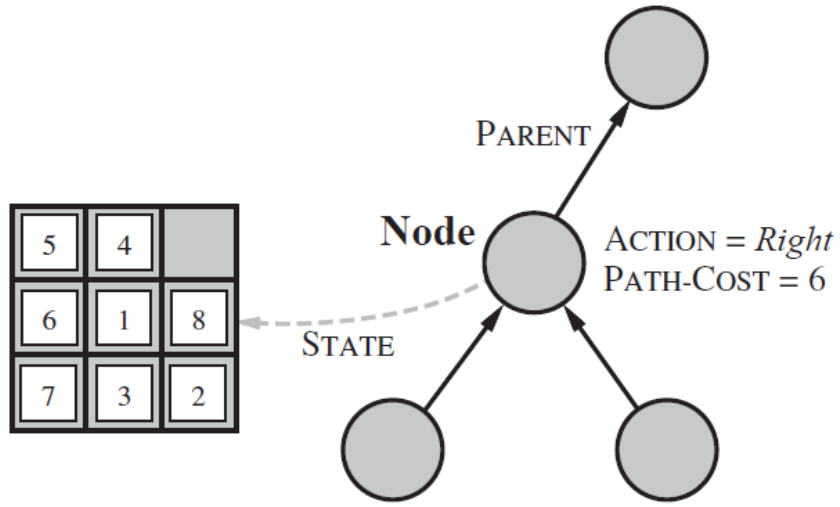


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Infrastructure for Search Algorithms (3)

- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
- The appropriate data structure for this is a queue. The operations on a queue are as follows:
  - 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. Three common variants are:
  1) the first-in, first-out or FIFO queue, which pops the oldest element of the queue;
  2) the last-in, first-out or LIFO queue (also known as a stack), which pops the newest element of the queue;
  3) the priority queue, which pops the element of the queue with the highest priority according to some ordering function.

- The explored set can be implemented with a hash table to allow efficient checking for repeated states.

Measuring Problem-Solving Performance (1)

- An algorithm's performance is evaluated in four ways:
  - Completeness: Is the algorithm guaranteed to find a solution when there is one?
  - Optimality: Does the strategy find the optimal solution?
  - Time complexity: How long does it take to find a solution?
  - Space complexity: How much memory is needed to perform the search?
- Time and space complexity are always considered with respect to some measure of the problem difficulty.

- In theoretical computer science, the typical measure is the size of the state space graph, |V | + |E|, where V is the set of vertices (nodes) of the graph and E is the set of edges (links).

- This is appropriate when the graph is an explicit data structure that is input to the search program. (E.g., The map of a city in the route-finding problem.)

Measuring Problem-Solving Performance (2)

- In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities:
  - 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);
  - 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.

- To assess the effectiveness of a search algorithm, we can consider just the search cost— which typically depends on the time complexity but can also include a term for memory usage—or we can use the total cost, which combines the search cost and the path cost of the solution found.

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

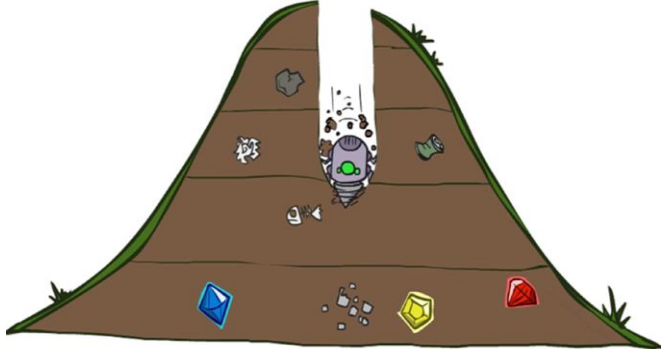
- These algorithms apply the strategies that know whether one non-goal state is “more promising” than another. They are also called heuristic search strategies.
- They can do quite well given some guidance on where to look for solutions.
- E.g., Hill Climbing, Best First Search, A* and AO* Algorithm, 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:**

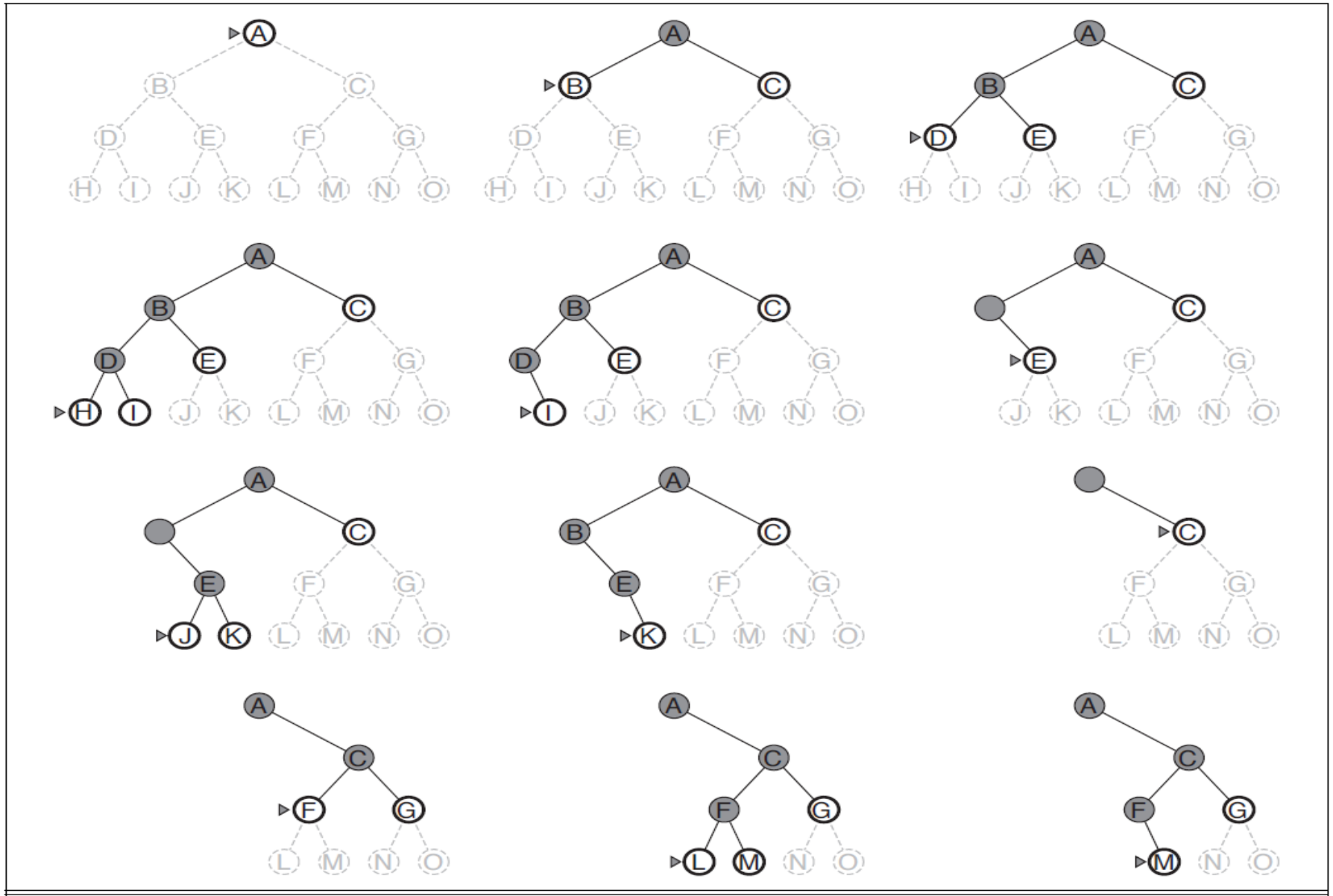
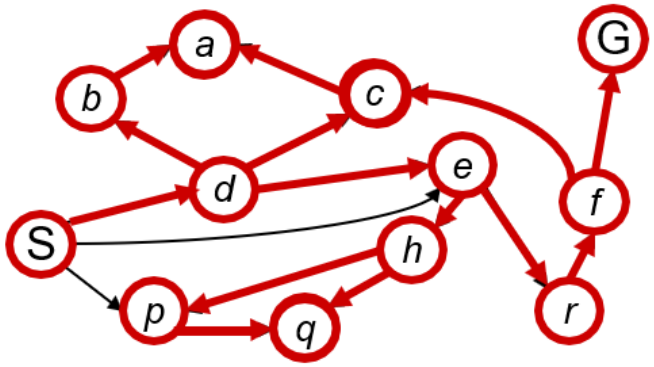
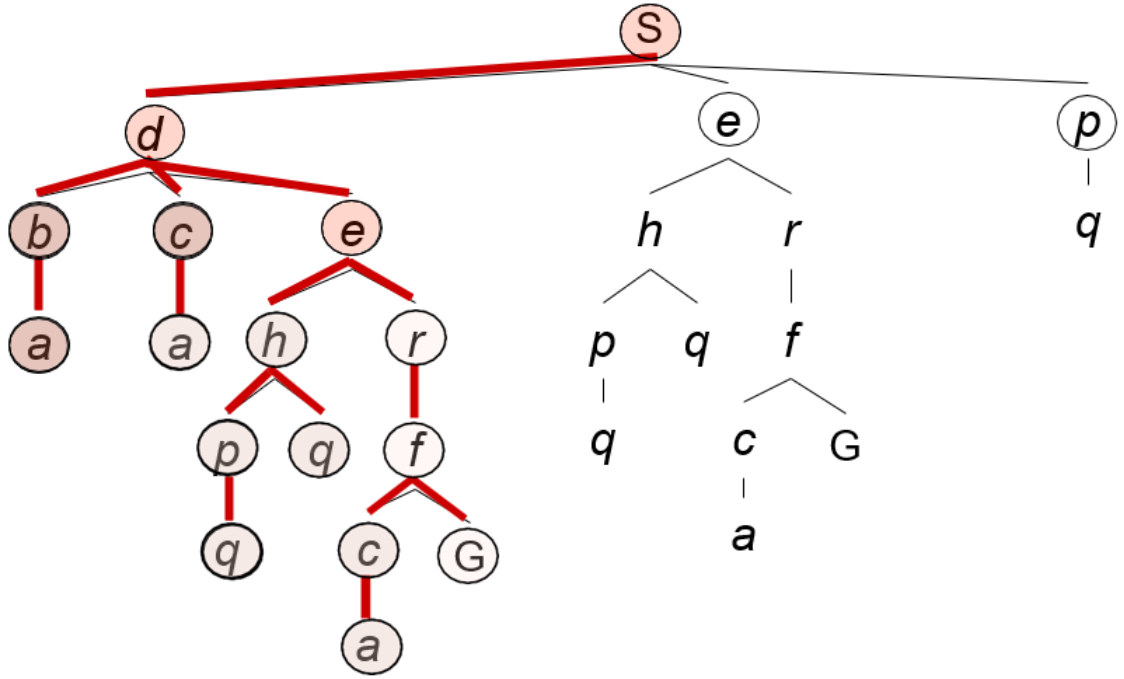


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

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).





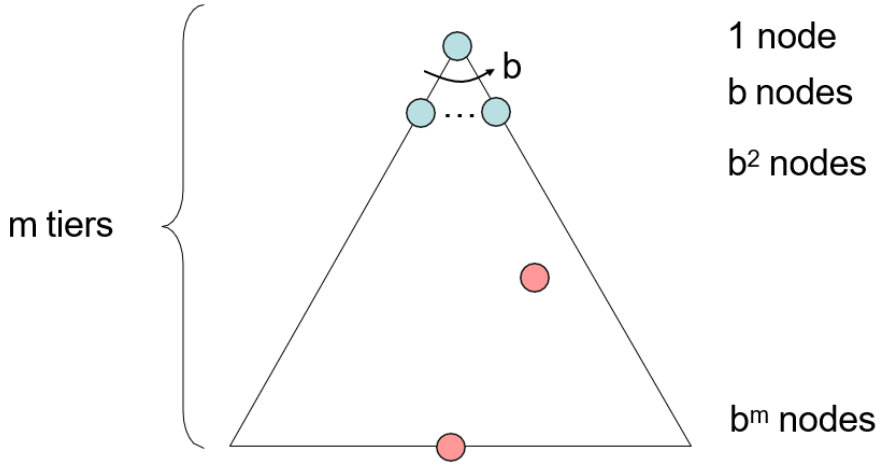
DFS (4)

- **DFS Pseudocode:**

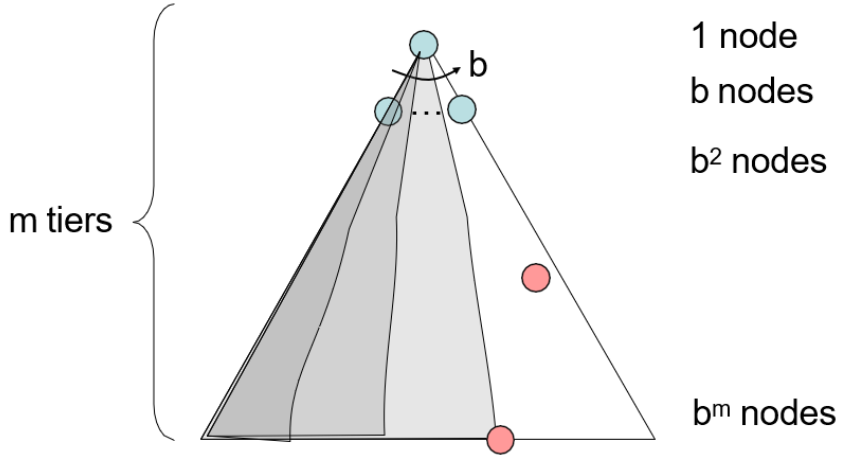
```
procedure DFS(G,v):
    for v ∈ V:
        explored[v] ← false
    end for
    for all v ∈ V do
        if not explored[v]:
            DFS-visit(v)
end procedure

procedure DFS-visit(v):
    explored[v] ← true
    for u ∈ 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 = $b^0 + b^1 + b^2 + \ldots. b^m = O(b^m)$



- 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.

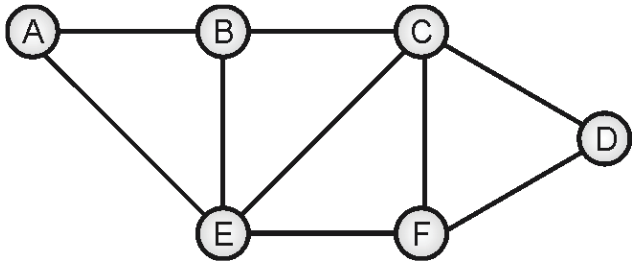


DFS (6)

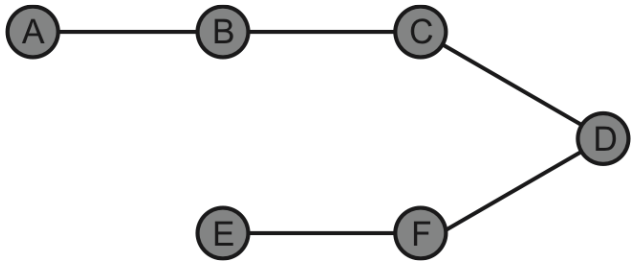
- A variant of depth-first search called backtracking search uses still less memory.
- In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next.
- In this way, only O(m) memory is needed rather than 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.

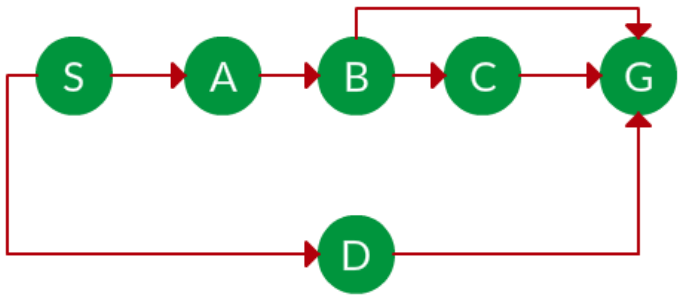
DFS (7)

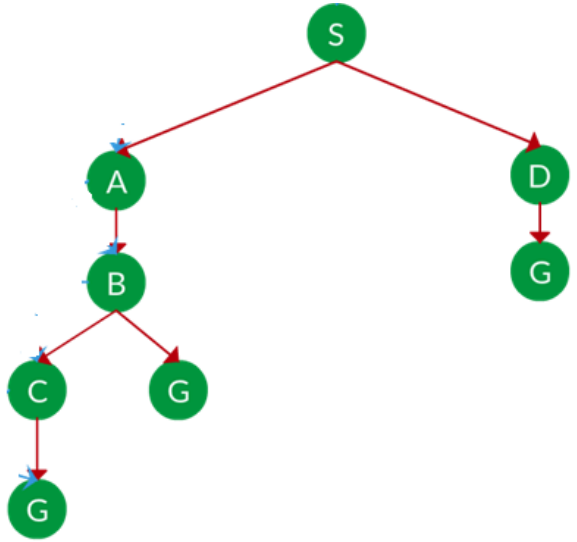
Examples:

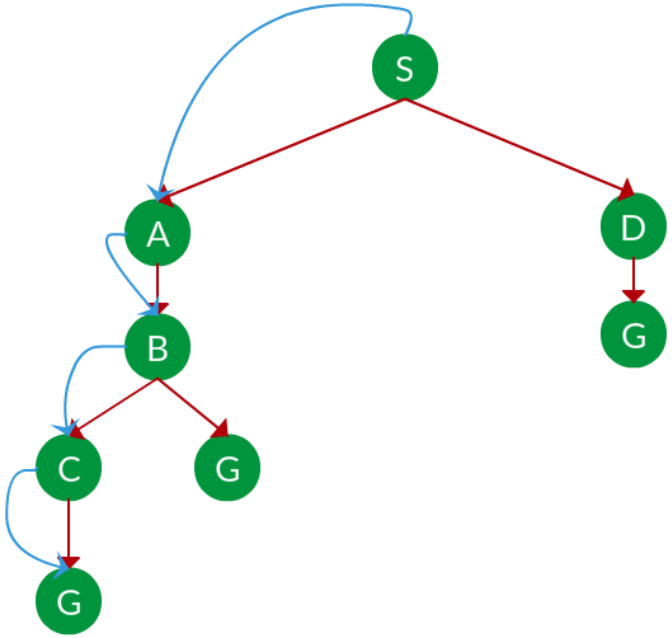


The LIFO order of visited nodes is A, B, C, D, F, E.







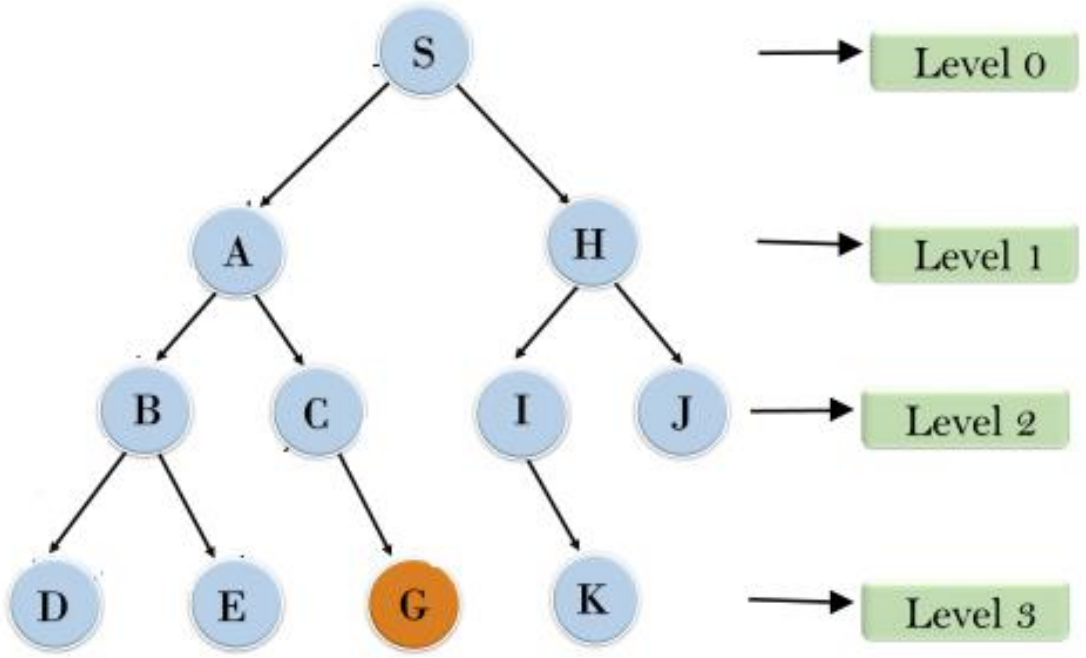


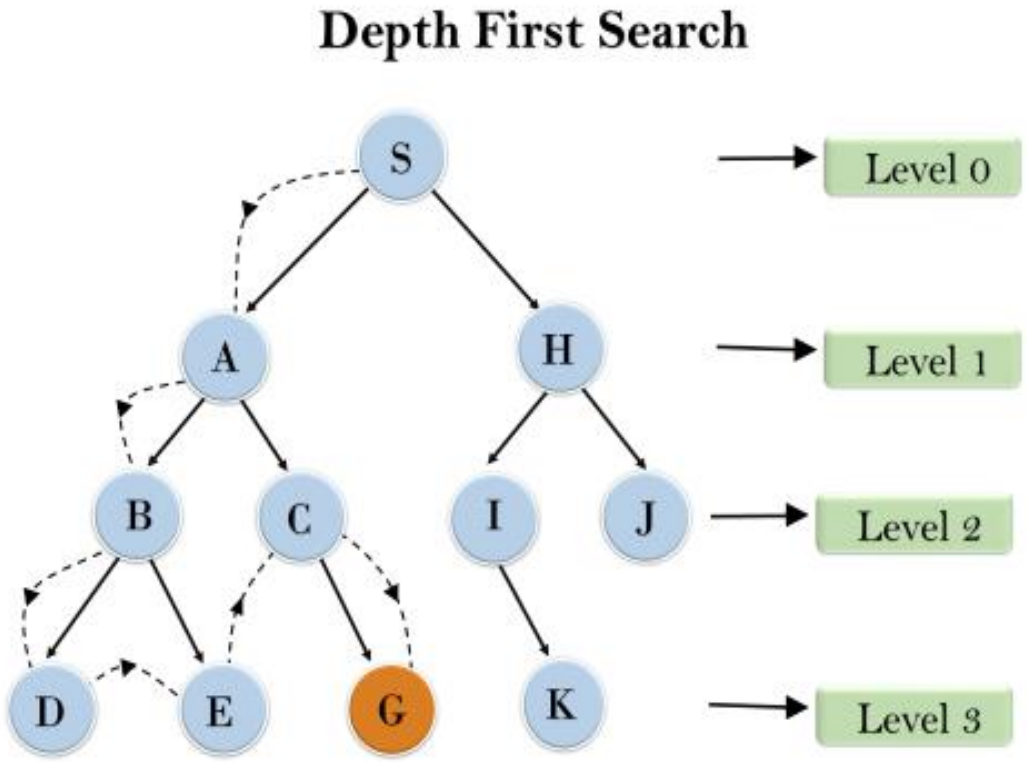
DFS traverses the tree “deepest node first”, it would always pick the deeper branch until it reaches the solution.

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:**

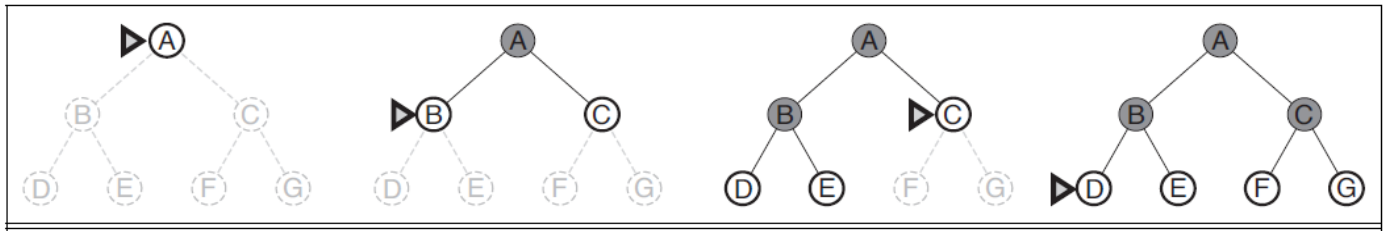
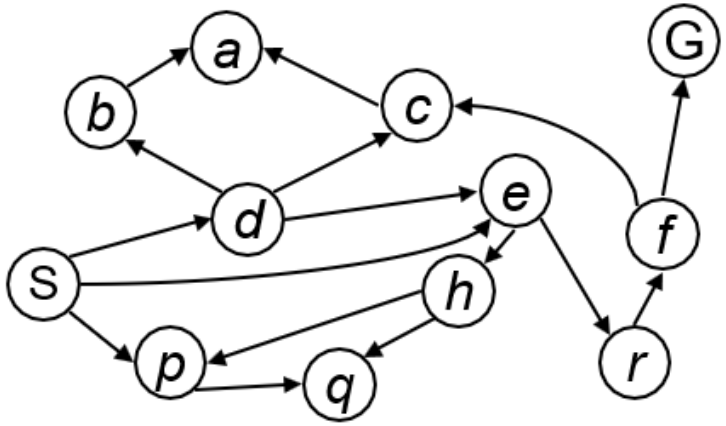
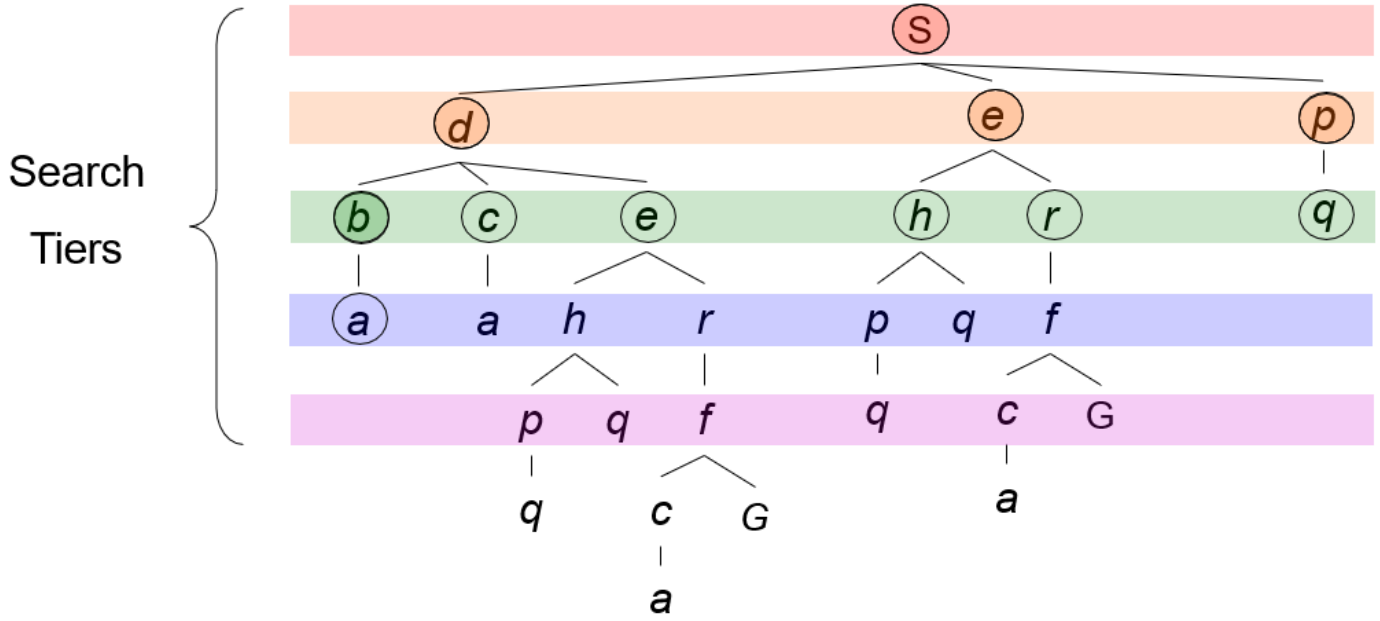


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

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).





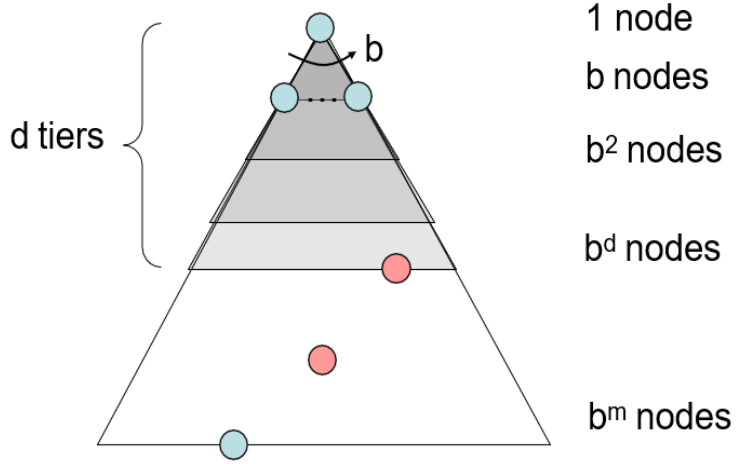
BFS (4)

- **BFS Pseudocode:**

```
procedure BFS(G,s)

    for each vertex v ∈ V[G] do
        explored[v] ← false
        d[v] ← ∞
    end for
    explored[s] ← true
    d[s] ← 0
    Q := a queue data structure, initialized with s
    while Q ≠ φ do
        u ← remove vertex from the front of Q
        for each v adjacent to u do
            if not explored[v] then
                explored[v] ← true
                d[v] ← 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

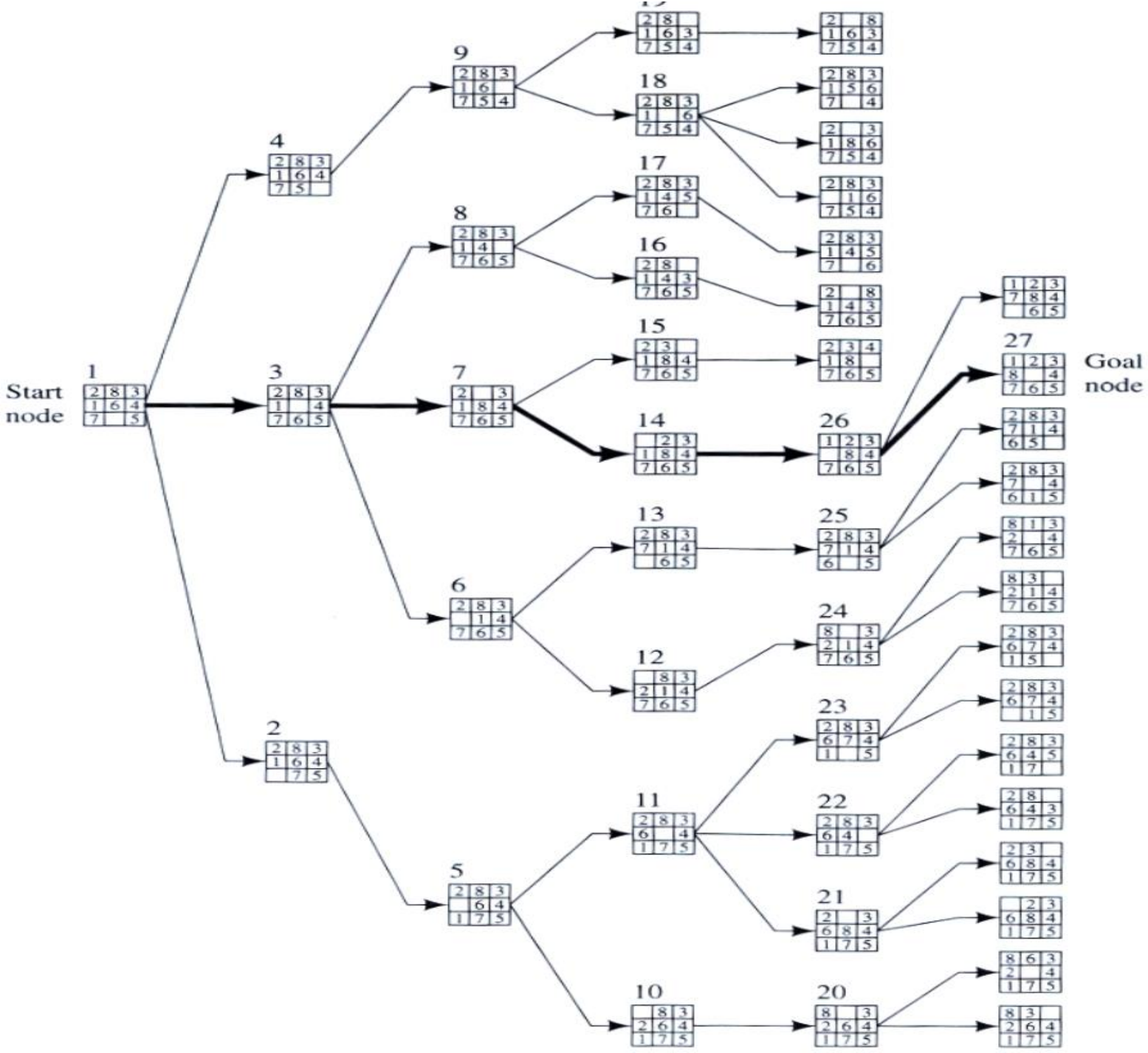
Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

- 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.

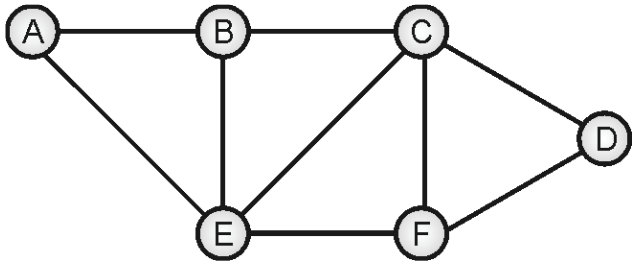
BFS(6)

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

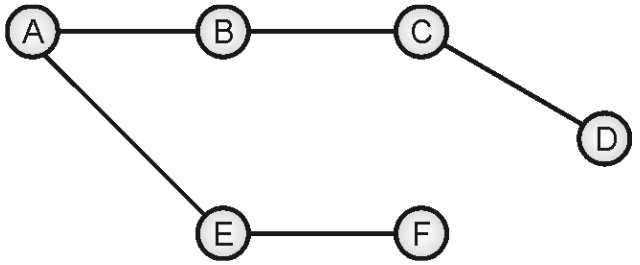


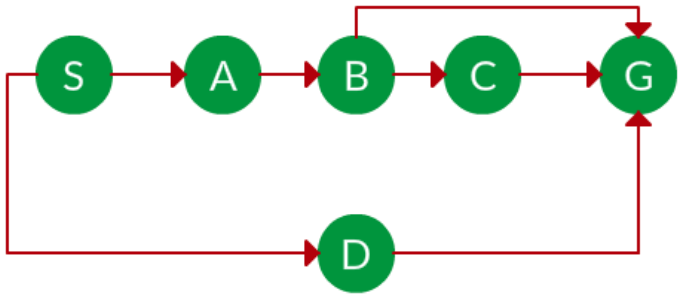
BFS (7)

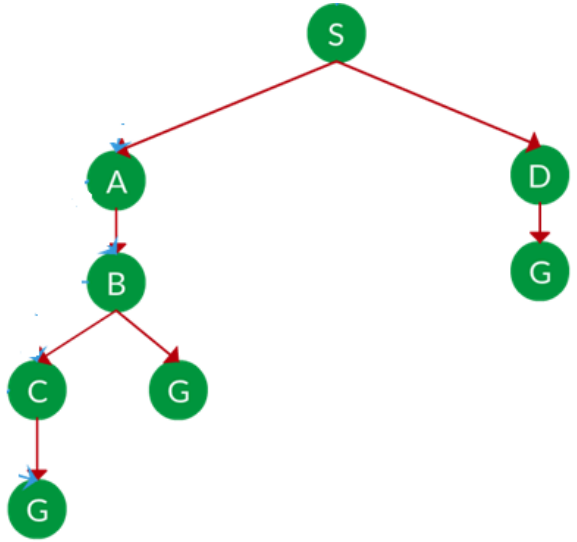
Examples:



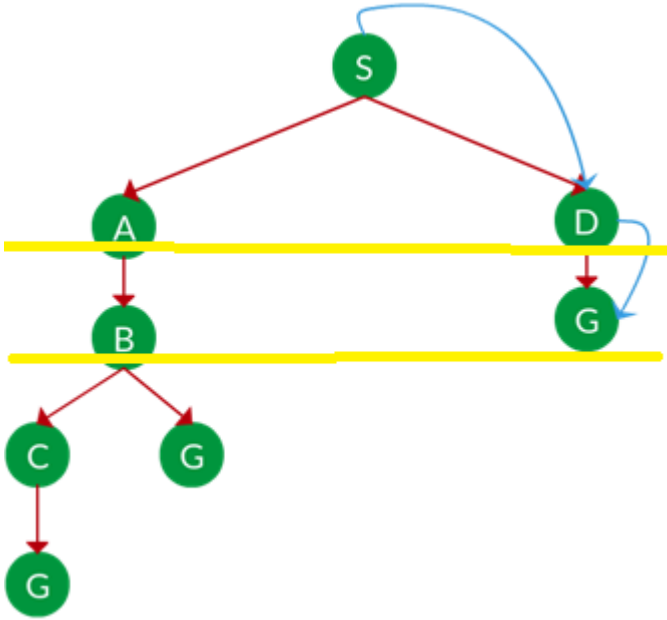
The FIFO order of visited nodes is A, B, E, C, F, D.







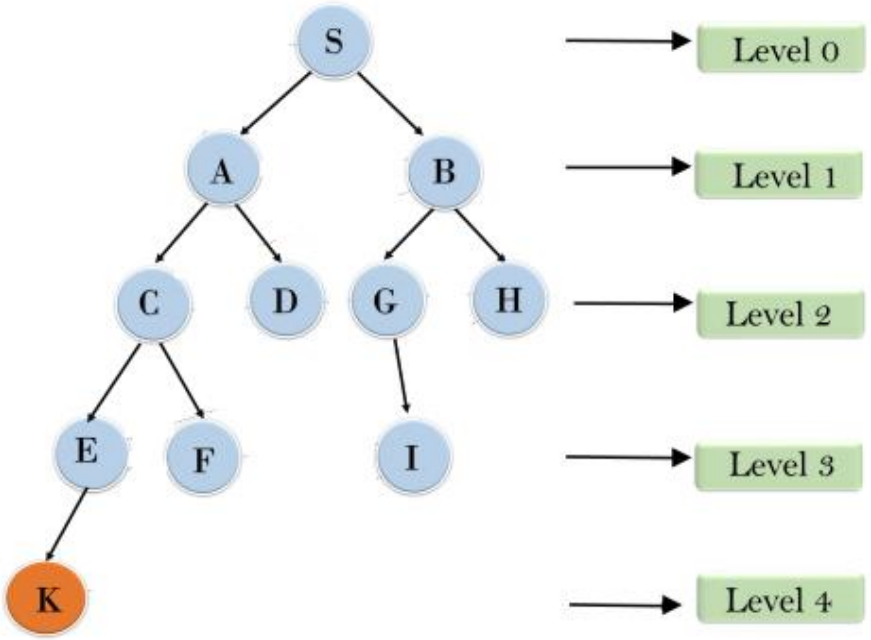
BFS traverses the tree “shallowest node first”, it would always pick the shallower branch until it reaches the solution.

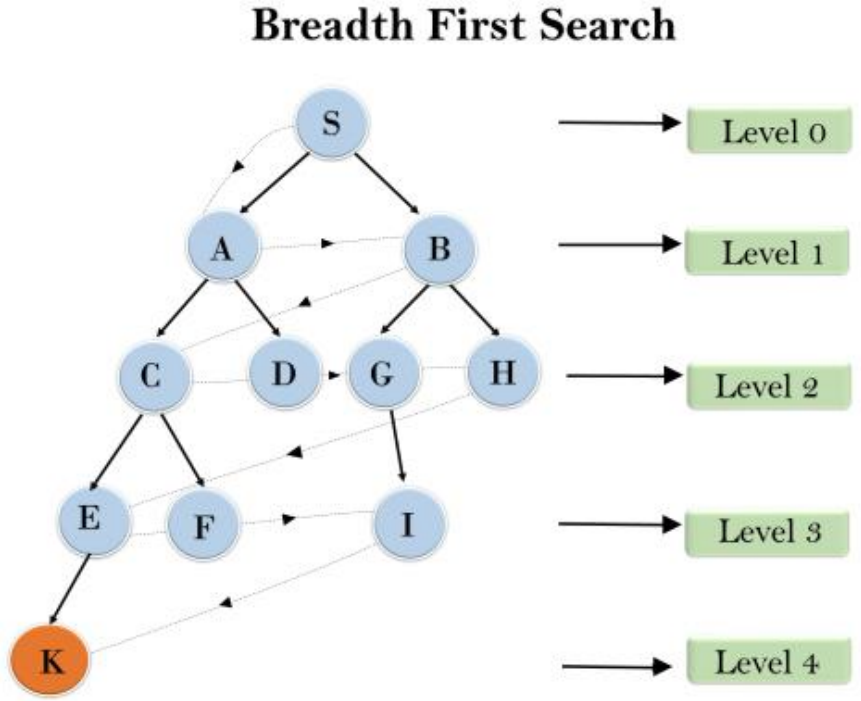


The path to Goal state G is: S, D, G.

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 ℓ. That is, nodes at depth ℓ 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
  - ℓ is the depth limit,
  - d is the shallowest solution depth
  - Number of nodes in entire tree = $b^0 + b^1 + b^2 + ... b^{\ell} = O(b^{\ell})$
  - 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 ℓ, it takes time $O(b^{\ell})$.

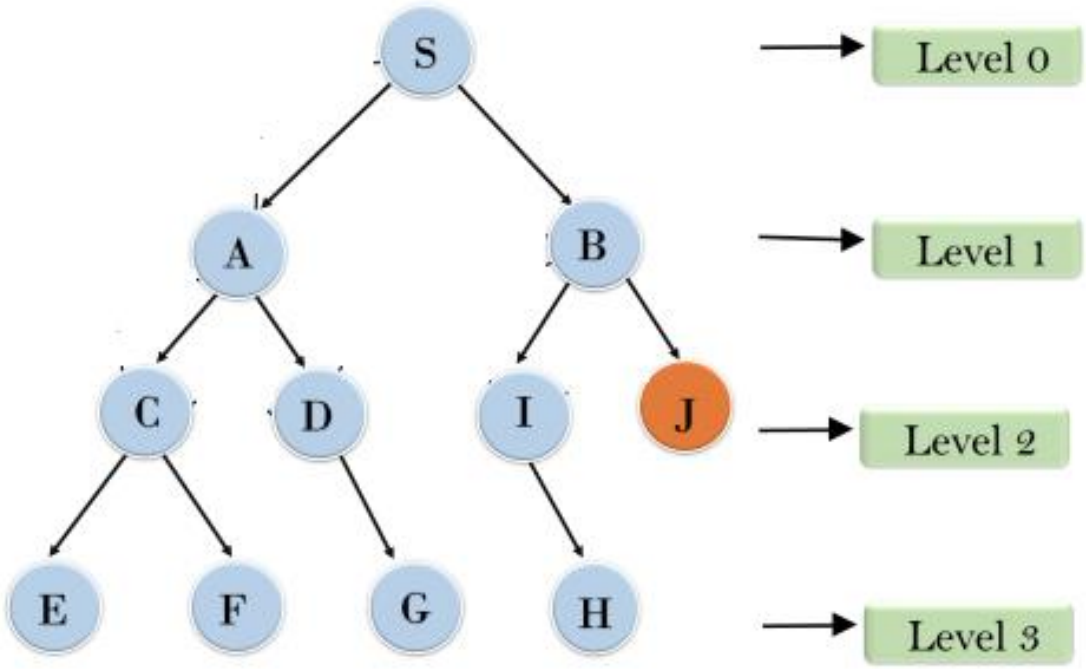
- Space Complexity: DLS algorithm needs to store only single path from the root node, hence, space complexity of DLS is equivalent to the size of the frontier (fringe), which is only $O(b\ell)$. (only has siblings on path to root).

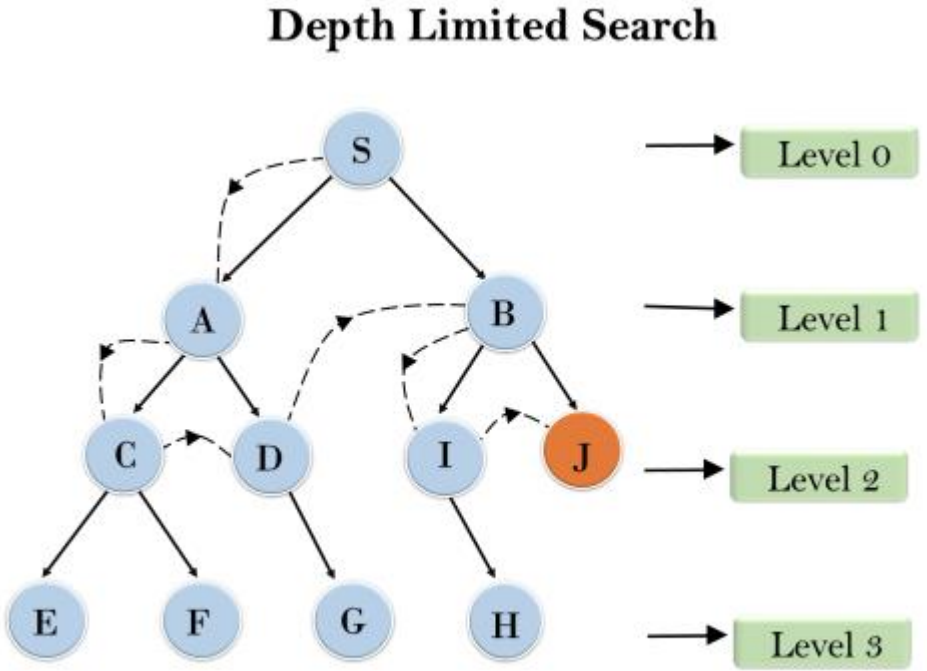
- Completeness: Unfortunately, it also introduces an additional source of incompleteness if we choose $\ell < d$, that is, the shallowest goal is beyond the depth limit. (This is likely when d is unknown.)

- 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.

Iterative Deepening Depth First Search

Source:

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

Iterative Deepening Depth-First Search (IDDFS) (1)

- Iterative Deepening Depth-First Search (IDDFS) or Iterative Deepening Search (IDS) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node.
- IDDFS 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.
- In general, IDDFS is the preferred uninformed search method when the search space is large, and the depth of the solution is not known.
- Advantages:
  - IDDFS is useful uninformed search when search space is large, and depth of goal node is unknown.
  - It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.
    - Like DFS, its memory requirements are modest: O(bd) to be precise.
    - Like BFS, 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.
- Disadvantages:
  - The main drawback of IDDFS is that it repeats all the work of the previous phase.

IDDFS (2)

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

IDDFS (3)

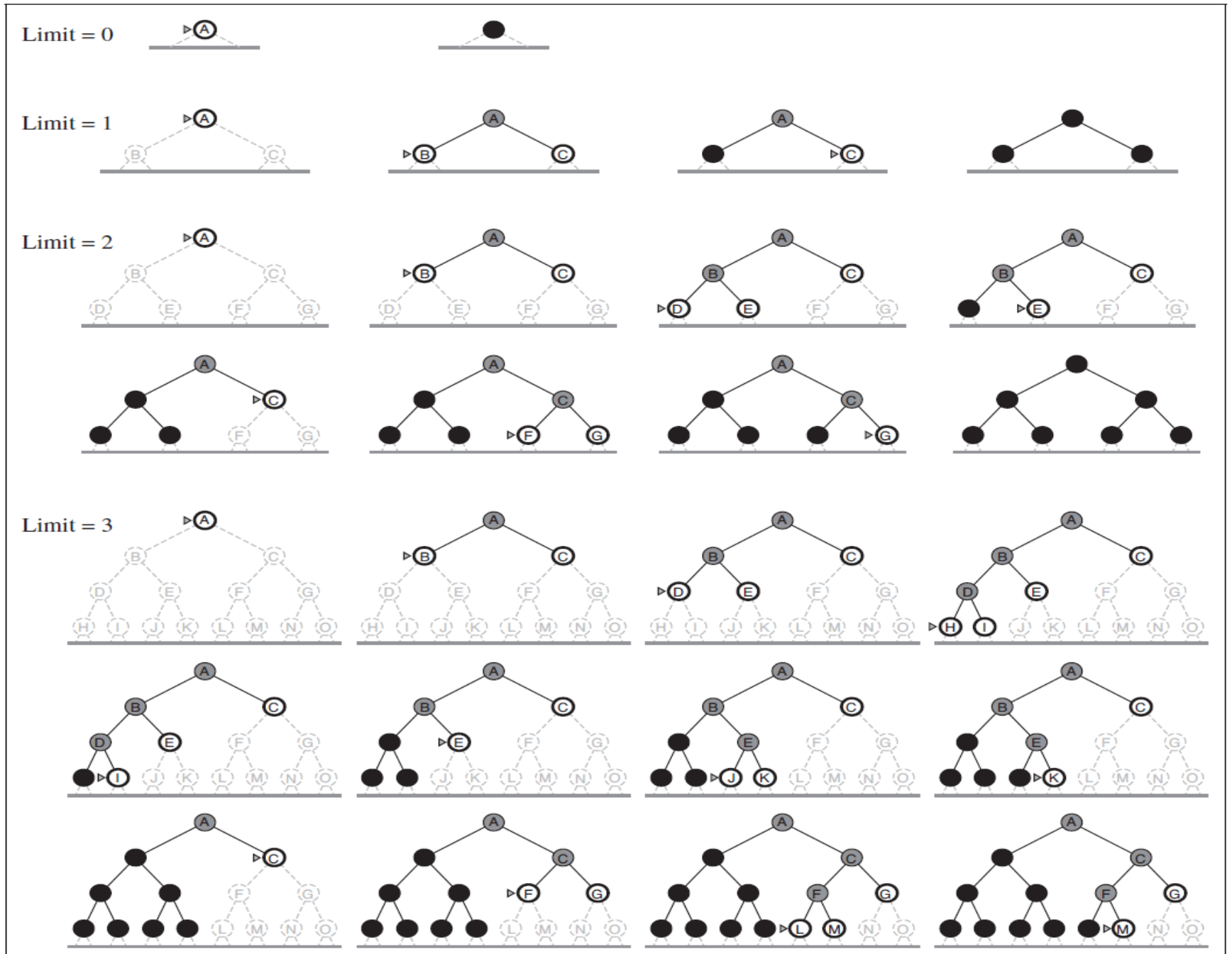


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

IDDFS (4)

- **IDDFS Properties:**
- Cartoon of search tree has:
  - b is the branching factor
  - d is the shallowest solution depth
  - Number of nodes in entire tree = $(d-1)b^0 + (d)b^1 + (d-1)b^2 + \ldots.+ (3)b^{d-2} + (2)b^{d-1} + (1)b^d = O(b^d)$

- Time Complexity: Time complexity of IDDFS will be equivalent to the node traversed by the algorithm. If depth of the shallowest solution is d, it takes time $O(b^d)$.

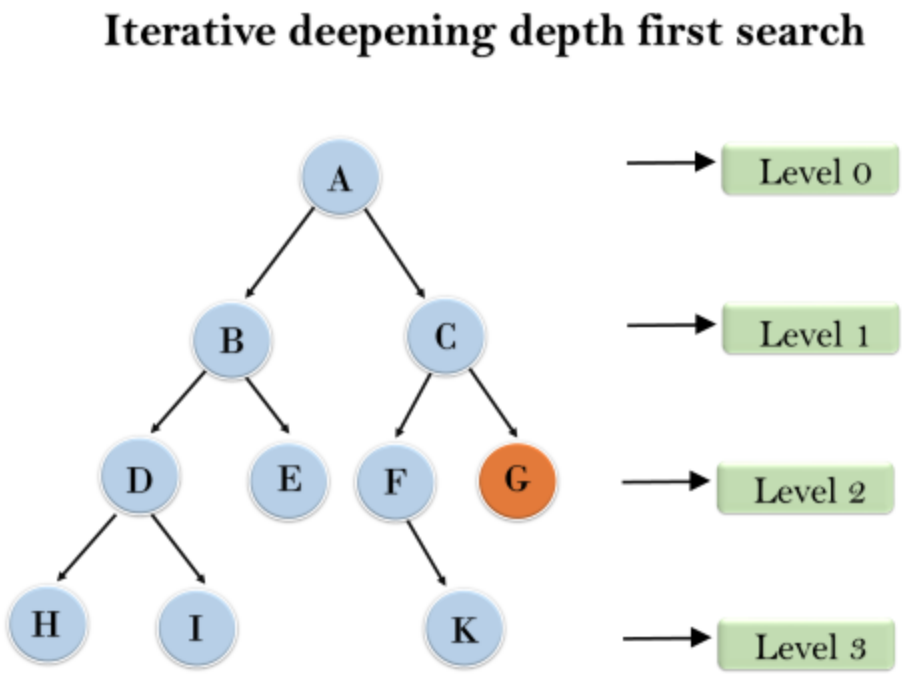
- Space Complexity: IDDFS algorithm needs to store only single path from the root node, hence, space complexity of IDDFS is equivalent to the size of the frontier (fringe), which is only O(bd). (only has siblings on path to root).

- Completeness: This algorithm is complete is if the branching factor b is finite.

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

IDDFS (5)

Example:



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

Bidirectional Search

Source:

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

Bidirectional Search (1)

- Bidirectional Search algorithm runs two simultaneous searches, one form initial state (S) called as forward-search and other from goal node (G) called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d, or in the figure 3.20, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

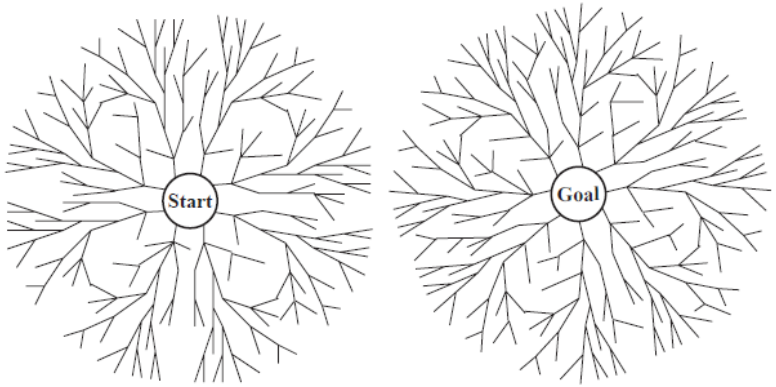


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Bidirectional Search (2)

- Advantages:
  - Bidirectional search is fast.
  - Bidirectional search requires less memory.
- Disadvantages:
  - Implementation of the bidirectional search tree is difficult.
  - In bidirectional search, one should know the goal state in advance.
  - Difficulties in backward search from a goal state G:
    - Need a way to specify the predecessors of G. It is very difficult. e.g., predecessors of checkmate in chess?
    - Which to take if there are multiple goal states?
    - Where to start if there is only a goal test, no explicit list?

Bidirectional Search (3)

- **Bidirectional Search Properties:**
- Time Complexity: Time complexity of bidirectional search using BFS in both directions is $O(b^{d/2})$. It will be equivalent to the node traversed by the algorithm.

- Space Complexity: Space complexity of bidirectional search is $O(b^{d/2})$.
  - We can reduce this by roughly half if one of the two searches is done by iterative deepening, but at least one of the frontiers must be kept in memory so that the intersection check can be done.
  - This space requirement is the most significant weakness of bidirectional search.

- Completeness: Bidirectional Search is complete if we use BFS in both searches..

- Optimal: Bidirectional search is Optimal.

Comparison of Uninformed search Strategies

Source:

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

Comparison of Uninformed Tree-Search Strategies

Criterion	Breadth-First	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Thank you!