

When solving problems in AI, we often need to find the best way to reach a goal. This brings us to **uninformed** and **informed** search strategies.

1. Uninformed Search (Blind Search)

Uninformed search doesn't have extra information about which direction to go. It explores the problem step by step, checking all possibilities.

Example:

- Imagine you're in a maze, blindfolded, and you don't know where the exit is. You explore by trying every path one by one until you find the way out.

Common Methods:

- **Breadth-First Search (BFS):** Checks all possible paths at the same distance from the start before moving further.
 - **Depth-First Search (DFS):** Goes as far as possible along one path before backtracking.
-

2. Informed Search (Smart Search)

Informed search uses extra clues, called **heuristics**, to guide the search and make it faster. Heuristics give hints about which path might be better to explore.

Example:

- Now imagine you're in the same maze, but this time you have a map showing where the exit is. You use the map to choose paths that get you closer to the exit.

Common Methods:

- **Greedy Search:** Picks the path that seems closest to the goal based on the clue.
- **A*:** Combines the distance you've already traveled and the estimated distance to the goal for the best choice.

Real-Life Example:

- **Uninformed Search:** Finding a restaurant by walking down every street in the area.
- **Informed Search:** Using Google Maps to find the shortest route to the restaurant.

In short, uninformed search is like guessing, while informed search is like solving with smart clues!

Feature	Uninformed Search	Informed Search
Knowledge	No problem-specific knowledge	Uses heuristics or domain knowledge
Efficiency	Often explores more paths (less efficient)	Explores fewer paths (more efficient)
Examples	BFS, DFS, UCS	Greedy Best-First, A*
Complexity	Higher for large spaces	Lower due to heuristics
Applications	Generic problems	Problems with clear heuristics

Applications of Search Algorithms

1. Route Planning and Navigation
2. Game Development
3. Robotics Pathfinding
4. Web Search Engines
5. Medical Diagnosis Systems
6. Logistics and Supply Chain Optimization
7. Natural Language Processing (NLP)
8. Scheduling and Planning
9. Puzzle Solving
10. AI for Mental Health Support

Properties of Search Algorithms:

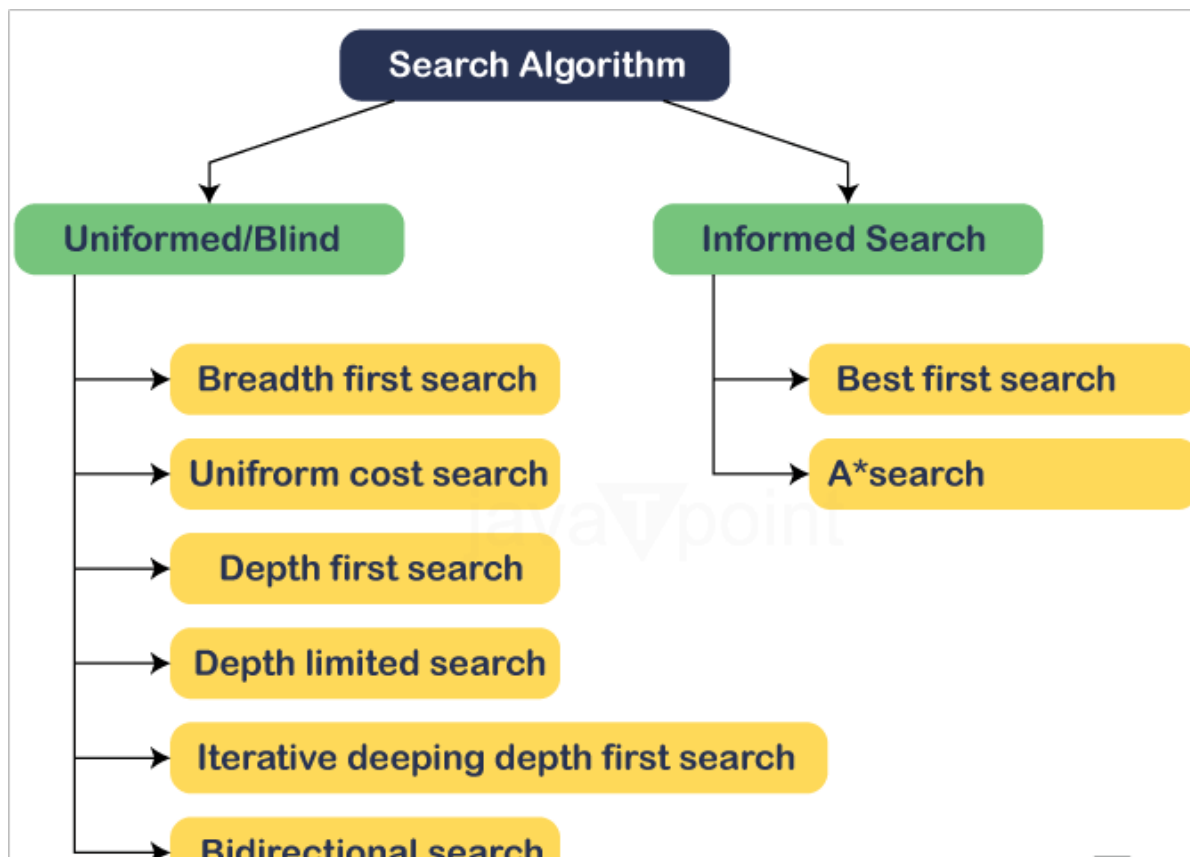
Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

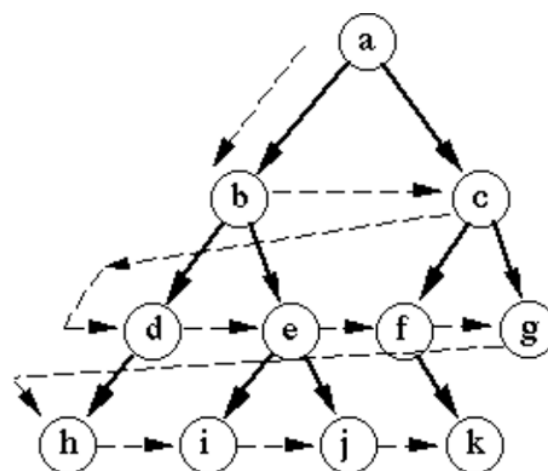
Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.



Breadth First Search

- In Breadth First Search(BFS), the root node of the graph is expanded first, then all the successor nodes are expanded and then their successor and so on i.e. the nodes are expanded level wise starting at root level.
- For Example :



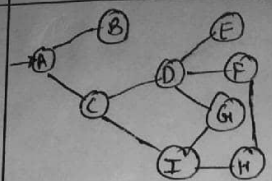
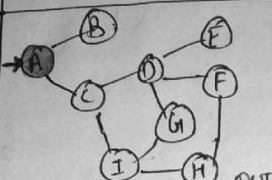
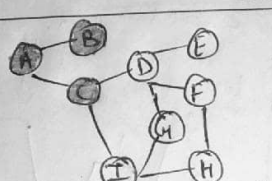
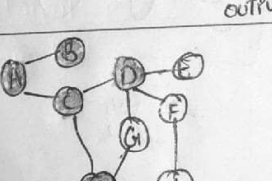
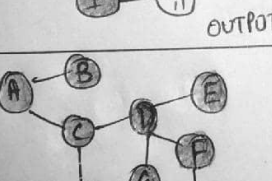
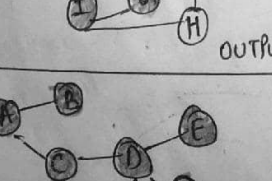
Breadth-first search

Advantages

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

Disadvantages

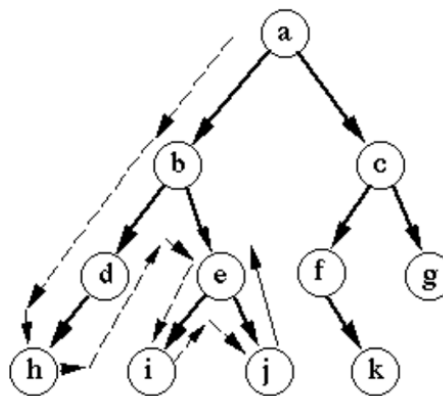
- It requires lots of memory since each level of the tree must be saved in memory to expand to the next level.
- BFS needs lots of time if the solution is far away from the root node

Steps	Graph Traversal	Description
1)	 <p>Queue [] (Empty)</p> <p>OUTPUT → NONE</p>	Initialization of Queue.
2)	 <p>Queue [] (Empty)</p> <p>OUTPUT → A</p>	Let, A be the starting node and mark it as Visited.
3)	 <p>Queue [B C]</p> <p>OUTPUT → A B C</p>	Two unvisited nodes B and C can now be Enqueued in the Queue and further adjacent neighbours can be visited.
4)	 <p>Queue [D I]</p> <p>OUTPUT → A B C D I</p>	Dequeue B and C. B does not have any adjacent node, hence adjacent nodes of C i.e. D and I are Enqueued in the Queue.
5)	 <p>Queue [E F G]</p> <p>OUTPUT → A B C D I E F G</p>	Now, D is dequeued and adjacent of D i.e. E, F & G are Enqueued into Queue and marked as visited.
6)	 <p>Queue [H]</p> <p>OUTPUT → A B C D I E F G H</p>	Visit adjacent node of 'F' as 'E' does not have any. 'H' will be Enqueued in the Queue and hence all the nodes will be marked as visited.

Searching Strategies : Breadth First Search

Depth First Search

- In Depth First Search(DFS), the deepest node is expanded in the current unexplored node of the search tree. The search goes into depth until there is no more successor nodes.
- For Example :



Depth-first search

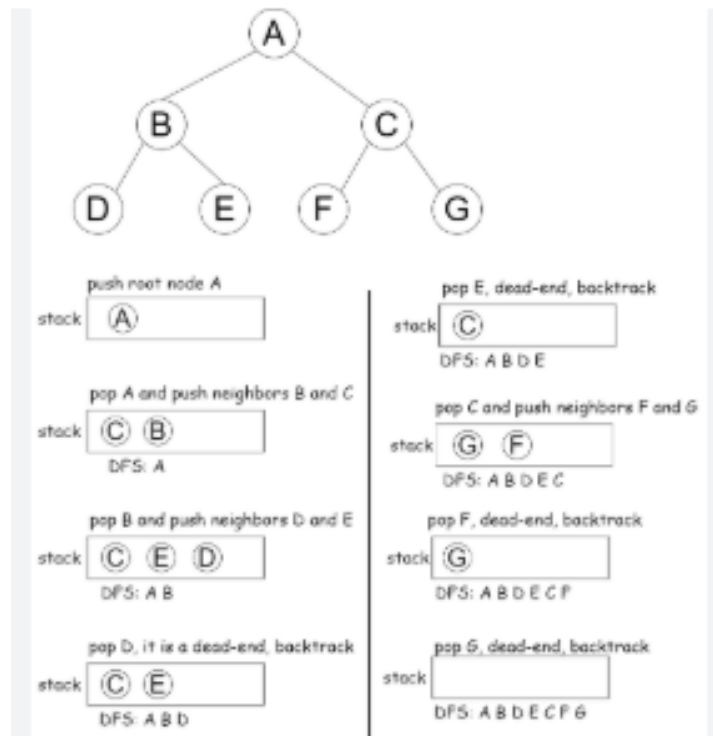
DFS:

Push source vertex into the stack

while stack is not empty:

 pop the top of the stack and,

 push all its unvisited neighbors into the stack



Searching Strategies : Depth First Search

Advantages

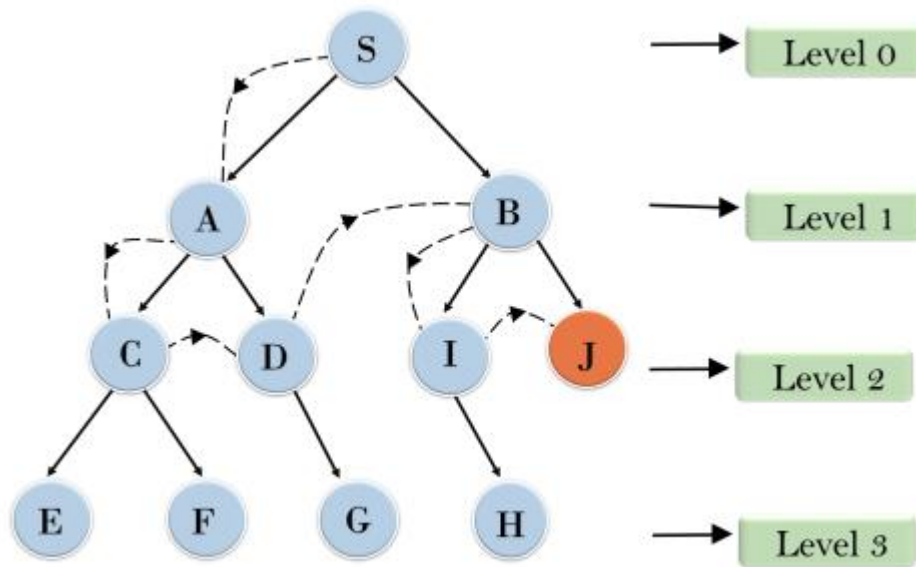
- DFS requires very little memory as it only needs to store a stack of the nodes on the path from the root node to the current node.

Disadvantages

- The DFS algorithm goes for deep-down searching, and sometimes it may go to the infinite loop. (if using graph representation than tree)

Depth Limited Search (DLS)

DLS is an uninformed search algorithm. This is similar to DFS but differs only in a few ways. The sad failure of DFS is alleviated by supplying a depth-first search with a predetermined depth limit. That is, nodes at depth are treated as if they have no successors. This approach is called a depth-limited search.



Advantages

- Depth-limited search is Memory efficient.

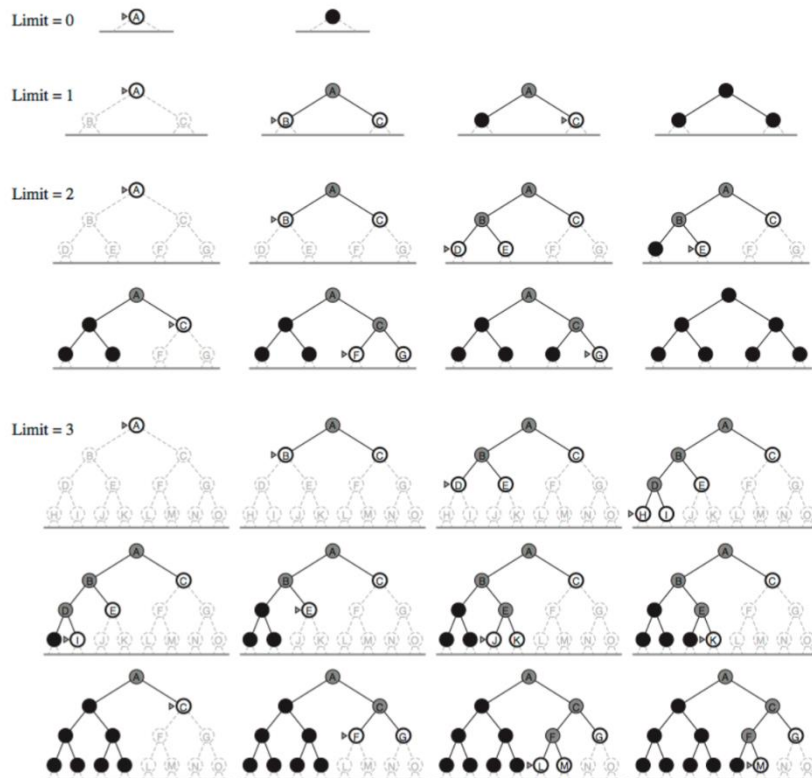
Disadvantages

- The DLS has disadvantages of non-completeness and is not optimal if it has more than one goal state.

Iterative Deepening Depth First Search (IDDFS)

It is a search algorithm that uses the combined power of the BFS and DFS algorithms. It is iterative in nature. Searches for the best depth in each iteration. It performs the Algorithm until it reaches the goal node. The algorithm is set to search until a certain depth and the depth keeps increasing at every iteration until it reaches the goal state.

Iterative-Deepening Search



The primary advantage of iterative deepening is its ability to achieve the completeness of Breadth-First Search (BFS) while using minimal memory, similar to Depth-First Search (DFS), making it ideal for problems with large search spaces where memory is a constraint; essentially, it combines the best aspects of both BFS and DFS by progressively increasing the search depth, ensuring a solution is found if one exists while keeping memory usage low.

Advantages

- It combines the benefits of BFS and DFS search algorithms in artificial intelligence in terms of fast search and memory efficiency.

Disadvantages

- The main drawback of IDDFS is that it repeats all the work from the previous phase.

Uniform Cost Search (UCS) Algorithm

Uniform Cost Search (UCS) explores graphs by expanding nodes from the start to the goal based on edge costs. It finds the lowest-cost path, essential when step costs vary for optimal solutions.

- Use Priority Queue

- Sorted increasing order of path cost
- Guaranteed to find optimal solution.
- UCS expands node with least path cost $g(n)$ so far

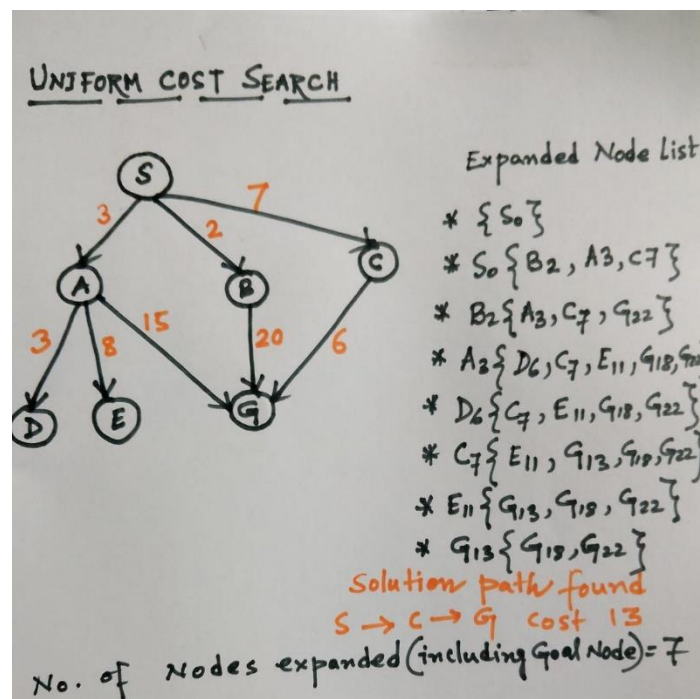
When all steps are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost $g(n)$. This is done by storing the frontier as a priority queue ordered by g .

Advantages

- Uniform cost search is an optimal search method because at every state, the path with the least cost is chosen.

Disadvantages

- It does not care about the number of steps or finding the shortest path involved in the search problem, and it is only concerned about path cost. This algorithm may be stuck in an infinite loop.



Example:

Suppose you have a graph where:

- The optimal path cost C^* is 10.
- The smallest edge cost ϵ is 2.
- The branching factor b is 3.

The time complexity becomes: $O(b^{1+\lceil C^*/\epsilon \rceil}) = O(3^{1+\lceil 10/2 \rceil}) = O(3^{1+5}) = O(3^6)$.

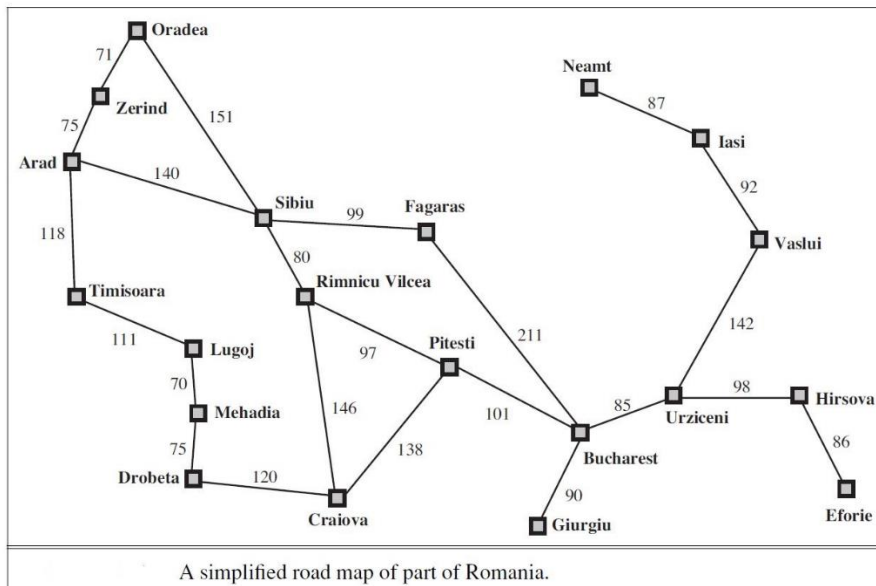
If ϵ were smaller, say $\epsilon = 1$, the time complexity would be: $O(3^{1+\lceil 10/1 \rceil}) = O(3^{1+10}) = O(3^{11})$.

Assignment I: Solve this problem by using Uniform Cost Search Algorithm

Start Node: Arad

Goal Node: Bucharest

Find The optimal path, if any.



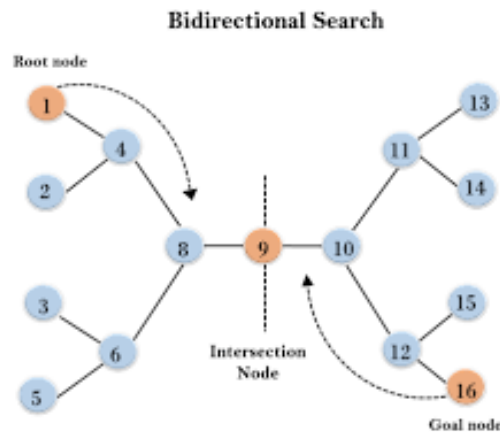
Bidirectional Search

Searching a graph is quite famous problem and have a lot of practical use. We have already discussed [here](#) how to search for a goal vertex starting from a source vertex using [BFS](#). In normal graph search using BFS/DFS we begin our search in one direction usually from source vertex toward the goal vertex, **but what if we start search from both direction simultaneously.**

Bidirectional search is a graph search algorithm which find smallest path from source to goal vertex. It runs two simultaneous search –

1. Forward search from source/initial vertex toward goal vertex
2. Backward search from goal/target vertex toward source vertex

Bidirectional search replaces single search graph(which is likely to grow exponentially) with two smaller sub graphs – one starting from initial vertex and other starting from goal vertex. **The search terminates when two graphs intersect.**



Why bidirectional approach?

Because in many cases it is faster, it dramatically reduce the amount of required exploration. Suppose if branching factor of tree is **b** and distance of goal vertex from source is **d**, then the normal BFS/DFS searching complexity would be $O(b^d)$. On the other hand, if we execute two search operation then the complexity would be $O(b^{d/2})$. for each search and total complexity would be $O(b^{d/2} + b^{d/2})$, which is far less than $O(b^d)$.

Advantages

- Since BS uses various techniques like DFS, BFS, DLS, etc., it is efficient and requires less memory.

Disadvantages

- In bidirectional search, one should know the goal state in advance.

Performance measures

- Completeness : Bidirectional search is complete if BFS is used in both searches.
- Optimality : It is optimal if BFS is used for search and paths have uniform cost.
- Time and Space Complexity : Time and space complexity is $O(b^{d/2})$.

Algorithm	Completeness	Optimality	Time Complexity	Space Complexity
Breadth-First Search	Yes	Yes (unweighted)	$O(b^d)$	$O(b^d)$
Uniform-Cost Search	Yes	Yes	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^{1+\lceil C^*/\epsilon \rceil})$
Depth-First Search	No	No	$O(b^m)$	$O(m)$
Depth-Limited Search	Yes (limited)	No	$O(b^l)$	$O(l)$
Iterative Deepening DFS	Yes	Yes	$O(b^d)$	$O(d)$
Bidirectional Search	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$

b -> branching factor

d -> depth of the shallowest solution

m -> max. depth of search tree

l -> depth limit

a -> complete if b is finite

b -> complete if step costs $\geq E$ for $E > 0$

c -> optimal if step costs are same

d -> if both directions use BFS