Uninformed Search Algorithms

DEFINE :

Uninformed search algorithms, also known as blind search algorithms, are a category of search algorithms used in artificial intelligence and computer science to explore and navigate through a problem space without having any prior information about the structure or nature of that space. These algorithms do not use heuristics or domain-specific knowledge to guide their search but instead systematically explore all possible options until a solution is found or a termination condition is met.

List or search alorithms :

- Breadth-First SEARCH
- Depth - limited SEARCH
- lterative deepening Search
- uniform cost search
- bidirectional search

BREADTH-FIRST SEARCH

DEFINE :

"Breadth-First Search" (BFS), which is a graph traversal algorithm used in computer science to explore and analyze the structure of a graph or tree. BFS starts at the root node (or an arbitrary node as the starting point) and explores all its neighbors at the current depth level before moving on to the next depth level. It's often used to find the shortest path between two nodes in an unweighted graph and also to explore all connected components in a graph.

STEPS :

Input: A graph G and a starting vertex s.

Output: The traversal order of all vertices in G starting from s.

Initialization:

Create an empty queue Q and an empty set or array to keep track of visited vertices, let's call it visited.
Enqueue the starting vertex s into the queue Q.
Mark s as visited and add it to the visited set or array.

BFS Algorithm:

Repeat until the queue Q is empty:

Step 1: Dequeue a vertex v from the front of the queue Q.

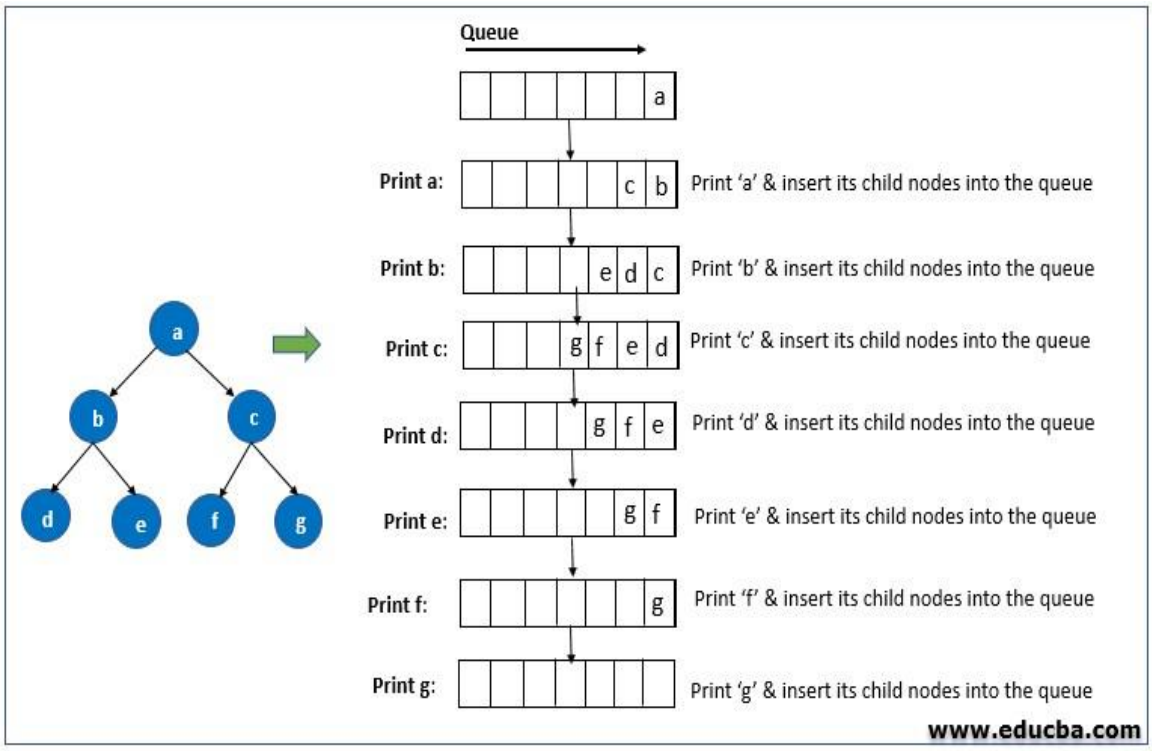
Step 2: Process the dequeued vertex v. You can print it or perform any desired operation with it.

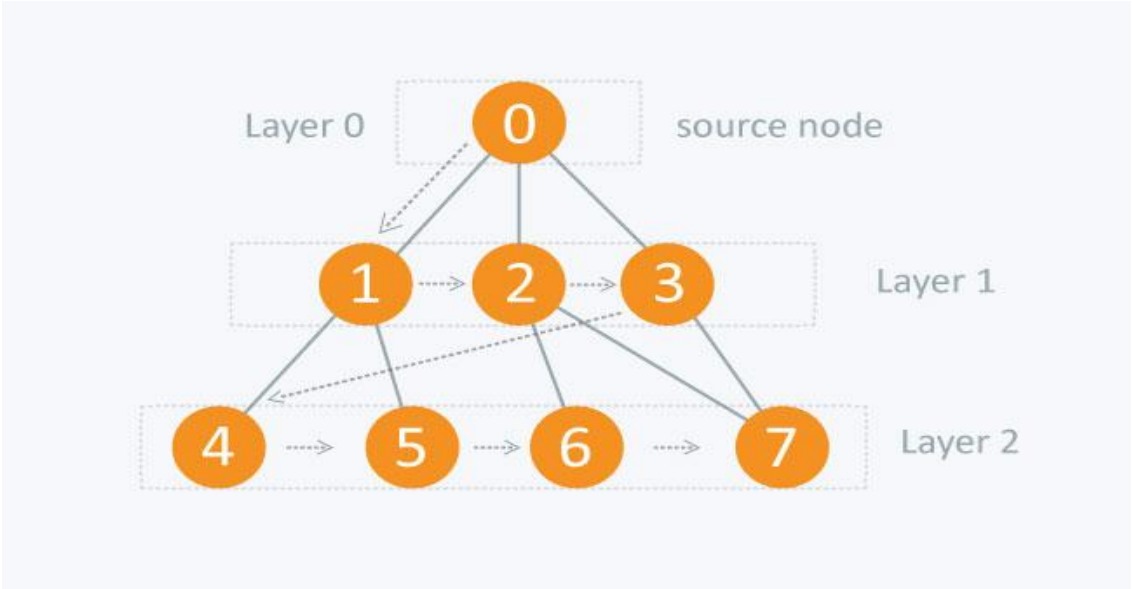
Step 3: For each neighbor u of v that has not been visited yet:

a. Mark u as visited.
b. Enqueue u into the queue Q.

Step 4: Repeat Steps 1 to 3 until the queue Q is empty.
Result: The BFS traversal order will be the order in which vertices were processed in Step 2.

EXAMPLE :


Queue
a
Print a:
c b
Print 'a' & insert its child nodes into the queue
Print b:
e d c
Print 'b' & insert its child nodes into the queue
Print c:
g f e d
Print 'c' & insert its child nodes into the queue
Print d:
g f e
Print 'd' & insert its child nodes into the queue
Print e:
g f
Print 'e' & insert its child nodes into the queue
Print f:
g
Print 'f' & insert its child nodes into the queue
Print g:
Print 'g' & insert its child nodes into the queue
www.educba.com
a
b
c
d
e
f
g


Layer 0
0
source node
1
2
3
Layer 1
4
5
6
7
Layer 2

DEPTH-LIMITED SEARCH

DEFINE :

Depth-limited search is a variant of depth-first search (DFS), which is an algorithm used for traversing or searching tree or graph data structures. In a depth-limited search, the search algorithm limits the depth of exploration in the search tree or graph, meaning it only explores nodes up to a certain depth level before backtracking. This limitation is useful for preventing the algorithm from going too deep into the tree or graph, which can be particularly important in cases where the search space is very large or infinite.

STEPS :

Input:

Starting node (usually the root node).
Depth limit (the maximum depth to search to).
Goal state (if searching for a specific goal).

Output:

A solution (if found) or an indication that no solution exists.

Algorithm:

Initialize:

Set the initial node as the current node. Initialize a stack (or recursion if you prefer) for keeping track of nodes to explore.Set the depth to 0.

Check for Goal:

If the current node is the goal state, return a solution.

Check Depth Limit:

If the current depth is equal to the depth limit, backtrack. (This is important for depth limitation.)

Expand Nodes:

Generate all child nodes of the current node.Push these child nodes onto the stack (or ecursively call the search on each child node).

Depth-First Exploration:

Pop the next node from the stack.Set this node as the current node.Increment the depth by 1.

Repeat:

Repeat steps 2-5 until one of the following conditions is met:

*A solution is found (current node is the goal state), return the solution.

*The depth limit is reached, backtrack.

*The stack is empty, indicating that the search has explored all possible nodes within the depth limit without finding a solution. In this case, terminate and report that no solution exists.

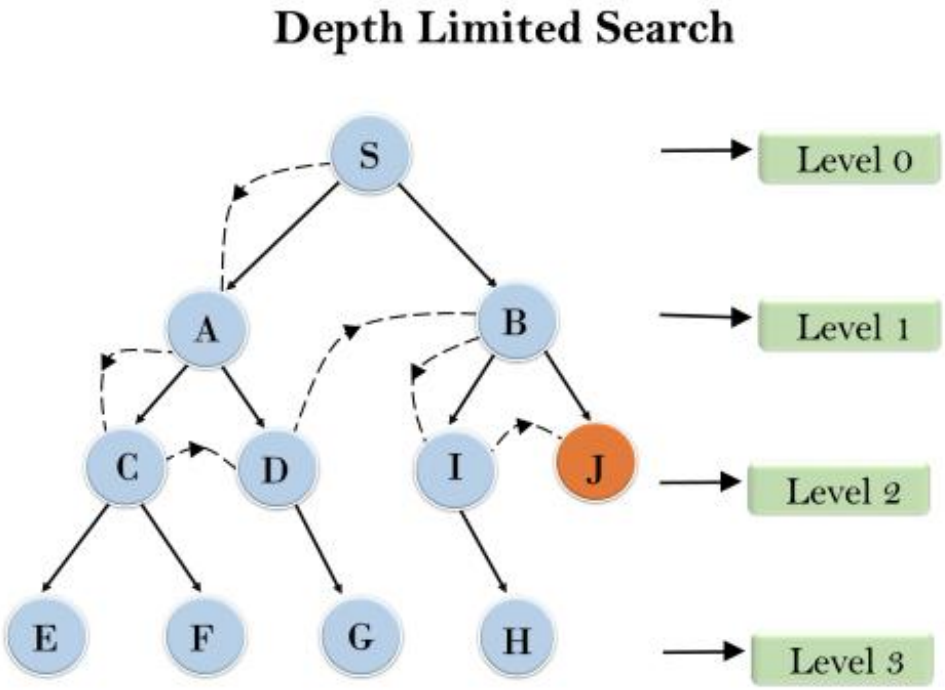
Termination Conditions:

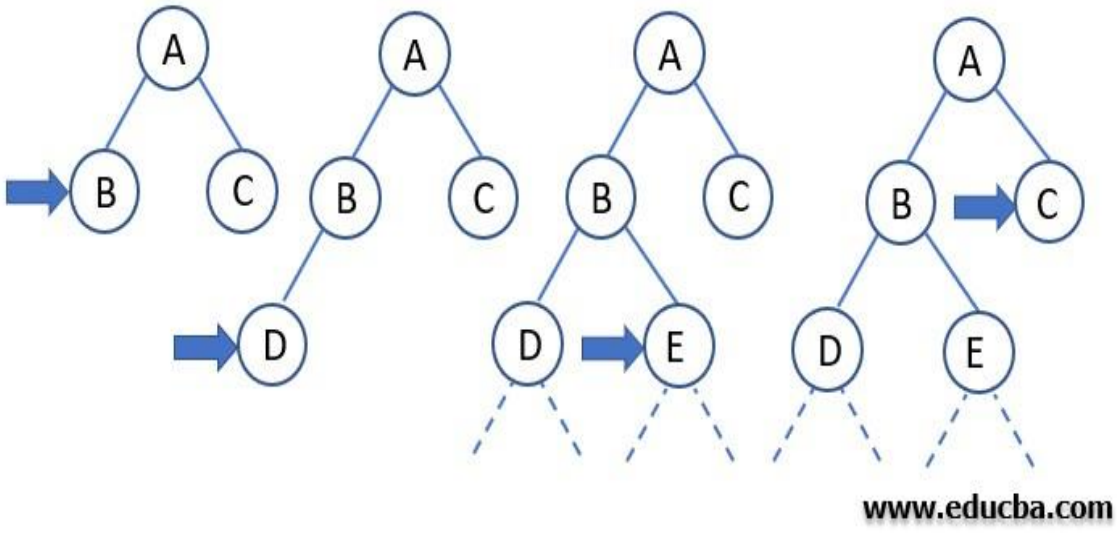
*If a solution is found, return it.

*If the depth limit is reached without finding a solution, backtrack.

*If the stack becomes empty and no solution is found within the depth limit, terminate and report that no solution exists.

EXAMPLE :


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3


A
B
C
D
E
www.educba.com

Iterative Deepening Search

DEFINE :

Iterative Deepening Search (IDS) is a graph traversal and search algorithm used in computer science to find a target node in a tree or graph. It combines the benefits of both Breadth-First Search (BFS) and Depth-First Search (DFS) while avoiding some of their limitations. The main idea behind IDS is to perform a series of depth-limited searches, increasing the depth limit with each iteration until the target node is found.

STEPS :

Step 1:

Initialize the depth limit to 0 and set a flag indicating whether the target node has been found to false.

Step 2:

Perform a Depth-First Search (DFS) starting from the root node, but limit the depth of the search to the current depth limit.

Step 3:

If the target node is found at the current depth limit, set the flag indicating that the target has been found to true and stop the search.

Step 4:

If the target node is not found at the current depth limit, increase the depth limit by 1 and repeat from Step 2.

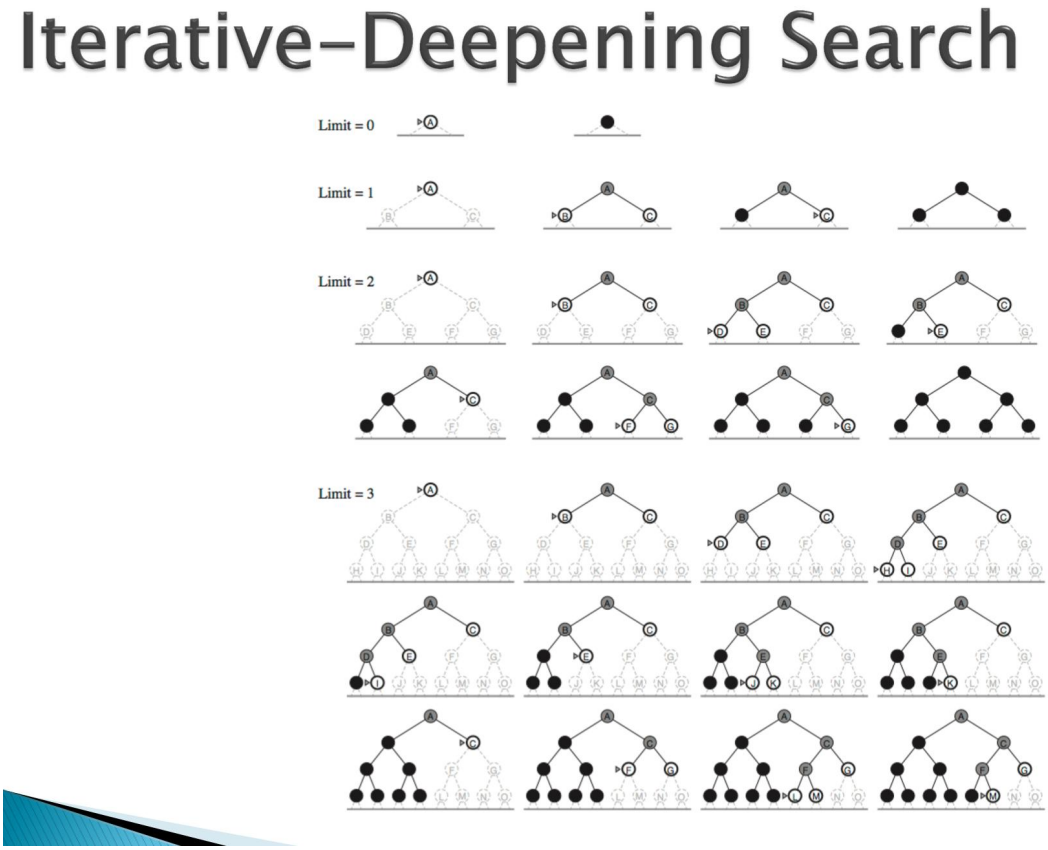
Step 5:

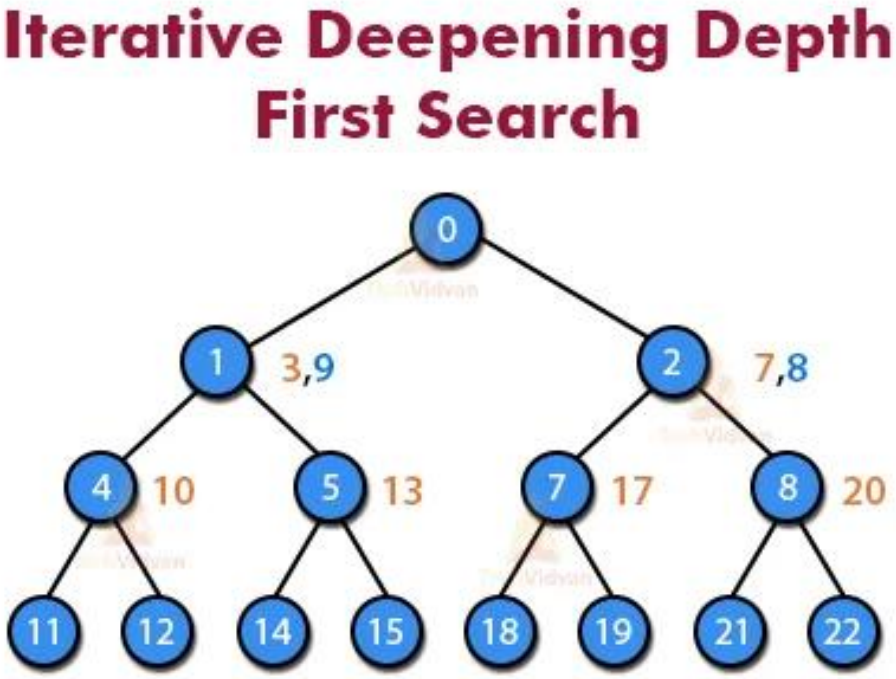
Repeat Steps 2-4 until the target node is found or until it's clear that the target node doesn't exist in the graph (in which case, the search will terminate when the depth limit exceeds the depth of the deepest node in the graph).

Step 6:

If the target node has been found, return the path from the root to the target node.

EXAMPLE :


Iterative-Deepening Search
Limit = 0
Limit = 1
Limit = 2
Limit = 3


Iterative Deepening Depth First Search
0
1
3,9
2
7,8
4
10
5
13
7
17
8
20
11
12
14
15
18
19
21
22

UNIFORM COST SEARCH

DEFINE :

Uniform Cost Search (UCS) is a graph search algorithm used in computer science and artificial intelligence to find the shortest path from a starting node to a goal node in a weighted graph. It's an uninformed search algorithm, meaning it doesn't use any heuristic information about the problem, but instead explores the search space systematically, always choosing the path with the lowest cumulative cost.

STEPS :

Input:

1. A graph, represented as a set of nodes and edges.

2 . A start node.

3 . A goal node.

4 . Edge costs, which are associated with the edges between nodes. These costs can represent distances, time, or any other measure you want to minimize.

Initialization:

1 . Create an empty priority queue (often implemented using a data structure like a min-heap) called the "frontier" to store nodes to be expanded. Initialize it with the start node and a cost of 0.

2 . Create an empty set called the "explored" set to keep track of nodes that have been visited.

Algorithm:

1 . While the frontier is not empty:

1 . 1 Pop the node with the lowest cumulative cost from the frontier. This node is the one with the currently lowest cost path to it.
1 . 2 If the popped node is the goal node, you have found the shortest path. Return it.

1 . 3 Otherwise, mark the current node as "explored" and expand it:

1 . 3 . 1 Generate all of its child nodes (neighbors) that have not been explored.

1 . 3 . 2 For each child node:

1 . 3 . 2 . 1 Calculate the cost to reach that child node from the start node through the current node.

1 . 3 .2 . 2If the child node is not in the frontier or has a lower cost than the cost stored in the frontier, add it to the frontier with the new lower cost.

2 . If the frontier becomes empty and you haven't found the goal node, there is no path from the start node to the goal node, indicating that there is no solution.

Output:

* If a path to the goal node is found, return the sequence of nodes from the start node to the goal node, which represents the shortest path.

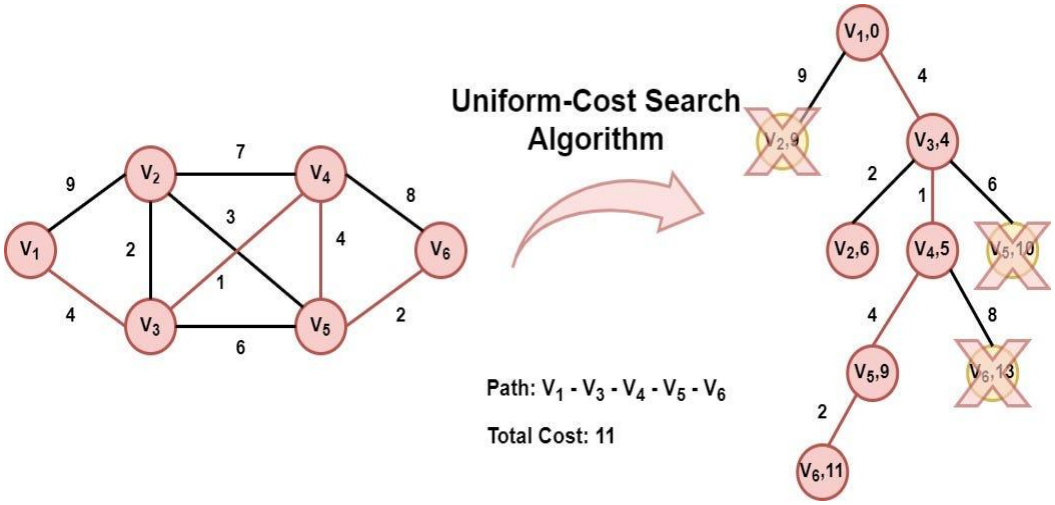
* If no path is found, return a message indicating that there is no solution.

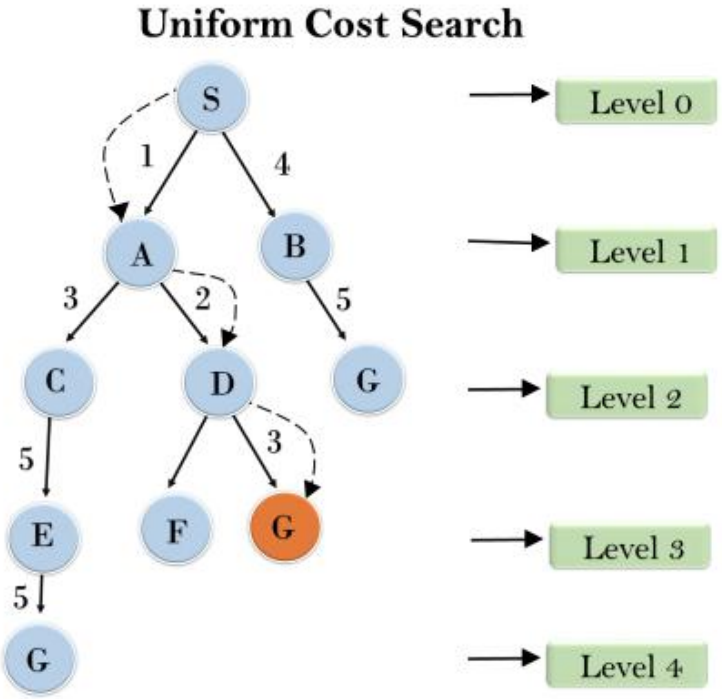
PTHON CODE :

In [3]:
```
import heapq

class Node:
    def __init__(self, state, cost, parent):
        self.state = state  # The current state (node) in the search space
        self.cost = cost    # The cost to reach this state
        self.parent = parent  # The parent node

    def __lt__(self, other):
        return self.cost < other.cost

def uniform_cost_search(graph, start, goal):
    frontier = []
    heapq.heappush(frontier, Node(start, 0, None))  # Priority queue
    explored = set()

    while frontier:
        node = heapq.heappop(frontier)

        if node.state == goal:
            path = []
            while node.parent:
                path.append(node.state)
                node = node.parent
            path.append(start)
            return path[::-1]

        explored.add(node.state)

        for neighbor, cost in graph[node.state]:
            if neighbor not in explored:
                heapq.heappush(frontier, Node(neighbor, node.cost + cost, node))

    return None

# Example graph representation as an adjacency list
graph = {
    'A': [('B', 4), ('C', 2)],
    'B': [('A', 4), ('C', 5), ('D', 10)],
    'C': [('A', 2), ('B', 5), ('D', 7)],
    'D': [('B', 10), ('C', 7)]
}
```

```
start_node = 'A'
goal_node = 'D'

path = uniform_cost_search(graph, start_node, goal_node)

if path:
    print("Path found:", ' -> '.join(path))
else:
    print("Path not found")
```

EXAMPLE :


V1
V2
V3
V4
V5
V6
9
7
3
8
2
4
1
4
6
2
Uniform-Cost Search
Algorithm
V1,0
9
4
V2,9
V3,4
2
1
6
V2,6
V4,5
V5,10
4
8
V5,9
V6,13
2
V6,11
Path: V1 - V3 - V4 - V5 - V6
Total Cost: 11


Uniform Cost Search
S
1
4
A
B
3
2
5
C
D
G
5
3
E
F
G
5
G
Level 0
Level 1
Level 2
Level 3
Level 4

BIDIRECTIONAL SEARCH

DEFINE :

Bidirectional search is a graph search algorithm that simultaneously explores the graph from both the start and the goal nodes in order to find the shortest path or determine if a path exists between them. It's commonly used in algorithms for solving problems like finding the shortest path in a graph or determining if a state is reachable in a state space search.

STEPS :

Step 1: Initialization:

* create two empty frontiers: one for the start node (let's call it start_frontier) and one for the goal node (let's call it goal_frontier).

* Add the start node to start_frontier and the goal node to goal_frontier.

Step 2: Search Expansion :

* Start a loop that continues until one of the termination conditions is met.

* In each iteration of the loop, alternate between expanding the start_frontier and the goal_frontier.

Step 3: Expand start_frontier:

* Take the first node from start_frontier (usually using a queue or a similar data structure).

* Generate all neighboring nodes of the current node that haven't been visited yet.

* Check for intersections: If any of the newly generated nodes are already in the goal_frontier, you have found a path from the start node to the goal node. Terminate the search.

Step 4: Expand goal_frontier :

* Similar to Step 3, take the first node from goal_frontier.

* Generate all neighboring nodes of the current node that haven't been visited yet.

* Check for intersections: If any of the newly generated nodes are already in the start_frontier, you have found a path from the goal node to the start node. Terminate the search.

Step 5: Termination Conditions :

Check for termination conditions:

* If a path is found connecting the start and goal nodes, you're done. You can reconstruct the path.
* If both start_frontier and goal_frontier are empty and no path is found, you can conclude that there is no path between the start and goal nodes.

Step 6: Path Reconstruction (if a path is found) :

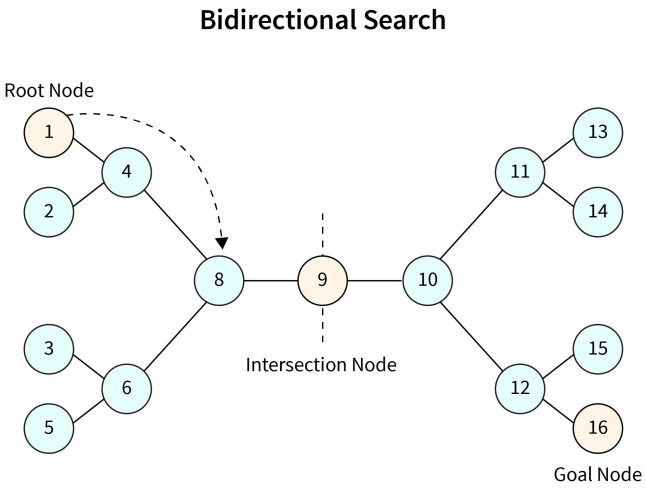
* f a path is found, you can reconstruct it by combining the paths from the start_frontier and goal_frontier. The point of intersection is typically the connecting node between the two paths.

* This combined path represents the shortest path from the start node to the goal node.

Step 7: End of Algorithm:

* The bidirectional search algorithm ends, and you have either found the shortest path or determined that no path exists between the start and goal nodes.

EXAMPLE :



SCALER Topics