ARTIFICIAL INTELLIGENCE & MACHINE LEARNING 23AD2001O

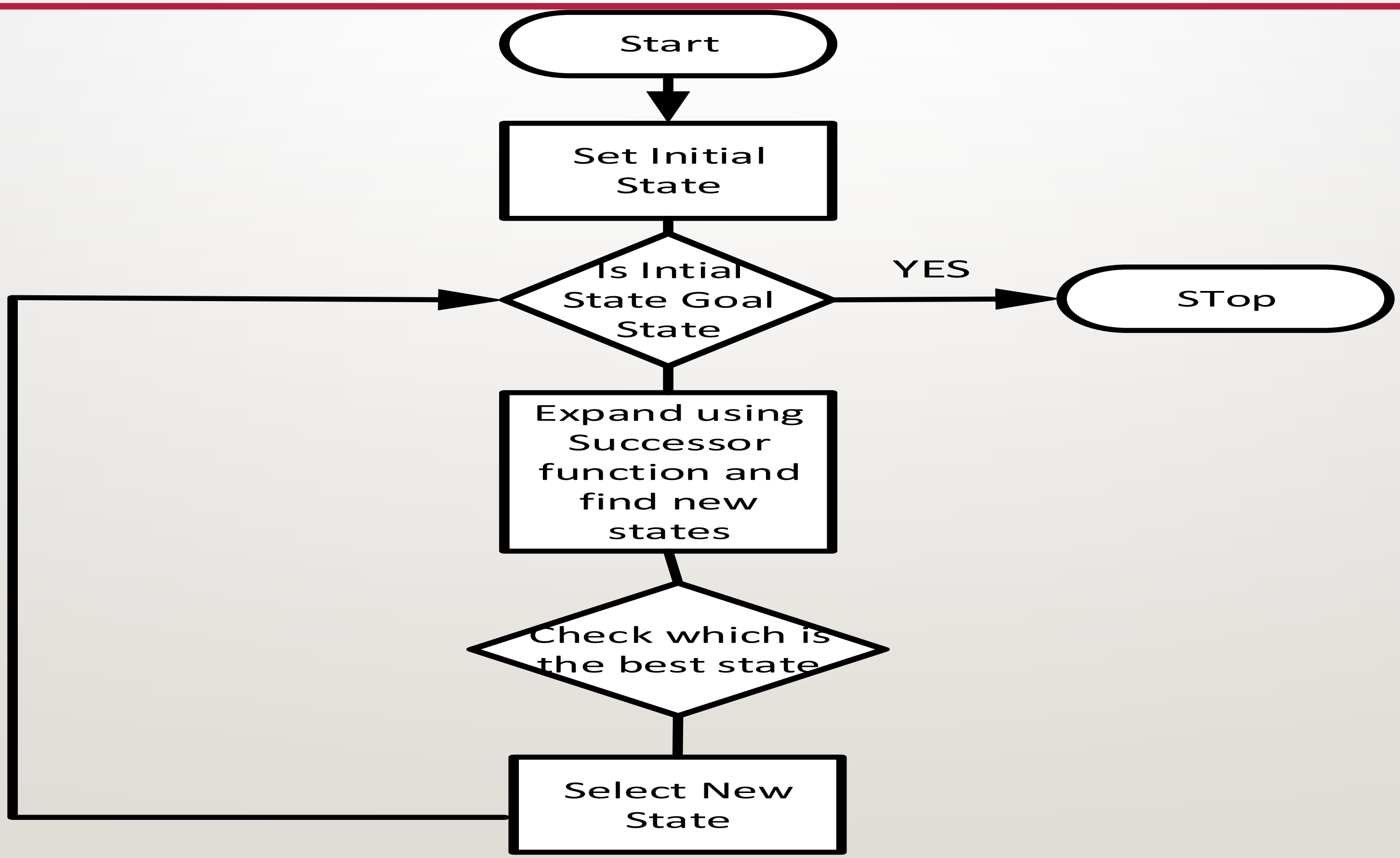
SEARCH ALGORITHMS

Session—4,5

ESSENCE OF SEARCH

- The essence of search in artificial intelligence (AI) revolves around systematically exploring a problem space to find a solution.
- There are a finite number of states in a route-finding problem (20) – One for each city
- There are an infinite number of paths in the state space
- There can be loops in the paths which need to be avoided
- A tree node is a search path, which means there are an infinite number of nodes

PROCESS FOR SEARCHING



SEARCH STRATEGY

- The Choice of a state to expand is called search Strategy.
- It involves systematically applying actions, evaluating states, and navigating through possible paths to find a solution

MEASURING PROBLEM-SOLVING PERFORMANCE

- **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?

MEASURING PROBLEM-SOLVING PERFORMANCE

- **Path cost:** How much does it cost?
- **Search space size:** How many states does it involve
- **Number of states explored:** Explain the extent of the search effort
- **Heuristic Evaluation:** Effectiveness of heuristics?
- **Search Path Length:** Number of steps or actions involved in generating the solution?
- **Search Depth or Breadth:** The largest breadth and depth of the search

SEARCH STRATEGIES

Uninformed Search Strategies

- BFS
- DFS
- Depth Limited Search
- Iterative deepening Search
- Bidirectional search

SEARCH STRATEGIES

Informed search Strategies (Heuristic Search Strategies)

- Best First Search
- Greedy Best First Search
- Informed BFS
- A* Search
- Iterative Deepening A* Search
- Hill Climbing Search (Local Search)
- Memory – Bounded heuristic search

SEARCH STRATEGIES

Local search

- Hill Climbing
- Simulated Annealing

SEARCH STRATEGIES

Adversarial search

- Alpha-Beta Pruning
- MIN-MAX
- Iterative Deepening (BFS+DFS)
- Monte Carlo Tree Search (MCTS)

SEARCH STRATEGIES

Constrained Satisfaction search

- Back Tracking search
- Forword Checking search
- Constraint propagation search
- Heuristic search (Rule Based Search)
- Local Search

Introduction to Un-informed Search

- In this session we discuss several search strategies or methods that are classified into category "uninformed search methods" also called "blind search methods".
- These strategies have no additional information about states beyond what is 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

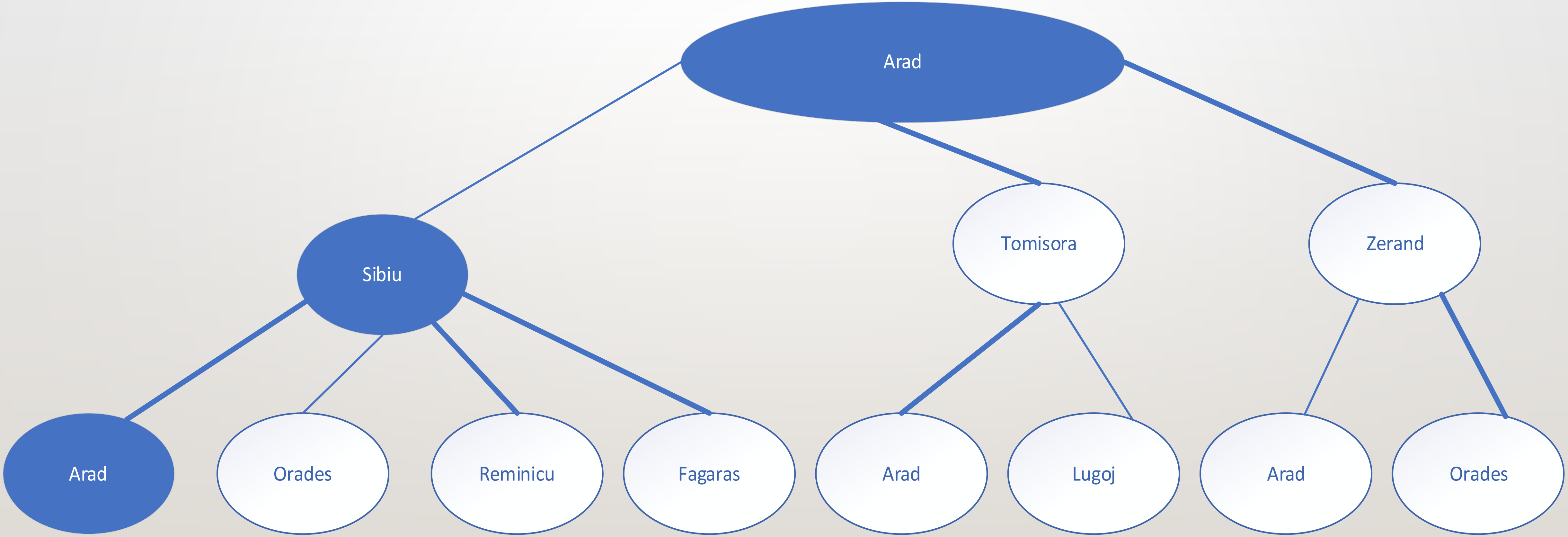
SEARCHING FOR SOLUTIONS

Solutions: A solution is an action sequence, so search algorithms work by considering various action sequences. This can be implemented using search tree.

Search Tree

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.

SEARCHING FOR SOLUTIONS



REPRESENTING NODES

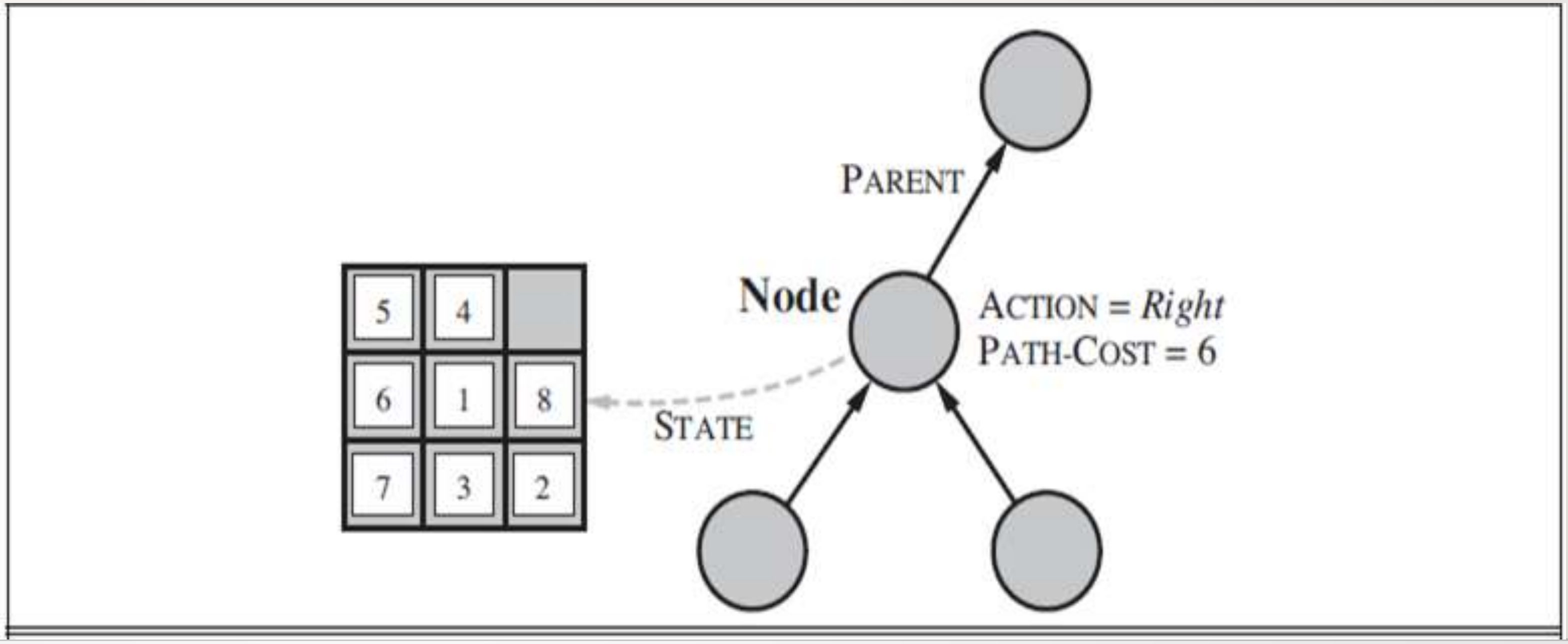
Element	Descriptions
STATE	A State belong to a state space and a node belong to a state
PARENT	Is a NODE that generates the Child Nodes
ACTION	The action executed while in PARENT NODE to generate the CHILD NODE
PATH-COST	The cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers
DEPTH:	The Number of steps along the path from the initial state

DISTINCTION BETWEEN NODE AND STATE

- A Node is a Bookkeeping data structure used to represent a search tree
- A state corresponds to a configuration of the world
- Nodes are on the paths but not the state
- A state can be reached through two different nodes

DISTINCTION BETWEEN NODE AND STATE

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.



Uninformed Search Algorithms

- Uninformed search is a class of general-purpose search algorithms which operates in brute force-way.
- Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Breadth-first Search

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm *starts* searching from the root node of the tree and *expands all successor node at the current level* before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Breadth-first Search

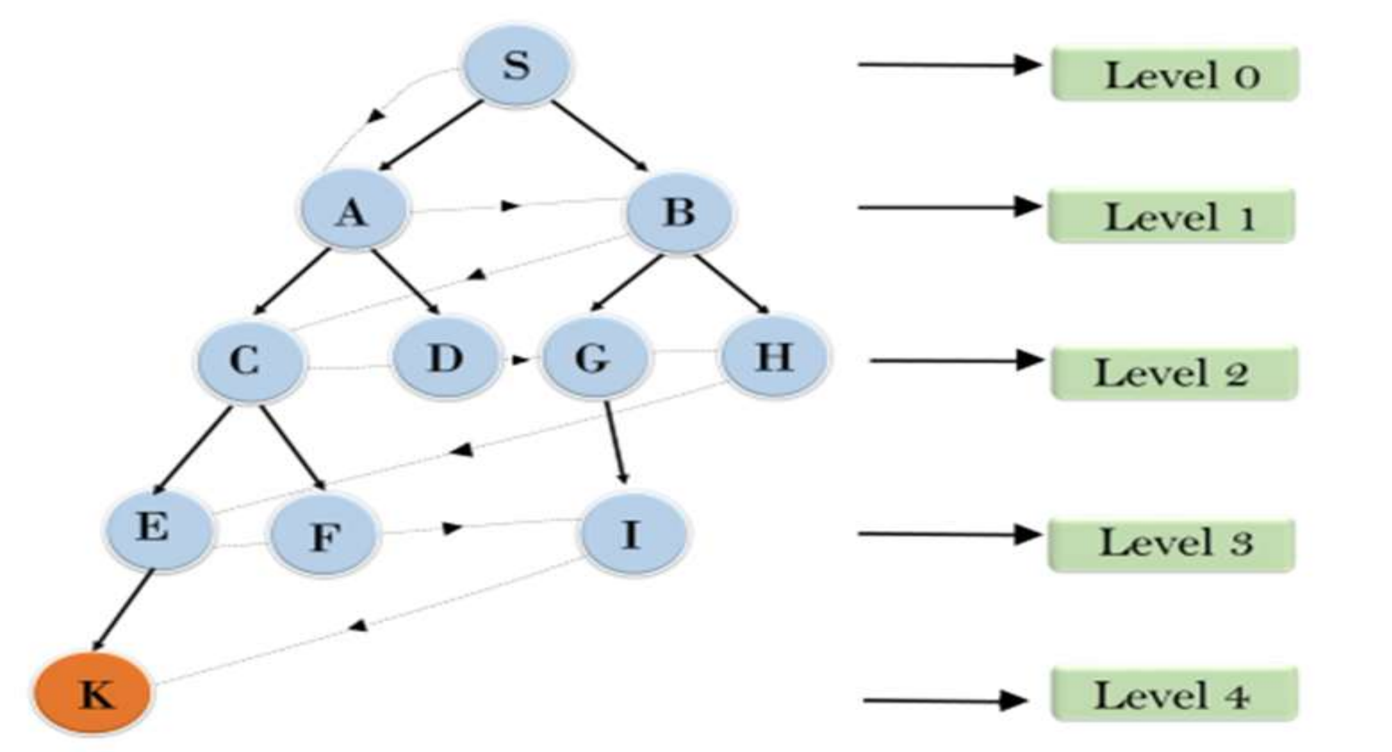
Advantages:

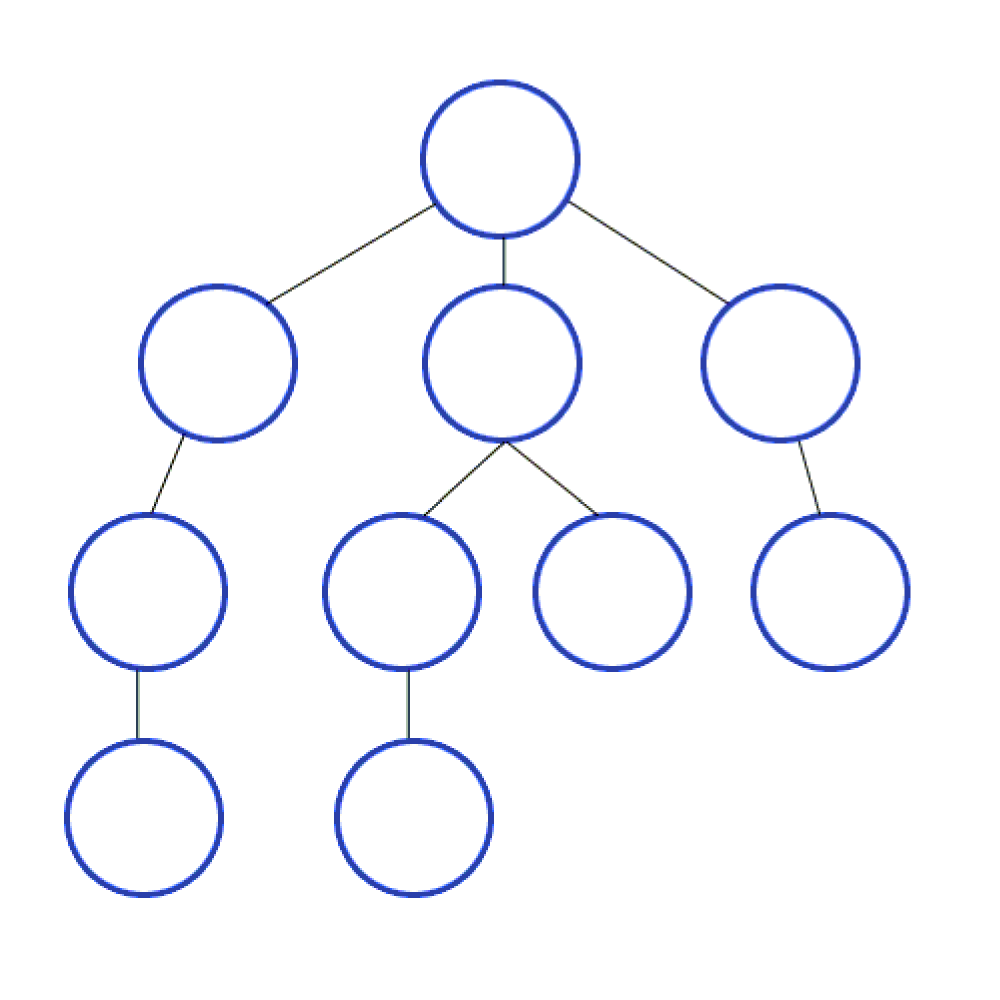
- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

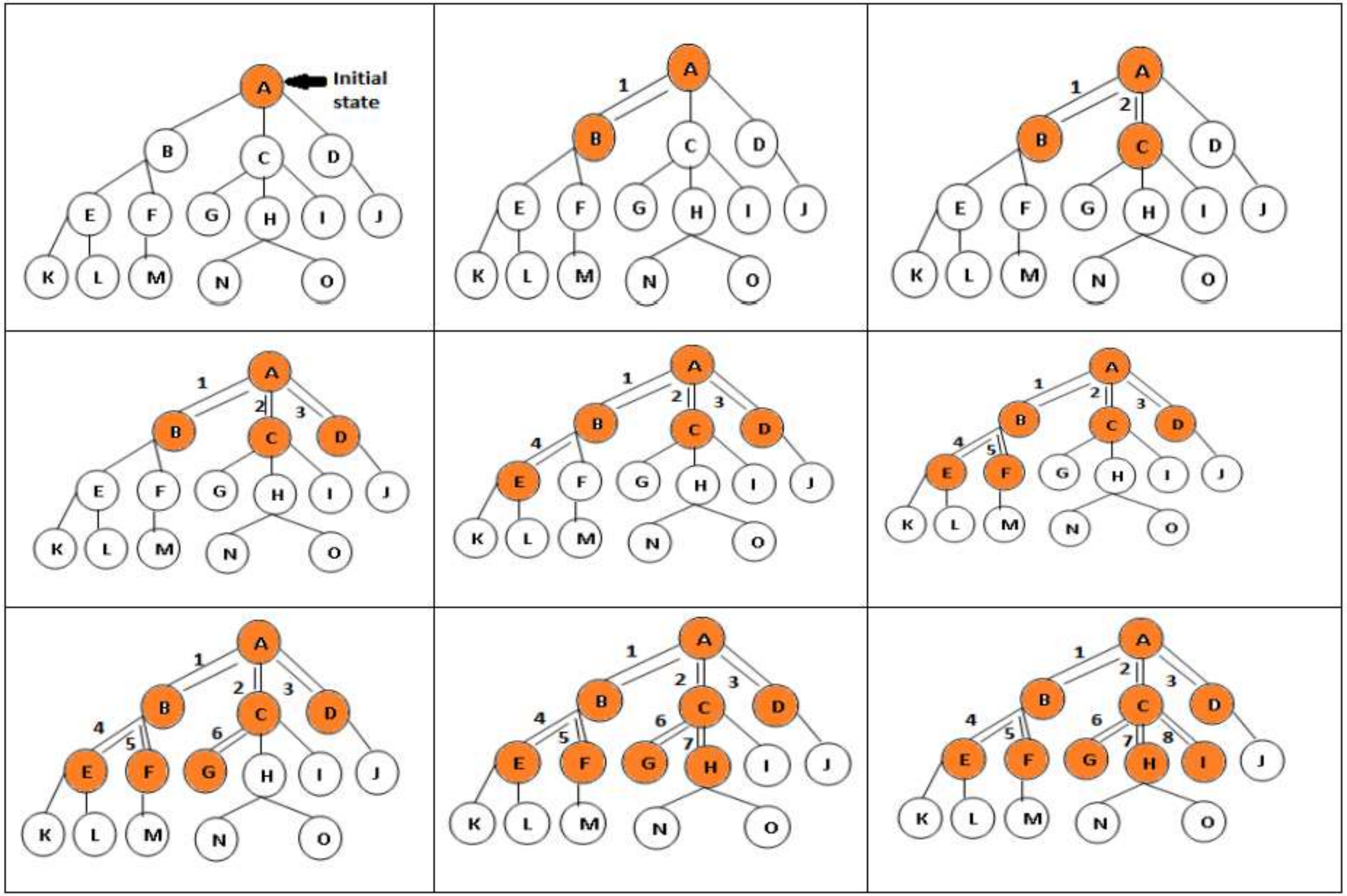
Disadvantages:

- 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.

Breadth-first Search






Initial state
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
1
2
3
4
5
6
7
8

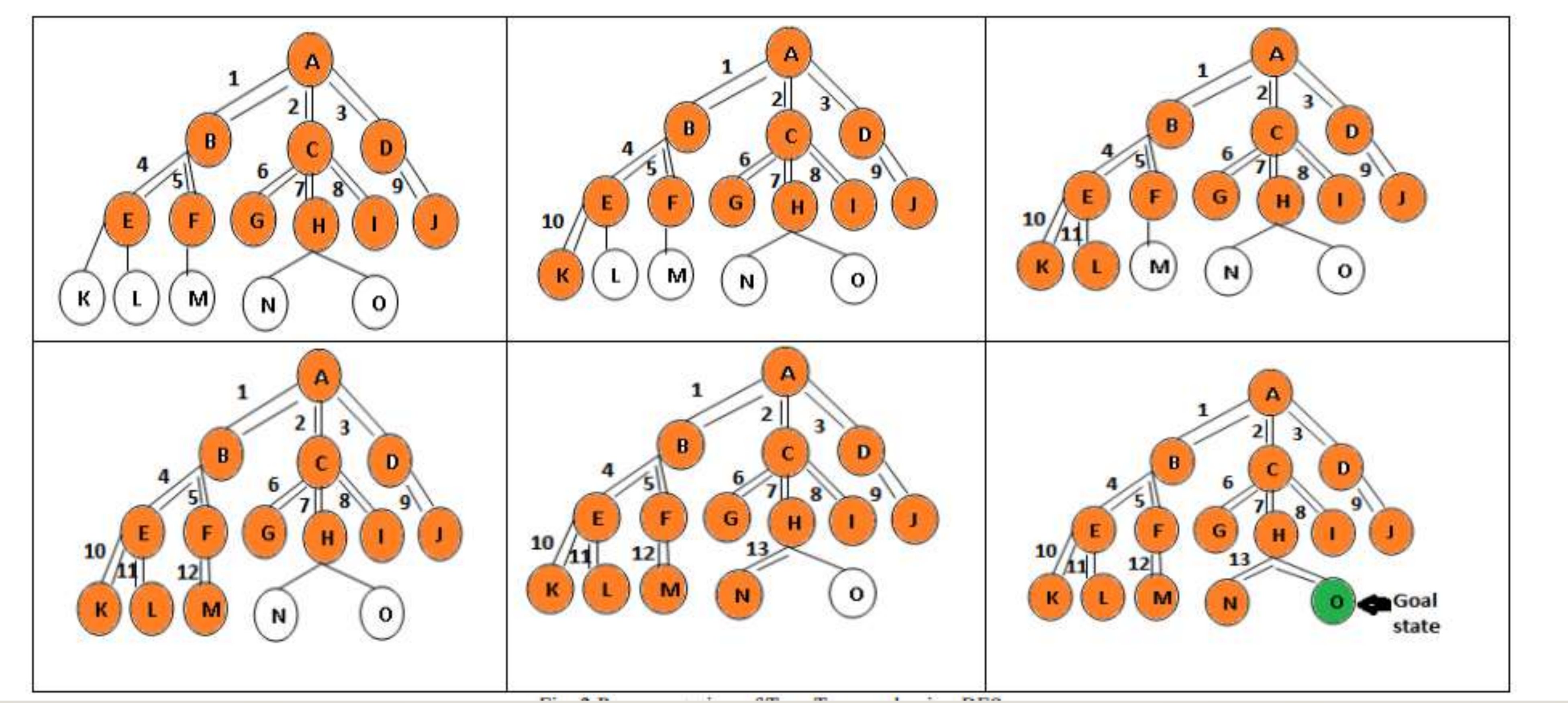

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
Goal
state

TABLE I: OPEN AND CLOSED LIST FOR BFS

Open list (Unexplored nodes)	Close list (Visited nodes)
A	A
B,C,D	B
C,D,E,F	C
D,E,F,G,H,I	D
E,F,G,H,I,J	E
F,G,H,I,J,K,L	F
G,H,I,J,K,L,M	G
H,I,J,K,L,M	H
I,J,K,L,M,N,O	I
J,K,L,M,N,O	J
K,L,M,N,O	K
L,M,N,O	L
M,N,O	M
N,O	N
O ← Goal state	-

Breadth-first Search

Time Complexity:

Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state. **$T(b) = 1+b^2+b^3+.......+ b^d= O(b^d)$**

Space Complexity:

Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness:

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

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

Depth-first Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.

- It is called the depth-first search because it *starts* from the root node and *follows each path to its greatest depth node* before moving to the next path.

- DFS uses a Stack data structure for its implementation.

- The process of the DFS algorithm is similar to the BFS algorithm.

Note: **Backtracking** is an algorithm technique for finding all possible solutions using recursion.

Depth-first Search

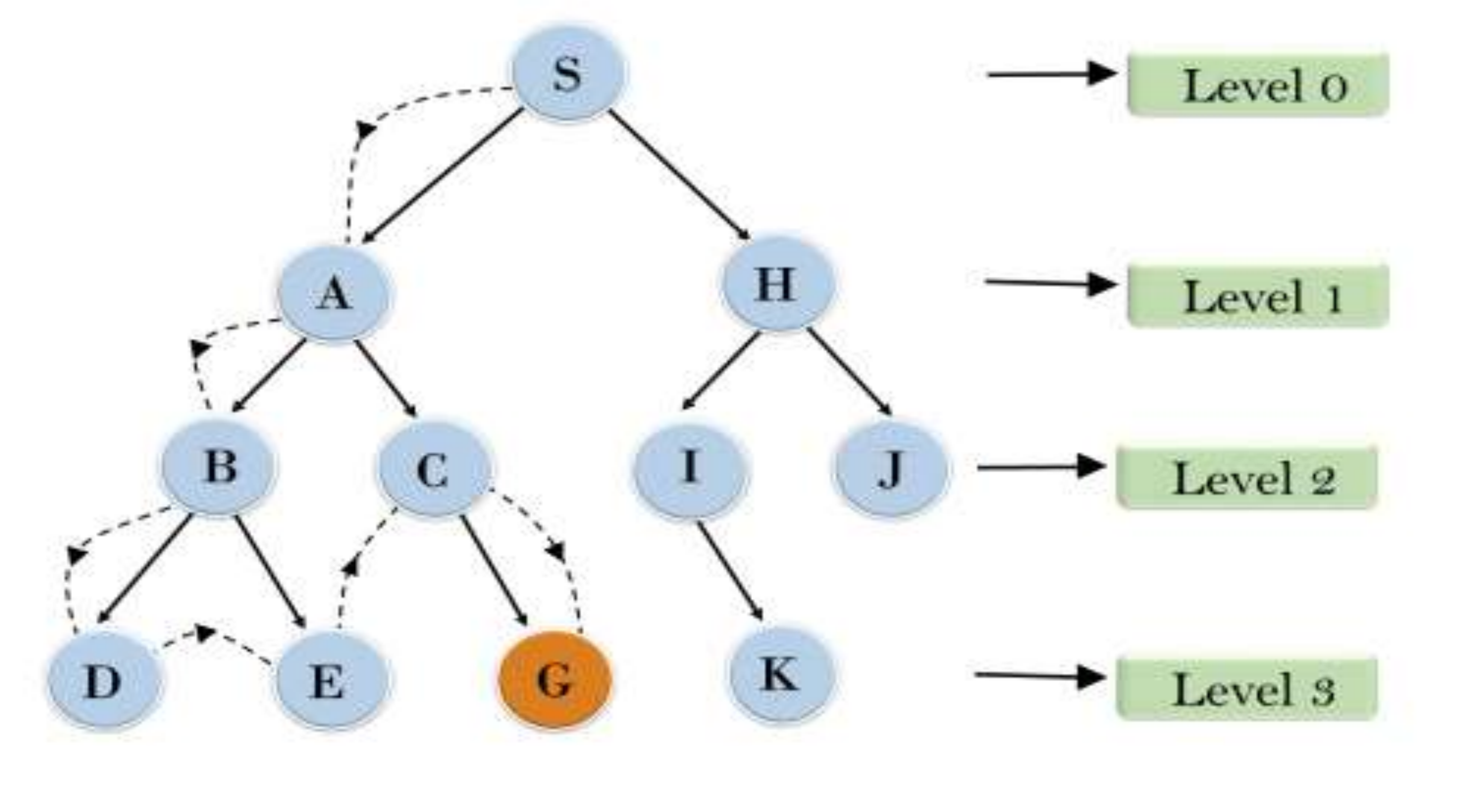
Advantage:

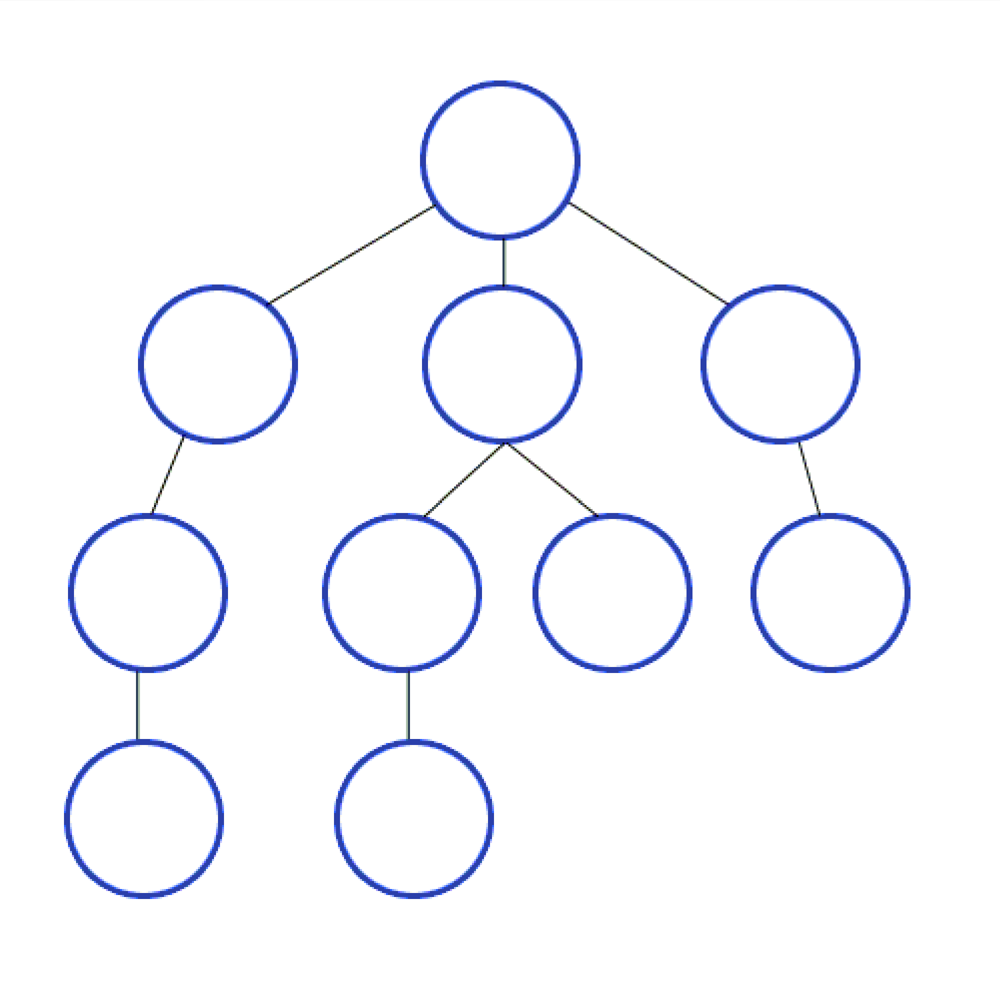
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

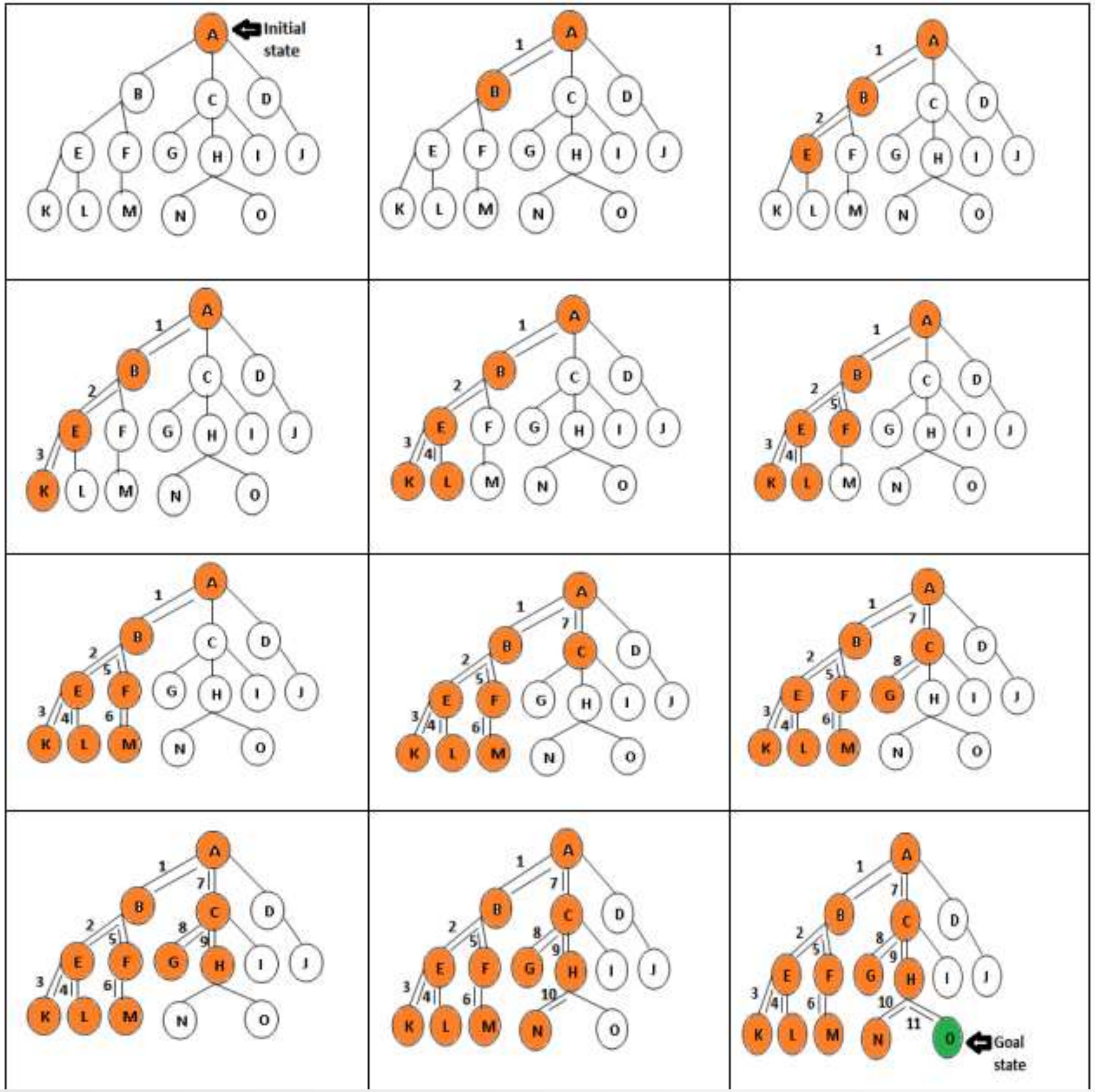
Disadvantage:

- 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 sometime it may go to the infinite loop.

Depth-first Search







Open list (Unexplored nodes)	Close list (Visited nodes)
A	A
B,C,D	B
E,F,C,D	E
K,L,F,C,D	K
L,F,C,D	L
F,C,D	F
M,C,D	M
C,D	C
G,H,I,D	G
H,I,D	H
N,O,I,D	N
O,I,D Goal state	-

Depth-first Search

Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by: $\mathbf{T(n)= 1+ n^2+ n^3 +.........+ n^m=O(n^m)}$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

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 fringe set, which is $\mathbf{O(b^m)}$.

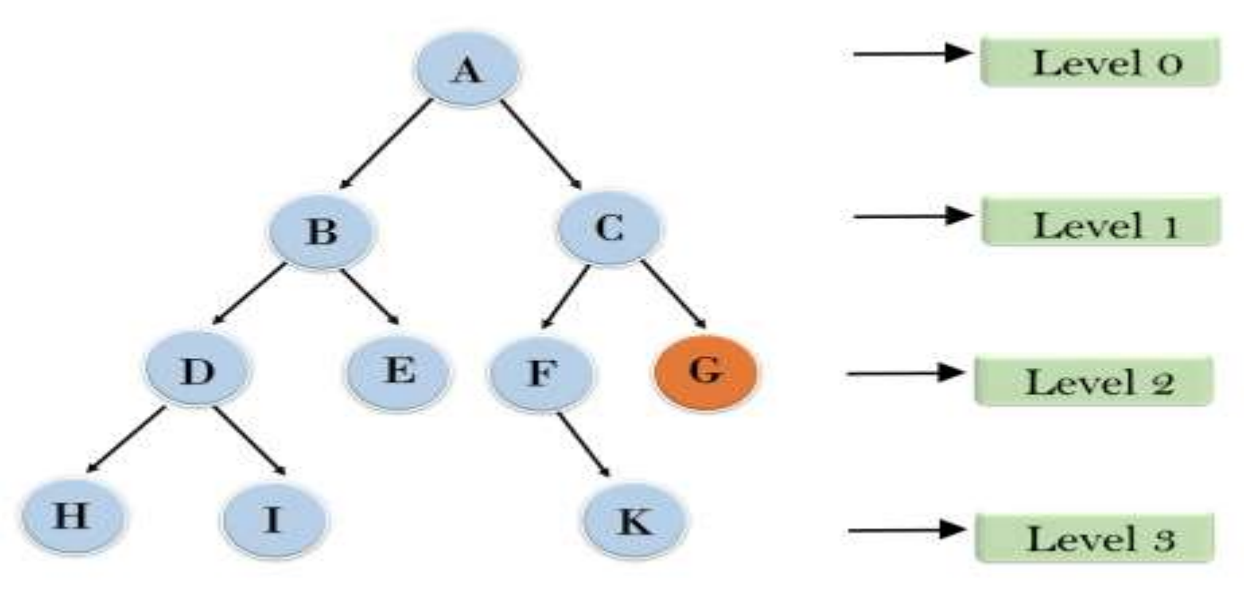
Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Iterative Deepening Search

- Iterative deepening repeatedly calls a depth-bounded searcher, a depth-first searcher that takes in an integer depth bound and never explores paths with more arcs than this depth bound.
- Iterative deepening first does a depth-first search to depth 1 by building paths of length 1 in a depth-first manner. If that does not find a solution, it can build paths to depth 2, then depth 3, and so on until a solution is found.
- When a search with depth bound n fails to find a solution, it can throw away all of the previous computation and start again with a bound of n+1.

Iterative Deepening Search

- Eventually, it will find a solution if one exists, and, as it is enumerating paths in order of the number of arcs, a path with the fewest arcs will always be found first.



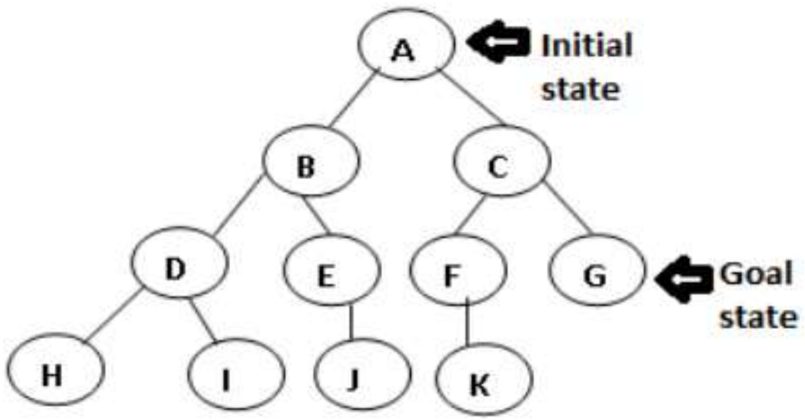
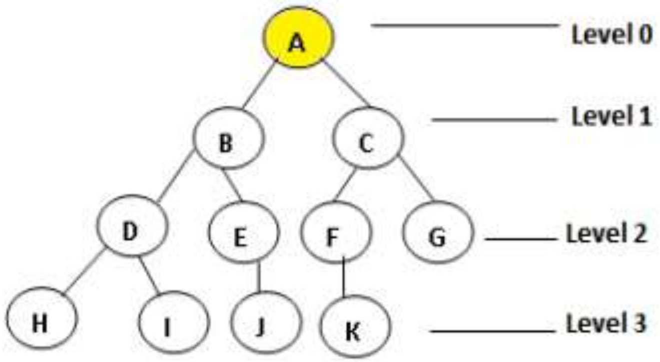
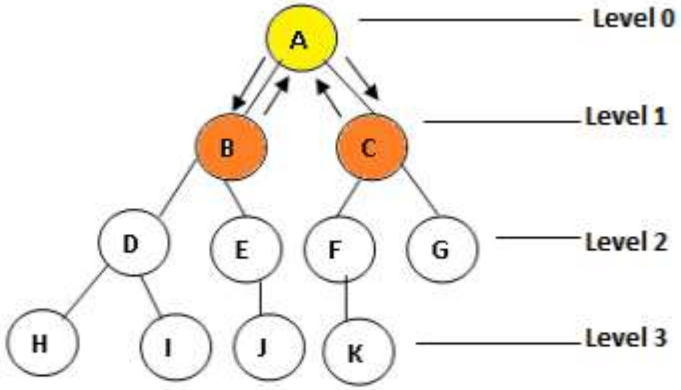


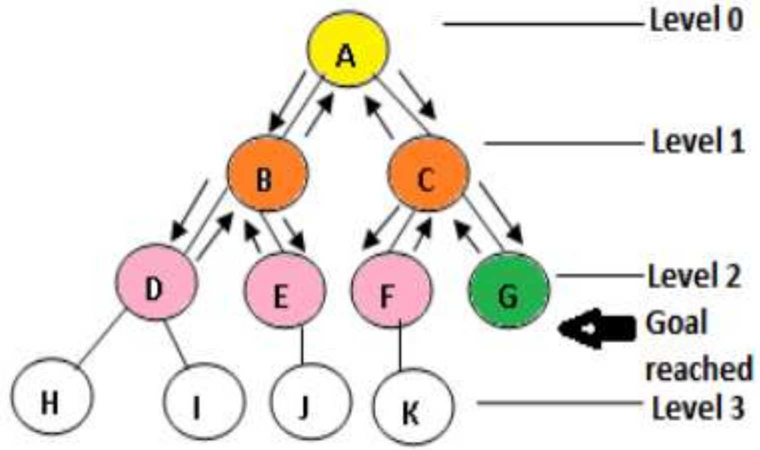
Fig. 5 Graph using IDS



IDS: Depth bound 0 (Level – 0)



IDS: Depth bound 1 (Level – 1)



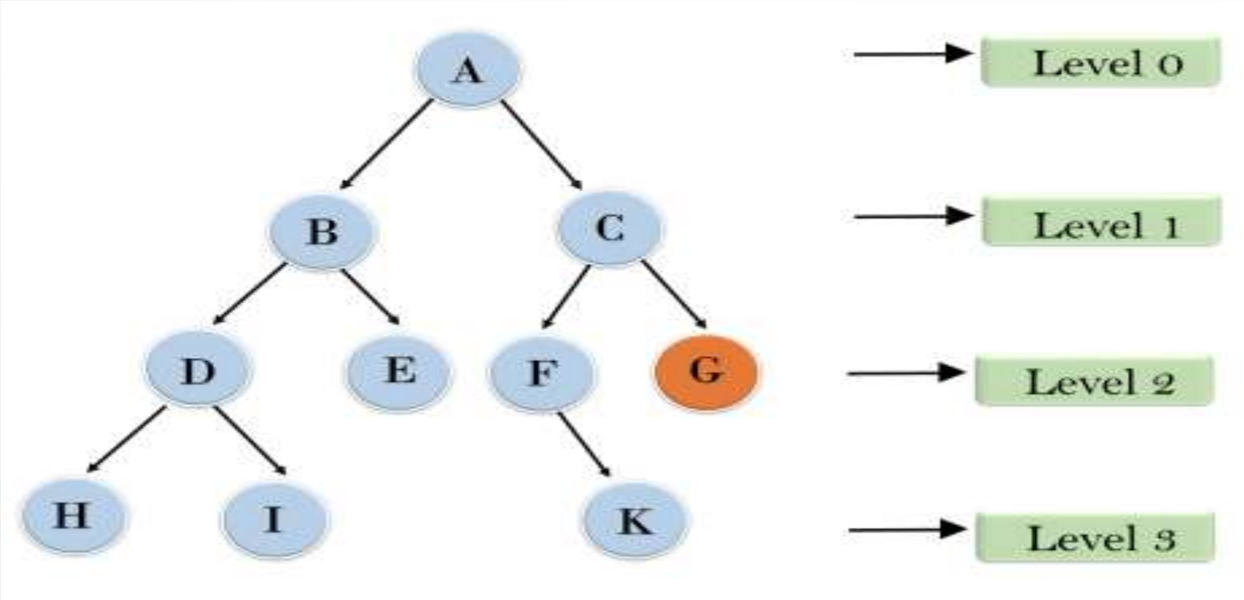
IDS: Depth bound 0 (Level – 0)

Depth (Level)	Iterative Deepening Search
Level – 0	A
Level – 1	A,B,C
Level – 2	A,B,C,D,E,F,G

Iterative Deepening Search

- **Completeness**: IDS is complete, meaning it will find a solution if one exists.
- **Optimality**: IDS is optimal if all actions have the same cost.
- **Time Complexity**: O(b^d), where b is the branching factor and d is the depth of the shallowest solution.
- **Space Complexity**: O(bd), which is more space-efficient than BFS.

Iterative Deepening Search



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

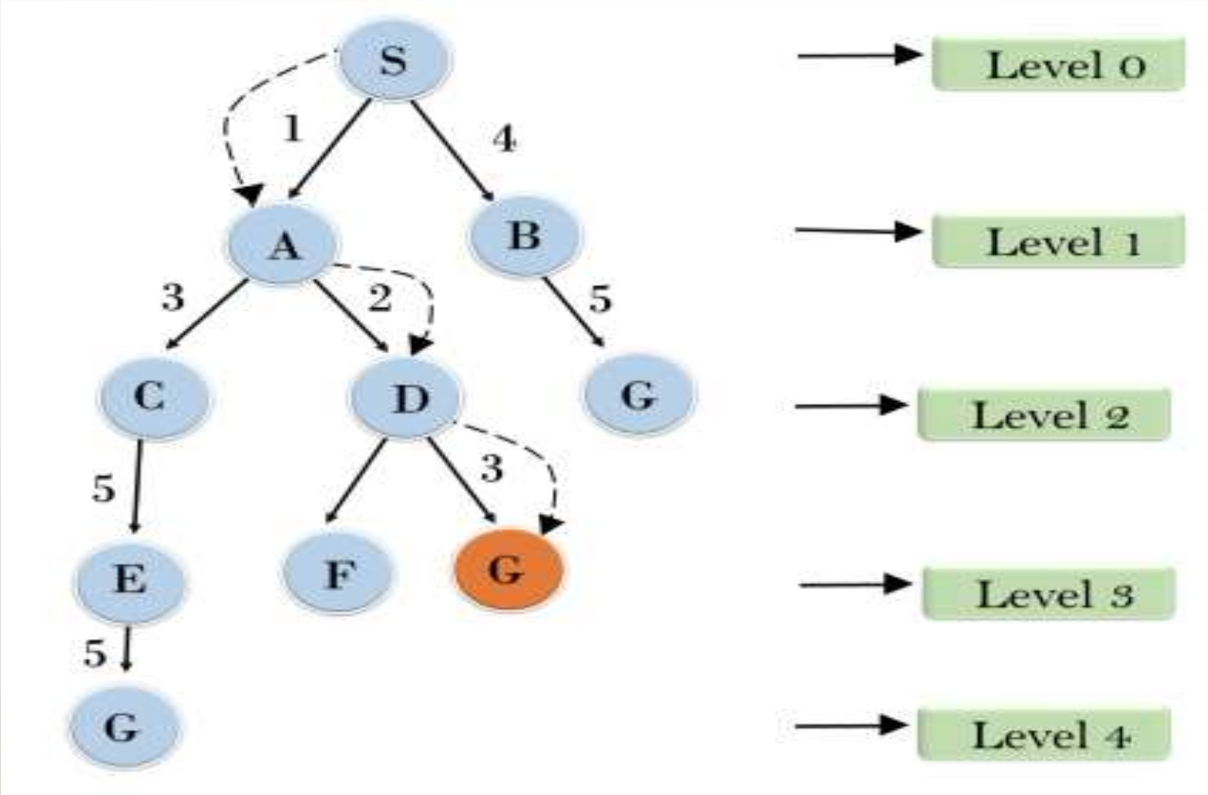
In the fourth iteration, the algorithm will find the goal node.

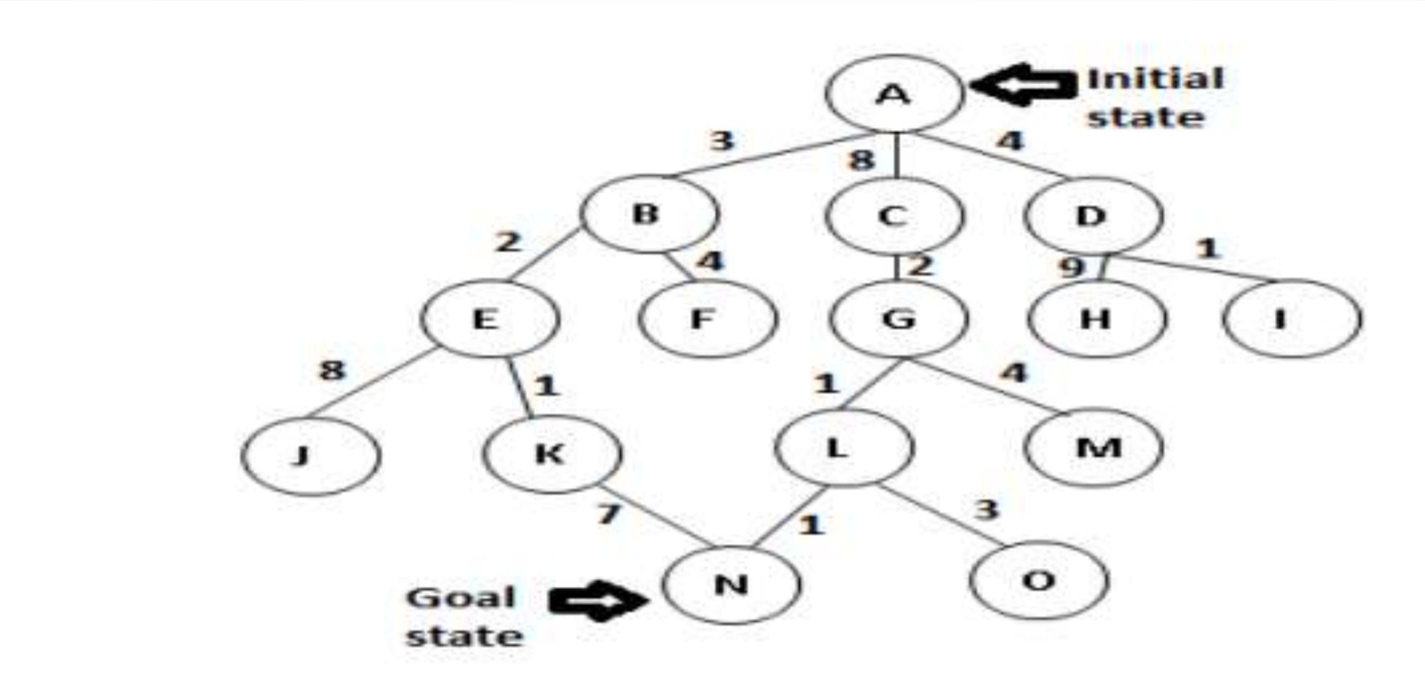
Least-cost search or Uniform-cost Search

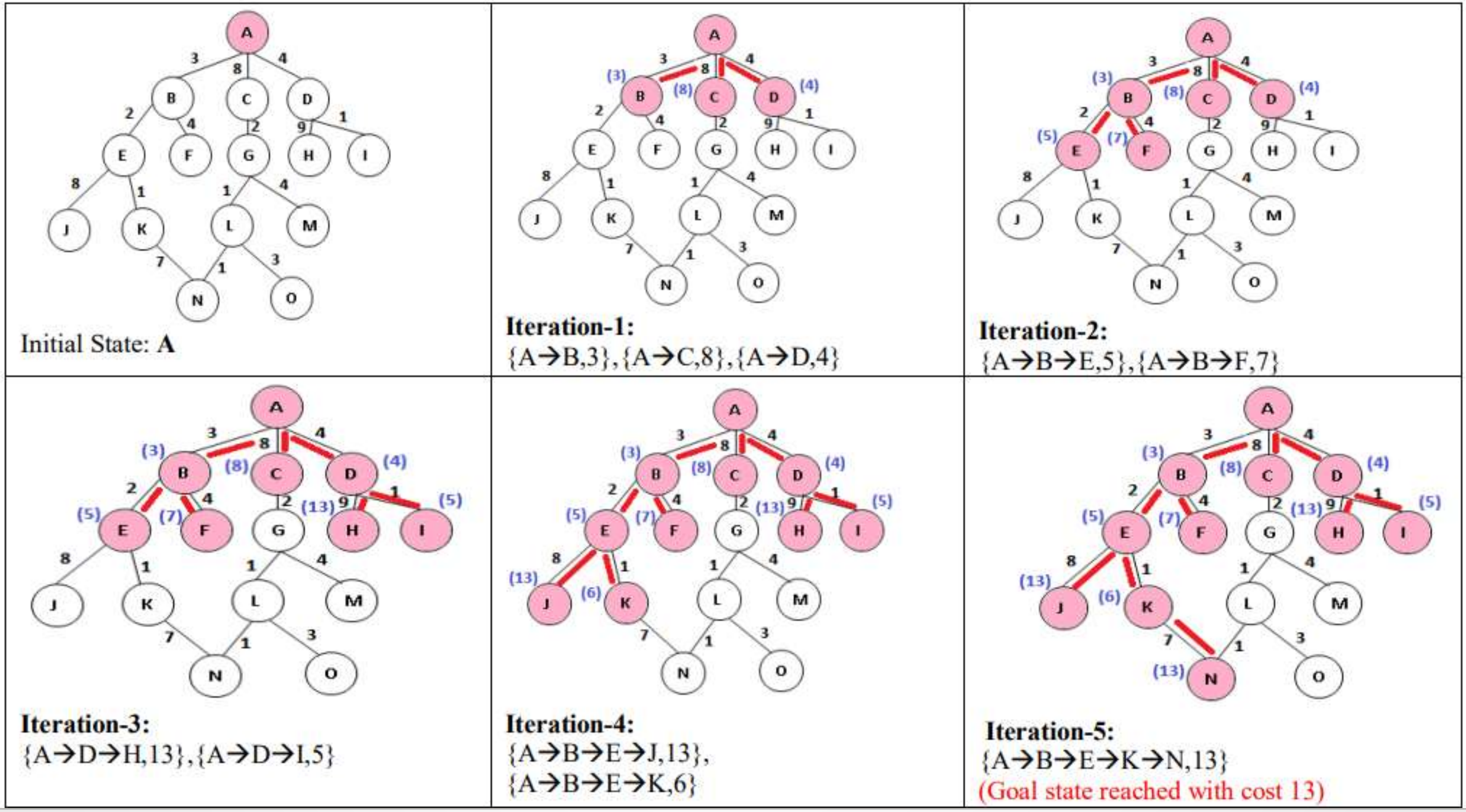
- which is similar to breadth-first search, but instead of expanding a path with the fewest number of arcs, it *selects a path with the lowest cost*. This is implemented by treating the frontier as a priority queue ordered by the *cost function*.
- If the costs of the arcs are all greater than a positive constant, known as ***bounded arc costs***, and the branching factor is finite, the lowest-cost-first search is guaranteed to find an optimal solution – a solution with lowest path cost – if a solution exists.

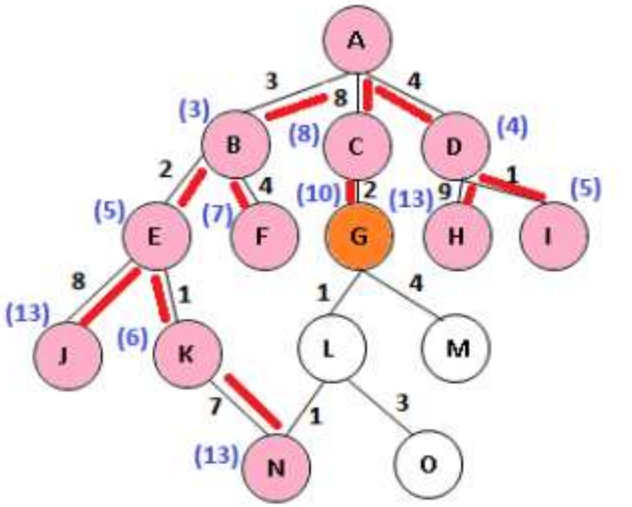
- Insert the root node into the priority queue.
- Remove the element with the highest priority.
- If the removed node is the goal node,
  - print total cost and stop the algorithm
- Else
  - Enqueue all the children of the current node to the priority queue, with their cumulative cost from the root as priority and the current node to the visited list.

Least-cost search or Uniform-cost Search

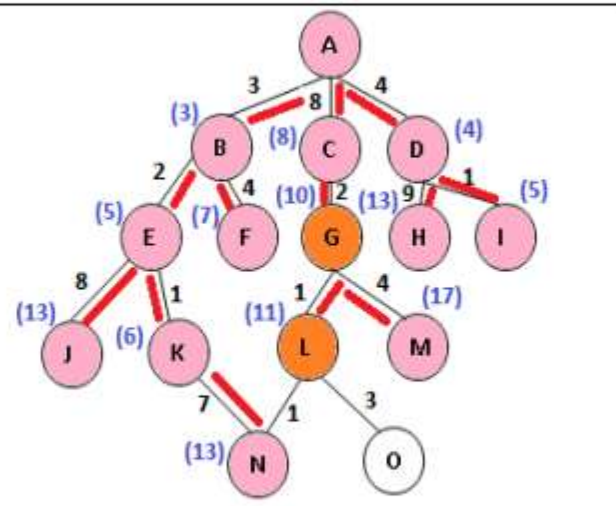



A
Initial state
3
8
4
B
C
D
2
4
2
9
1
E
F
G
H
I
8
1
1
4
J
K
L
M
7
1
3
Goal state
N
O

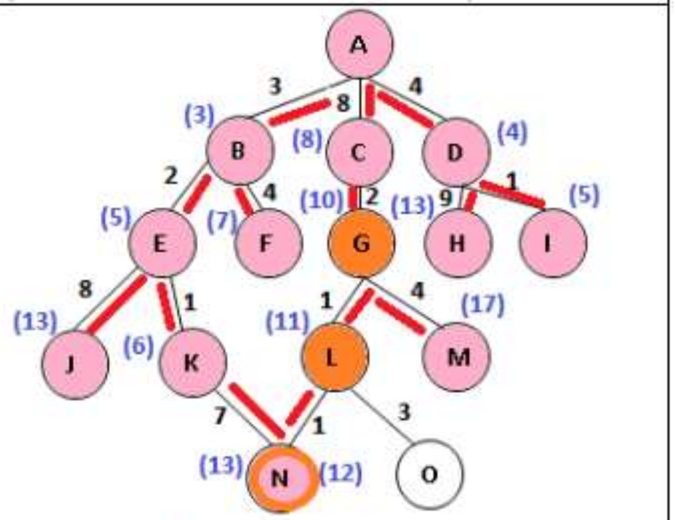

Initial State: **A**
Iteration-1:
{A→B,3},{A→C,8},{A→D,4}
Iteration-2:
{A→B→E,5},{A→B→F,7}
Iteration-3:
{A→D→H,13},{A→D→I,5}
Iteration-4:
{A→B→E→J,13},
{A→B→E→K,6}
Iteration-5:
{A→B→E→K→N,13}
(Goal state reached with cost 13)



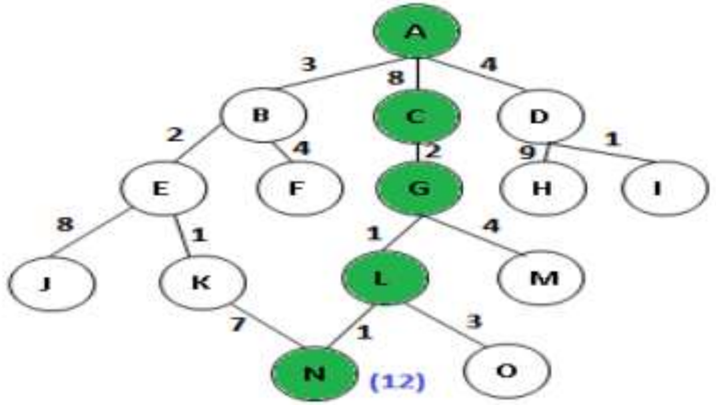
Iteration-6:
{A→C→G,10}



Iteration-7:
{A→C→G→L,11},
{A→C→G→M,17}



Iteration-8:
{A→C→G→L→N,12}
(Goal state reached with cost 12)



Minimum cost from A to N is **{A→C→G→L→N, 12}** .

Iteration	UCS Traversal (Path with cost)
Iteration – 1	{A→B,3},{A→C,8},{A→D,4}
Iteration – 2	{A→B→E,5},{A→B→F,7}
Iteration – 3	{A→D→H,13},{A→D→I,5}
Iteration – 4	{A→B→E→J,13},{A→B→E→K,6}
Iteration – 5	{A→B→E→K→N,13}
Iteration – 6	{A→C→G,10}
Iteration – 7	{A→C→G→L,11},{A→C→G→M,17}
Iteration – 8	{A→C→G→L→N,12}

Bidirectional Search

- Bidirectional Search is an algorithm that simultaneously *searches forward from the initial state and backward from the goal state* until the two searches meet.
- This method can significantly reduce the search time, as each search only needs to explore half the depth of the search space.

Bidirectional Search

- **Completeness**: Bidirectional search is complete if both searches use an algorithm like BFS.
- **Optimality**: Bidirectional search is optimal if both searches use BFS and all step costs are equal.
- **Time Complexity**: O(b^(d/2)), where b is the branching factor and d is the depth of the solution. This is exponentially faster than O(b^d).
- **Space Complexity**: O(b^(d/2)), as it needs to store the nodes at the frontier of both searches.

Bidirectional Search