Green University of Bangladesh
Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering
Semester: (Summer, Year:2024), B.Sc. in CSE (Day)

Lab Report NO 02
Course Title: Artificial Intelligence Lab
Course Code: CSE 316 **Section:** 212-D3

Lab Experiment Name: Implement BFS & DFS Traversal

Student Details

Name	ID
Sajid rahman rifan	212902017

Lab Date : 18/03/2024
Submission Date : 31/03/2024
Course Teacher's Name : Tasnim Tayiba Zannat

Lab Report Status

Marks: …… **Signature:.....**
Comments:..... **Date:.....**

1. TITLE OF THE LAB EXPERIMENT

Implement BFS & DFS Traversal

2. OBJECTIVES/AIM

1. Implement a Breadth-First Search (BFS) traversal algorithm to efficiently explore and discover all nodes within a graph, ensuring each level of the graph is visited before proceeding to deeper levels.
2. Develop a Breadth-First Search (BFS) traversal method capable of systematically navigating through a graph, prioritising breadth over depth to efficiently locate target nodes and identify shortest paths.
3. The program should implement the DFS algorithm to traverse the graph and compute the topological order. The DFS algorithm explores as far as possible along each branch of the graph before backtracking.

3.PROCEDURE / ANALYSIS / DESIGN

Algorithm:

- Define a Node class to represent each cell in the grid with attributes x and y coordinates, level, and parent.
- implement a BFS class with necessary attributes and methods.
- Initialise the grid graph, source node, and goal node.
- Perform breadth-first search traversal starting from the source node.
- Enqueue the source node into a deque and iterate until the deque is empty.
- For each node dequeued, explore its neighbouring cells in all four directions.
- If a valid neighbouring cell is found, update its level, mark it as visited, and enqueue it.
- If the goal cell is found, mark the goal as reached and store the level required to reach it.
- Once the goal is found, backtrack through parent nodes to print the path from the source to the goal.
- It first calculates the in-degree of each node by iterating over all nodes and their neighbours, similar to the previous implementation.
- A queue is created, and the starting node is added to it.
- For each removed node, it decreases the in-degree of its neighbours. If the in-degree of a neighbour becomes 0, it is added to the queue.
- The algorithm then iterates over the queue, removing a node from the front, and adding it to the topological_order list.

4. IMPLEMENTATION

Code:

```python
from collections import deque

class Node:
    def __init__(self, x, y, level):
        self.x = x
        self.y = y
        self.level = level
        self.parent = None

class BFS:
    def __init__(self):
        self.directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        self.found = False
        self.goal_level = 0
        self.N = 0
        self.source = None
        self.goal = None

    def init(self):
        graph = [
            [0, 0, 1, 0, 1],
            [0, 1, 1, 1, 1],
            [0, 1, 0, 0, 1],
            [1, 1, 0, 1, 1],
            [1, 0, 0, 0, 1]
        ]
        self.N = len(graph)

        source_x, source_y = 0, 2
        goal_x, goal_y = 4, 4
        self.source = Node(source_x, source_y, 0)
        self.goal = Node(goal_x, goal_y, float('inf'))
        self.st_bfs(graph)

        if self.found:
            print("Goal found")
            print("Number of moves required =", self.goal_level)
            self.print_path(self.goal)
        else:
            print("Goal cannot be reached from the starting block")

    def st_bfs(self, graph):
        queue = deque()
        queue.append(self.source)

        while queue:
            u = queue.popleft()

            for dx, dy in self.directions:
                v_x, v_y = u.x + dx, u.y + dy

                if 0 <= v_x < self.N and 0 <= v_y < self.N and graph[v_x][v_y] == 1:
                    v_level = u.level + 1
                    if v_x == self.goal.x and v_y == self.goal.y:
                        self.found = True
                        self.goal_level = v_level
                        self.goal.parent = u
                        return
                    graph[v_x][v_y] = 0
                    child = Node(v_x, v_y, v_level)

    def print_path(self, node):
        if node.parent:
            self.print_path(node.parent)
        print(f"({node.x}, {node.y})")

if __name__ == "__main__":
    bfs = BFS()
    bfs.init()
```

Code:

```python
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices  # Total number of vertices in the graph

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

    def topological_sort(self):
        in_degree = [0] * self.V  # To store the in-degree of each vertex

        # Calculate the in-degree of each vertex
        for i in self.graph:
            for j in self.graph[i]:
                in_degree[j] += 1

        queue = []
        for i in range(self.V):
            if in_degree[i] == 0:
                queue.append(i)

        top_order = []
        count = 0  # Counter to track the topological order
        while queue:
            u = queue.pop(0)
            top_order.insert(count, u)
            count += 1

            # Update the in-degree of the adjacent vertices
            for neighbour in self.graph[u]:
                in_degree[neighbour] -= 1
                if in_degree[neighbour] == 0:
                    queue.append(neighbour)

        # Check if there is a cycle in the graph
        if count != self.V:
            print("Cycle exists in the graph")
        else:
            print("Topological Order:", top_order)

# Create a graph
g = Graph(6)
g.add_edge(5, 2)
g.add_edge(5, 0)
g.add_edge(4, 0)
g.add_edge(4, 1)
g.add_edge(2, 3)
g.add_edge(3, 1)

# Find the topological order
g.topological_sort()
```

5. TEST RESULT / OUTPUT

Output:

Fig 01: BFS

```
# Create a graph
g = Graph(6)
g.add_edge(5, 2)
g.add_edge(5, 0)
g.add_edge(4, 0)
g.add_edge(4, 1)
g.add_edge(2, 3)
g.add_edge(3, 1)

# Find the topological order
g.topological_sort()
```

```
Topological Order: [4, 5, 2, 0, 3, 1]
```

Fig 02: DFS

6. ANALYSIS AND DISCUSSION

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that systematically explores all the vertices of a graph level by level. It starts at a chosen vertex and explores all its neighbours before moving to the next level. Implementing BFS involves using a queue data structure to keep track of vertices to visit next. This algorithm is particularly useful for finding the shortest path between two vertices in an unweighted graph and for traversing trees.Depth-First Search (DFS),Find the topological order of node traversal for a robot on a 2D graph plane, we can use a similar approach as the previous topological sorting algorithm. However, we need to consider the additional constraints and requirements for the robot's movement on a 2D plane.