Ex. No. 1 Implementation of uninformed search algorithm Breadth First Search - BFS

Date:

Aim:

To implement the uninformed search algorithm Breadth First Search (BFS) using python.

Algorithm:

Step 1 : Start.

Step 2 : Initialize a visited set to keep track of visited nodes.

Step 3 : Initialize a queue and add the starting node to the queue.

Step 4 : While the queue is not empty, do the following:

a) Remove the first node from the queue.

b) If the node has not been visited, mark it as visited and process it.

c) For each neighbour of the node that has not been visited, add it to the queue.

Step 5 : When all reachable nodes have been visited, terminate.

Step 6 : Stop

Program:

```
from collections import deque
def bfs(visited, graph, node):
  visited.append(node)
  queue = deque([node])
  while queue:
    m = queue.popleft()
```

Output:

```
Following is the breadth first search
5 3 7 2 4 8
```

```
    print(m, end=" ")
    for neighbor in graph[m]:
      if neighbor not in visited:
        visited.append(neighbor)
        queue.append(neighbor)
visited = []
graph = {'5': ['3', '7'], '3': ['2', '4'], '7': ['8'], '2': [],
'4': ['8'], '8': []}
print("Following is the breadth first search")
bfs(visited, graph, '5')
```

Result:

Thus the Breadth First Search (BFS) algorithm was written and executed successfully.

Ex. No. 2 Implementation of uninformed search algorithm Depth First Search - DFS

Date:

Aim:

To implement the uninformed search algorithm Depth First Search (DFS) using python.

Algorithm:

Step 1 : Start.

Step 2 : Create a set to keep track of visited nodes.

Step 3 : Choose a starting node and mark it as visited.

Step 4 : For each of the neighbours of the starting node, if it has not been visited, recursively call the DFS function on that neighbour.

Step 5 : Repeat step 3 for all the neighbours of the current node.

Step 6 : When all the neighbours of the current node have been visited, return to the node that led to the current node.

Step 7 : Repeat steps 3-5 until all the nodes have been visited.

Program:

```
graph = {'5': ['3', '7'], '3': ['2', '4'], '7': ['8'], '2': [],
'4': ['8'], '8': []}

visited = set()

def dfs(visited, graph, node):
  if node not in visited:
    print(node)
    visited.add(node)
    for neighbour in graph[node] :
      dfs(visited,graph,neighbour)

print("Following is the Depth First Search:")
```

Output:

```
Following is the Depth First Search:
5
3
2
4
8
7
```

```
dfs(visited, graph, '5')
```

Result:

Thus the Depth First Search (DFS) algorithm was written and executed successfully.

Ex. No. 3 Implementation of informed search algorithm A*

Date:

Aim:

To implement the uninformed search algorithm Depth First Search (DFS).

Algorithm:

Step 1 : Initialize the start node and the goal node.

Step 2 : Create an open set and add the start node to it.

Step 3 : Create a closed set to store visited nodes.

Step 4 : . Initialize a dictionary to keep track of the cost from the start node to each node.

Step 5 : Initialize a dictionary to keep track of the parent node for each node.

Step 6 : Set the cost of the start node to 0 and set its parent as itself.

Step 7 : While the open set is not empty, do the following:

a. Select the node with the lowest cost f(n) = g(n) + h(n) from the open set, where g(n) is the cost from the start node to node n, and h(n) is the heuristic estimate of the cost from node n to the goal node.
b. If the selected node is the goal node or there are no neighbors, terminate the algorithm.
c. Otherwise, for each neighbor m of the selected node, do the following:
    i. If m is not in the open or closed set, add it to the open set, set its cost g(m) as g(n) + cost(n, m), and set its parent as the selected node.
    ii. If m is already in the open set, update its cost and parent if the new cost is lower.
    iii. If m is already in the closed set, remove it from the closed set, update its cost and parent if the new cost is lower, and add it to the open set.

Step 8 : If the goal node is reached, construct the path by following the parent pointers from the goal node to the start node.

Step 9 : Return the path if it exists, otherwise return null or print "Path does not exist".

Program:

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] +
heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:

                        g[m] = g[n] + weight

                        parents[m] = n

                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None

        if n == stop_node:
```

```
            path = []
            while parents[n] ≠ n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            return path
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None

def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]

Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}

aStarAlgo('A', 'J')
```

Output:

```
Following is the breadth first search
['A', 'F', 'G', 'I', 'J']
```

Result:

Thus the A* algorithm was written and executed successfully.

Ex. No. 4 Implementation of informed search algorithm Memory Bounded Search

Date:

Aim:

To implement the informed search algorithm Memory Bounded Search using python.

Algorithm:

Step 1 : Initialize a priority queue called frontier with a tuple containing the . heuristic estimate of the start node and the start node itself.

Step 2 : Initialize an empty list called explored to keep track of nodes that . . have already been explored.

Step 3 : . Initialize a variable called memory_used to keep track of the number of nodes that have been added to the frontier or explored.

Step 4 : While the frontier is not empty, do the following:

a) If the value of memory_used exceeds the memory_limit, return "Memory limit exceeded".
b) Pop the node with the lowest estimated cost (i.e., the first item in the tuple) from the frontier, and add its node value to the explored list.
c) Increment the memory_used variable by 1. If the node value is equal to the goal node, return the explored list.
d) Generate the children of the current node using the graph's get_children method.
e) For each child node, do the following:
    i. If the child node has not already been explored, calculate its actual cost (g) as the weight of the edge from the current node to the child node, using the graph's get_edge_weight method.
    ii. Calculate the heuristic estimate (h) for the child node using the heuristic_fn dictionary.

iii. Calculate the total estimated cost (f) of reaching the child node as the sum of the actual cost (g) and the heuristic estimate (h).

iv. Add a tuple containing the estimated cost (f) and the child node value to the frontier.

Step 5 : . . If the goal node is not found and the memory limit is not exceeded, return "Goal state not found".

Program:

```
import heapq
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adjacency_list = {v: [] for v in vertices}

    def add_edge(self, u, v, weight):
        self.adjacency_list[u].append((v, weight))

    def get_children(self, vertex):
        return [edge[0] for edge in self.adjacency_list[vertex]]

    def get_edge_weight(self, u, V):
        for edge in self.adjacency_list[u]:
            if edge[0] == V:
                return edge[1]
        raise ValueError("No such edge exists")



def memory_bounded_heuristic_search(graph, start, goal,
heuristic_fn, memory_limit):
```

```
    frontier = [(heuristic_fn[start], start)]
    explored = []
    memory_used = 0
    while frontier:
        if memory_used > memory_limit:
            return "Memory limit exceeded"
        node = heapq.heappop(frontier)[1]
        explored.append(str(node))
        memory_used += 1
        if node == goal:
            return explored
        for child in graph.get_children(node):
            if child not in explored:
                g = graph.get_edge_weight(node, child)
                h = heuristic_fn[child]
                f = g + h
                heapq.heappush(frontier, (f, child))
    return "Goal state not found"



vertices = [1, 2, 3, 4, 5, 6]
graph = Graph(vertices)
graph.add_edge(1, 2, 1)
graph.add_edge(1, 3, 3)
graph.add_edge(2, 4, 5)
graph.add_edge(2, 5, 1)
graph.add_edge(3, 5, 2)
graph.add_edge(4, 6, 2)
graph.add_edge(5, 6, 4)
```

Output:

```
Goal state found: ['1', '2', '5', '6']
```

```
start = 1
goal = 6
memory_limit = 100

heuristic_fn = {1: 6, 2: 4, 3: 5, 4: 3, 5: 2, 6: 0}
result = memory_bounded_heuristic_search(graph, start,
goal, heuristic_fn, memory_limit)



if result == "Memory limit exceeded":
    print("Memory limit exceeded")
elif result == "Goal state not found":
    print("Goal state not found")
else:
    print("Goal state found:", result)
```

Result:

Thus the Memory Bounded Search algorithm was written and executed successfully.