AISC

Class: B.E COMP

Experiment 02: Solve a given problem using uninformed search technique

Learning Objective:

Basic Experiments

Solve a given problem using uninformed search technique.

Tools: Python

Theory:

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.

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.

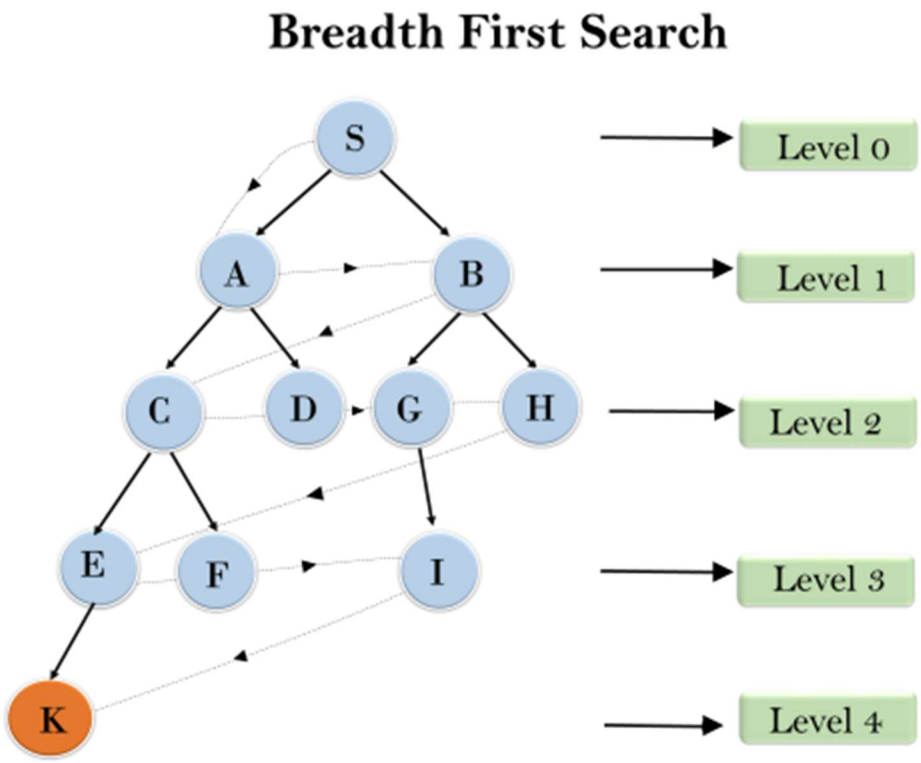
Disadvantages:

- 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.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1. S---> A--->B----->C--->D----->G--->H--->E----->F----->I----->K



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+b2+b3+.....+ bd= O (bd)

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is O(bd).

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.

2. Depth-first Search

- Depth-first search isa 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.

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:

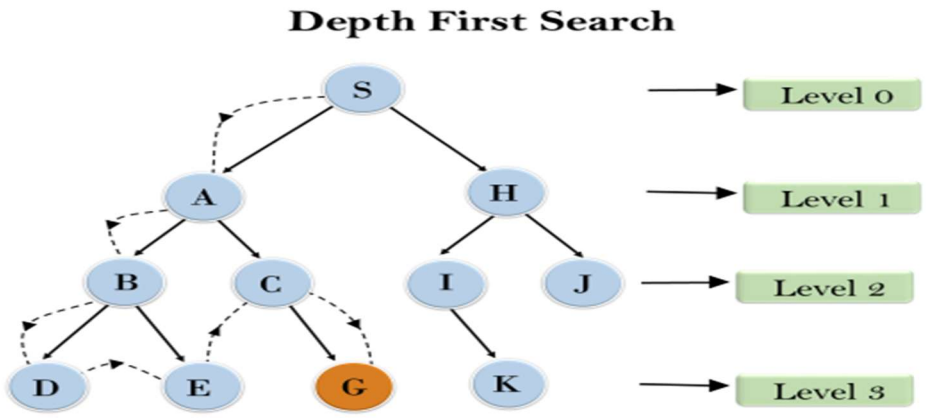
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



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:

T(n)= 1+ n2+ n3 +.....+ nm=O(nm)

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 **O(bm)**.

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.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search 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.

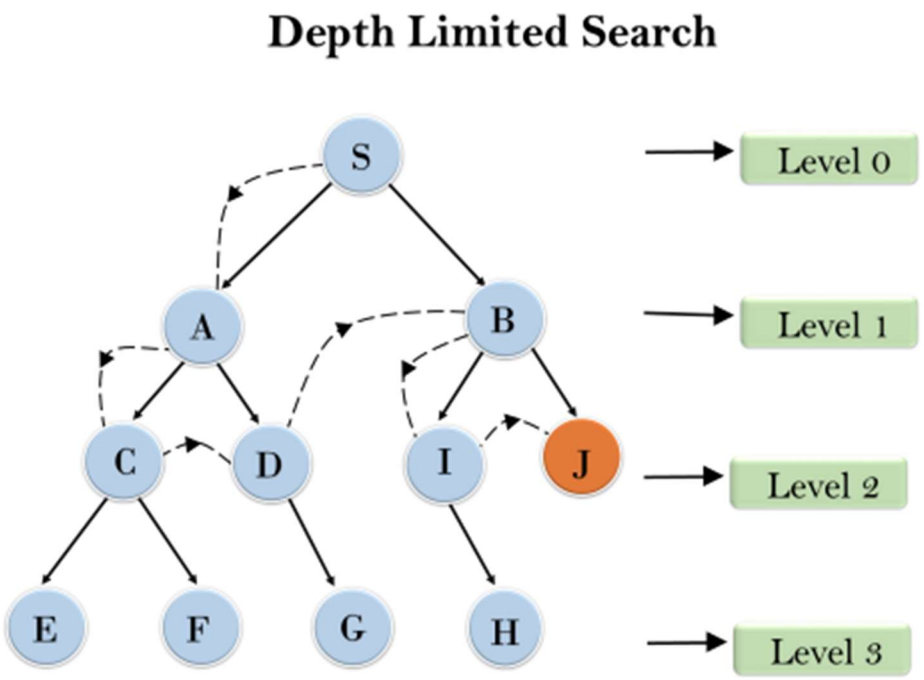
Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is **O(bℓ)**.

Space Complexity: Space complexity of DLS algorithm is O**(b×ℓ)**.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

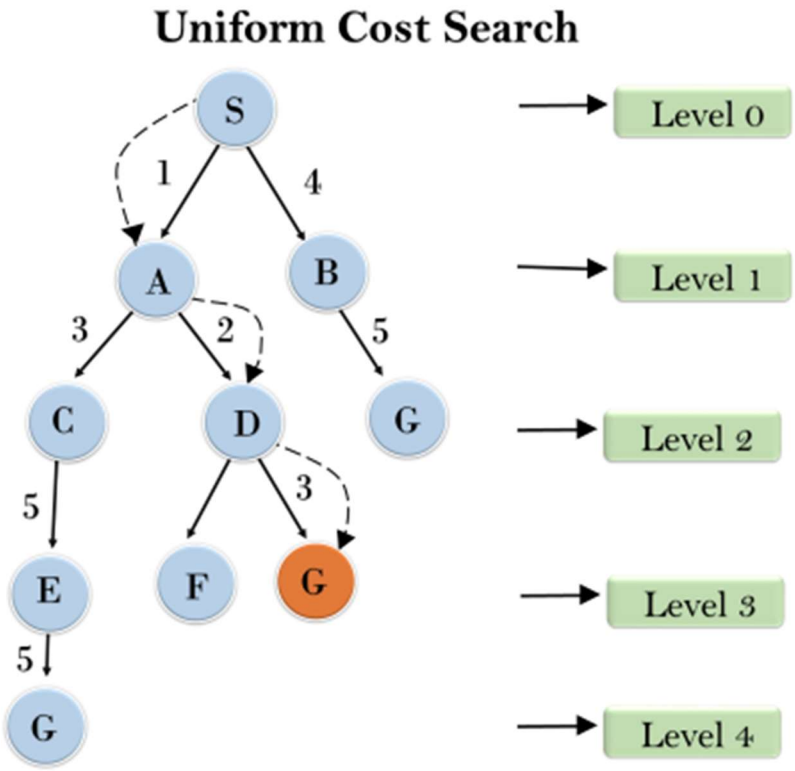
Advantages:

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

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is**O(b1 + [C*/ε])/**.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **O(b1 + [C*/ε])**.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepeningdepth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

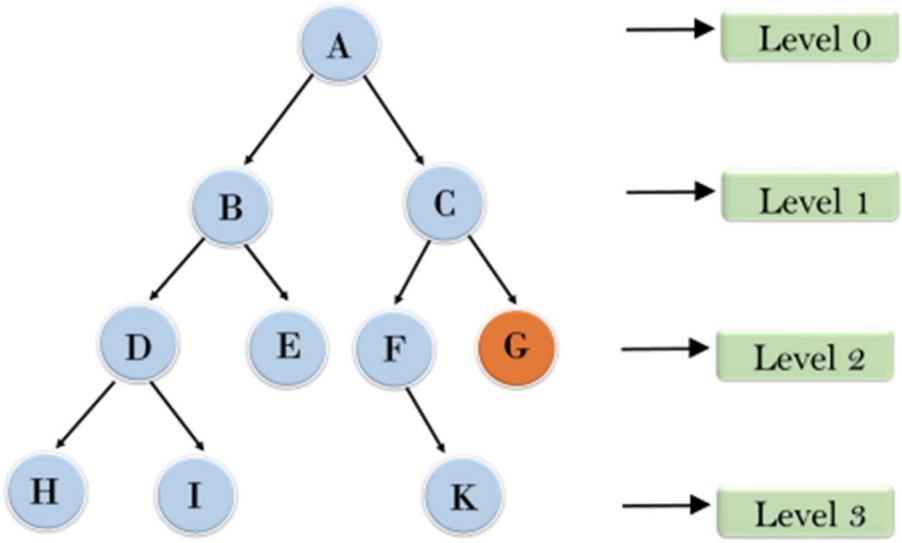
Disadvantages:

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

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

Iterative deepening depth first 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.

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **O(bd)**.

Space Complexity:

The space complexity of IDDFS will be **O(bd)**.

Optimal:

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

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node 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.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

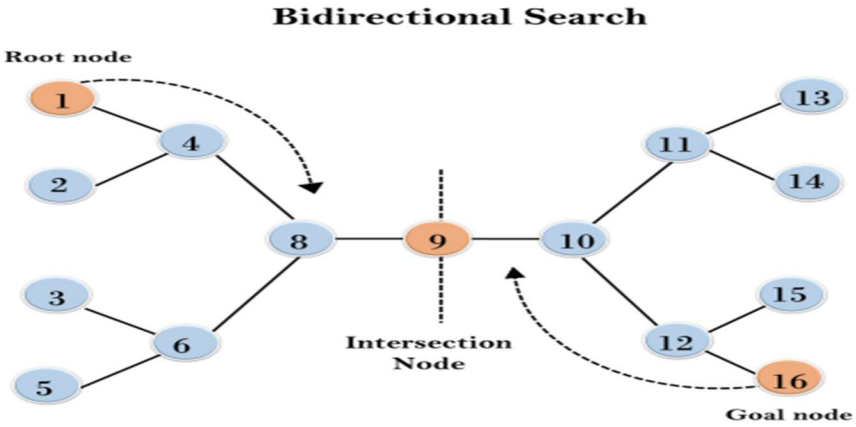
Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.



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

Time Complexity: Time complexity of bidirectional search using BFS is **O(bd)**.

Space Complexity: Space complexity of bidirectional search is **O(bd)**.

Optimal: Bidirectional search is Optimal.

Implementation:

Source Code:

#Iterative Deepening Depth First Search

```
print("Now implementing Iterative Deepening Depth First search")
from collections import defaultdict
class Graph:
    def __init__(self,vertices):
        self.V = vertices
        self.graph = defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def DLS(self,src,target,maxDepth):
        if src == target : return True
        if maxDepth <= 0 : return False
        for i in self.graph[src]:
            if(self.DLS(i,target,maxDepth-1)):
                return True
        return False
    def IDDFS(self,src, target, maxDepth):
        for i in range(maxDepth):
            if (self.DLS(src, target, i)):
                return True
        return False
g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)
target = int(input("Enter the Target Element to be found(0-6): "))
maxDepth = 3; src = 0
if g.IDDFS(src, target, maxDepth) == True:
    print ("Target is reachable from source " +
        "within max depth")
else :
    print ("Target is NOT reachable from source " +
        "within max depth")
```

#Uniform Cost Search

```
print("Now implementing Uniform Cost Search")
import queue as Q

def search(graph, start, end):
    if start not in graph:
        raise TypeError(str(start) + ' not found in graph !')
        return
    if end not in graph:
        raise TypeError(str(end) + ' not found in graph !')
        return
    queue = Q.PriorityQueue()
    queue.put((0, [start]))

    while not queue.empty():
        node = queue.get()
        current = node[1][len(node[1]) - 1]
        if end in node[1]:
            print("Path found: " + str(node[1]) + ", Cost = " + str(node[0]))
            break
        cost = node[0]
        for neighbor in graph[current]:
            temp = node[1][:]
            temp.append(neighbor)
            queue.put((cost + graph[current][neighbor], temp))
def readGraph():
    lines = int( input("Enter number of Cities: ") )
    graph = {}
    for line in range(lines):
        line = input()
        tokens = line.split()
```

```
        node = tokens[0]
        graph[node] = {}
        for i in range(1, len(tokens) - 1, 2):
            # print(node, tokens[i], tokens[i + 1])
            #    graph.addEdge(node,    tokens[i],
int(tokens[i + 1]))
            graph[node][tokens[i]] = int(tokens[i
+ 1])
    return graph
def main():
    graph = readGraph()
    search(graph, 'Arad', 'Bucharest')
if __name__ == "__main__":
    main()
```

OUTPUT:

#Now implementing Iterative Deepening Depth First search
Enter the Target Element to be found(0-6): 4
Target is reachable from source within max depth

#Now implementing Uniform Cost Search
Enter Number of Cities: 14
Arad Zerind 75 Timisoara 118 Sibiu 140
Zerind Oradea 71 Arad 75
Timisoara Arad 118 Lugoj 111
Sibiu Arad 140 Oradea 151 Fagaras 99 RimnicuVilcea 80
Oradea Zerind 71 Sibiu 151
Lugoj Timisoara 111 Mehadia 70
RimnicuVilcea Sibiu 80 Pitesti 97 Craiova 146
Mehadia Lugoj 70 Dobreta 75
Craiova Dobreta 120 RimnicuVilcea 146 Pitesti 138
Pitesti RimnicuVilcea 97 Craiova 138 Bucharest 101
Fagaras Sibiu 99 Bucharest 211
Dobreta Mehadia 75 Craiova 120
Bucharest Fagaras 211 Pitesti 101 Giurgiu 90
Giurgiu Bucharest 90
Path found: ['Arad', 'Sibiu', 'RimnicuVilcea', 'Pitesti', 'Bucharest'], Cost = 418

Learning Outcomes: Students should have the ability to
LO1: Understand the various Search Algorithms.

Course Outcomes: Upon completion of the course students will be able to understand problem formulation in ALSC.

Conclusion: Thus we have successfully understood various search algorithms and also implemented two of the above listed algorithms.

Viva Questions:

Ques.1 What do you understand by AI Agent?

Ques. 2. Explain the basic steps in PEAS?

Ques. 3. Write types of problem discussed in detail.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				