Assignment – 01

Course Code: Xai302

Course Name: Foundation Of

Ai &

R – Language

Submission: 04/09/2023

Done By: Jana venkatesan

BREADTH FIRST SEARCH

ALGORITHM:-

Breadth-first search (BFS) is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighbouring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node 1,2,3.

The algorithm is used to search a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level 2.

The steps involved in the BFS algorithm to explore a graph are given as follows 1:

- SET STATUS = 1 (ready state) for each node in G
- Enqueue the starting node A and set its STATUS = 2 (waiting state)
- Repeat Steps 4 and 5 until QUEUE is empty
- Dequeue a node N. Process it and set its STATUS = 3 (processed state).
- Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]
- EXIT

BFS can be used to find neighboring locations from a given source location, as well as in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages 1. It is also used to determine the

shortest path and minimum spanning tree, and in ford-Fulkerson method to compute the maximum flow in a flow network 1.

ITERATIVE DEEPENING SEARCH:-

Iterative deepening search (IDS) is a graph traversal algorithm that is used to find the shortest path between two nodes in a tree or graph data structure. It is a hybrid of depth-first search (DFS) and breadth-first search (BFS) algorithms 1,2.

The algorithm works by repeatedly performing DFS with a gradually increasing depth limit until the goal node is found. In each iteration, the DFS algorithm starts from the root node and explores all the nodes at the current depth level before moving on to the next level. The maximum depth limit is increased by one in each iteration until the goal node is found 2.

The steps involved in the IDS algorithm are given as follows 1:

- SET DEPTH_LIMIT = 0
- Repeat Steps 3 to 6 until the goal node is found
- SET DEPTH_LIMIT = DEPTH_LIMIT + 1
- Perform DFS with depth limit DEPTH_LIMIT
- If goal node is found, return it
- If goal node is not found, repeat from Step 3

An IDS has several advantages over other search algorithms such as BFS and DFS. It has a space complexity of O(bd), where b is the branching factor and d is the depth of the shallowest solution 2. This makes it more space-efficient than

BFS, which has a space complexity of O(bd). IDS also guarantees that it will find the shortest path between two nodes in a tree or graph data structure 1.

BI-DIRECTIONAL SEARCH:-

Bidirectional search is a graph traversal algorithm that finds the shortest path between two nodes in a directed graph. It runs two simultaneous searches: one forward from the initial state, and one backward from the goal state, stopping when the two meet 1. The algorithm is a variant of breadth-first search (BFS) and is used to reduce the time complexity of finding the shortest path between two nodes in a graph 2.

The bidirectional search algorithm works by starting two BFS searches simultaneously, one from the source node and one from the target node. The algorithm terminates when both searches meet at a common node, which is considered as the middle point of the path. The path from the source node to the middle point and from the target node to the middle point is then combined to form the shortest path between the two nodes 1.

The bidirectional search algorithm has several advantages over other search algorithms such as BFS and DFS. It reduces the time complexity of finding the shortest path between two nodes in a graph by exploring only half of the graph 2. It also reduces the space complexity of BFS by exploring only half of the graph 1. However, it requires more memory than BFS because it needs to store two sets of visited nodes 1.

DEPTH LIMITED SEARCH:-

Depth-limited search is a variation of the Depth-first search algorithm. It is an uninformed search algorithm that is used to traverse a tree or graph data structure. The algorithm is similar to DFS, but it has a predetermined depth limit 'l' 1. Nodes at depth l are considered to be nodes without any successors 1.

The algorithm starts at the root node and follows each branch to its deepest node until it reaches the depth limit 'l' 1. The algorithm then backtracks to the next deepest node that has unexplored branches and continues the process until it reaches the goal node or all nodes within the depth limit have been visited 1.

The Depth-limited search algorithm is useful when we want to avoid infinite loops that can occur in DFS 2. However, it can be less efficient than BFS because it may explore many nodes that are not on the optimal path to the goal node 3.

UNIFORM COST SEARCH:-

Uniform Cost Search is an algorithm used to traverse a weighted graph or tree data structure. It is an uninformed search algorithm that is used to find the path from the starting node to the goal node with the lowest cumulative cost 1. The algorithm is similar to Dijkstra's algorithm, but it does not use any heuristic information 1. Instead, it uses a priority queue to keep track of the nodes that have been visited and their associated costs 2. The algorithm starts at the root node and expands the node with the lowest cost first 2. It then adds all of its neighbors to the priority queue and continues this process until it reaches the goal node or all nodes have been visited 2.

In this example, we have a graph with four nodes: A, B, C, and D. Each node has a different cost of traversal. We want to find the path from node A to node D with the lowest cumulative cost. The uniform_cost_search function takes three arguments: graph, start, and goal. The graph argument is a dictionary that represents the graph. The keys of the dictionary are the nodes of the graph, and the values are dictionaries that represent the edges between nodes and their associated costs. The start argument is the starting node of the search, and the goal argument is the goal node of the search.

The function uses a priority queue to keep track of the nodes that have been visited and their associated costs. It starts at the starting node and expands the node with the lowest cost first. It then adds all of its neighbors to the priority queue and continues this process until it reaches the goal node or all nodes have been visited. Finally, it returns the cost of reaching the goal node.

In this example, Uniform Cost Search returns 6 as it is the minimum cost from A to D.

In [7]:
```
import heapq
```

In [8]:
```
def uniform_cost_search(graph, start, goal):
    visited = set()
    heap = [(0, start)]
    while heap:
        (cost, current) = heapq.heappop(heap)
        if current == goal:
            return cost
        if current in visited:
            continue
        visited.add(current)
        for neighbor in graph[current]:
            heapq.heappush(heap, (cost + graph[current][neighbor], neighbor))
    return -1
```

In [9]:
```
graph = {
    'A': {'B': 2, 'C': 3},
    'B': {'D': 5},
    'C': {'D': 1},
    'D': {}
}

print(uniform_cost_search(graph, 'A', 'D'))
```

```
4
```