Akshat Agarwal
RA1911003010646

Artificial Intelligence
Lab - 4

Aim : Implementation and Analysis of DFS and BFS for an application

DETECTING CYCLE IN DIRECTED GRAPHS USING DEPTH-FIRST-SEARCH (DFS)

Algorithm :

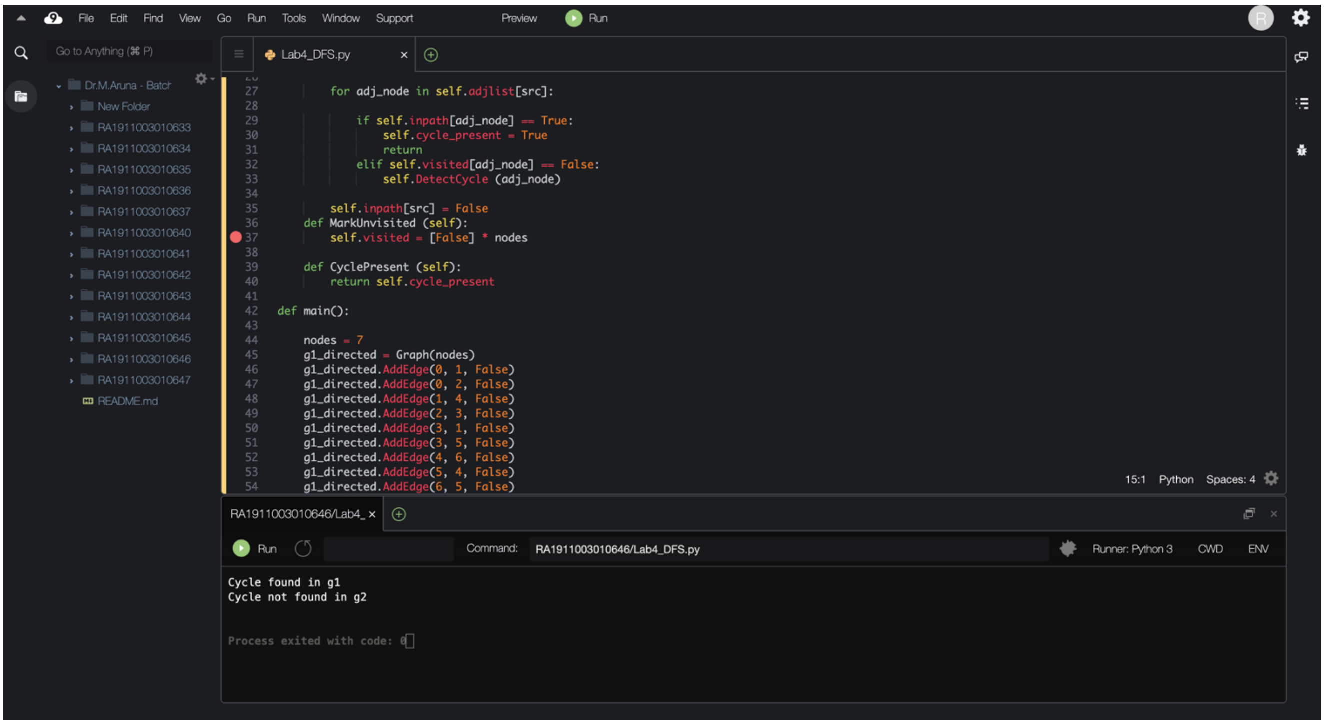
1. Mark the source_node as visited.
2. Mark the source_node as in_path node.
3. **For** all the adjacent nodes to the source_node do
4. **If** the adjacent node has been marked as in_path node, **then**
5. Cycle found. Return.
6. **If** the adjacent node has not been visited, **then**
7. **Detect_Cycle** (adjacent_node)
8. Now that we are backtracking unmark the source_node in in_path as it might be revisited.

Code :

```
from collections import defaultdict

class Graph:

    def __init__(self, nodes : int):

        self.adjlist = defaultdict(list)
        self.nodes = nodes
        self.visited = [False] * nodes

        self.inpath = [False] * nodes
        self.cycle_present = False

    def AddEdge (self, src : int, dst : int, bidirectional : bool):

        self.adjlist[src].append(dst)

        if bidirectional:
            self.adjlist[dst].append(src)

    def DetectCycle (self, src : int):

        self.visited[src] = True

        self.inpath[src] = True

        for adj_node in self.adjlist[src]:

            if self.inpath[adj_node] == True:
                self.cycle_present = True
                return
            elif self.visited[adj_node] == False:
                self.DetectCycle (adj_node)

        self.inpath[src] = False
    def MarkUnvisited (self):
        self.visited = [False] * nodes

    def CyclePresent (self):
        return self.cycle_present

def main():

    nodes = 7
    g1_directed = Graph(nodes)
    g1_directed.AddEdge(0, 1, False)
    g1_directed.AddEdge(0, 2, False)
    g1_directed.AddEdge(1, 4, False)
    g1_directed.AddEdge(2, 3, False)
    g1_directed.AddEdge(3, 1, False)
```

```
    g1_directed.AddEdge(3, 5, False)
    g1_directed.AddEdge(4, 6, False)
    g1_directed.AddEdge(5, 4, False)
    g1_directed.AddEdge(6, 5, False)

    g1_directed.DetectCycle(0)

    if g1_directed.CyclePresent() == True:
       print("Cycle found in g1")
    else:
       print("Cycle not found g1")


    nodes = 5
    g2_directed = Graph(nodes)

    g2_directed.AddEdge(0, 1, False)
    g2_directed.AddEdge(0, 2, False)
    g2_directed.AddEdge(2, 3, False)
    g2_directed.AddEdge(3, 4, False)
    g2_directed.AddEdge(4, 1, False)

    g2_directed.DetectCycle(0)

    if g2_directed.CyclePresent() == True:
        print("Cycle found in g2")
    else:
        print("Cycle not found in g2")

if __name__ == "__main__":
    main()
```

Output :



SHORTEST PATH IN A BINARY MAZE

Algorithm :

1. We start from the source cell and calls BFS procedure.
2. We maintain a queue to store the coordinates of the matrix and initialize it with the source cell.
3. We also maintain a Boolean array visited of same size as our input matrix and initialize all its elements to false.
    1. We LOOP till queue is not empty
    2. Dequeue front cell from the queue
    3. Return if the destination coordinates have reached.
    4. For each of its four adjacent cells, if the value is 1 and they are not visited yet, we enqueue it in the queue and also mark them as visited.

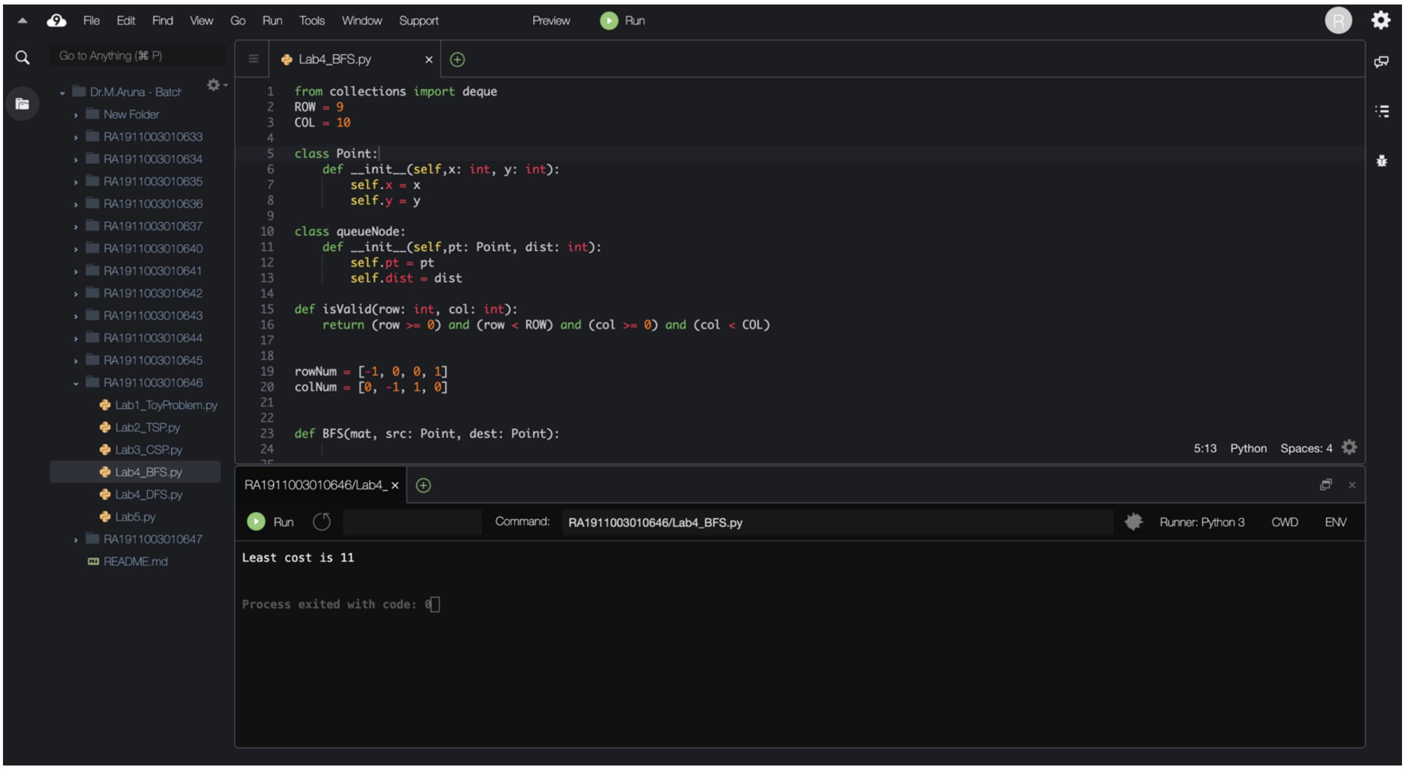
Code :

```
from collections import deque
ROW = 9
COL = 10

class Point:
    def __init__(self,x: int, y: int):
        self.x = x
        self.y = y

class queueNode:
    def __init__(self,pt: Point, dist: int):
        self.pt = pt
        self.dist = dist

def isValid(row: int, col: int):
    return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)


rowNum = [-1, 0, 0, 1]
colNum = [0, -1, 1, 0]
```

```
def BFS(mat, src: Point, dest: Point):


    if mat[src.x][src.y]!=1 or mat[dest.x][dest.y]!=1:
        return -1

    visited = [[False for i in range(COL)]
                       for j in range(ROW)]


    visited[src.x][src.y] = True


    q = deque()

    s = queueNode(src,0)
    q.append(s)


    while q:

        curr = q.popleft()

        pt = curr.pt
        if pt.x == dest.x and pt.y == dest.y:
            return curr.dist

        for i in range(4):
            row = pt.x + rowNum[i]
            col = pt.y + colNum[i]


            if (isValid(row,col) and
               mat[row][col] == 1 and
                not visited[row][col]):
                visited[row][col] = True
                Adjcell = queueNode(Point(row,col),
                                    curr.dist+1)
                q.append(Adjcell)

    return -1

# Driver code
def main():
    mat = [[ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 ],
           [ 1, 0, 1, 0, 1, 1, 1, 0, 1, 1 ],
           [ 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 ],
           [ 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 ],
           [ 1, 1, 1, 0, 1, 1, 1, 0, 1, 0 ],
           [ 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 ],
           [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 ],
           [ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 ],
           [ 1, 1, 0, 0, 0, 0, 1, 0, 0, 1 ]]
```

```
    source = Point(0,0)
    dest = Point(3,4)

    dist = BFS(mat,source,dest)

    if dist!=-1:
        print("Least cost is",dist)
    else:
        print("No path")
main()
```

Output :



RA1911003010646
Akshat Agarwal

Date ___/___/___
Page ______
STUDY BUDDIES

Artificial Intelligence
Lab-4

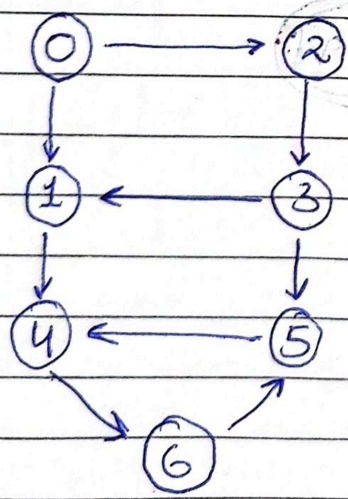
Aim: Implementation of DFS & BFS

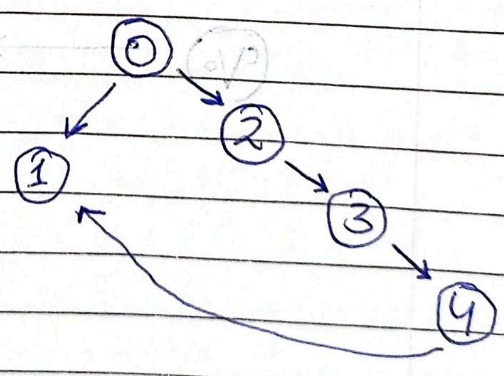
DFS- Whether the graph has cycle or not

Problem Formulation: Given a directed graph with edge relationship, we have to find if there is any cycle in the graph. Which means that if any path can be found such that the same node repeats again while following the edges.

Problem Solving:

While doing a depth-first search traversal, we keep track of the nodes visited in the current traversal path in addition to the list of all the visited nodes. During the traversal of the current path, if we come to a node that was already marked visited then we have found a cycle.





Traversal: 0 1 2 3 4

Traversal: 0 1 4 6 5 2 3

No cycle detected

Cycle detected

cello

BFS - Maze Problem

Problem Formulation:

We will be given a maze with starting and end point specified. There will be only one correct path to reach the end point. We have to find the correct path.

Problem Solving:

The maze will be taken in the form of $m \times m$ matrix with only two values filling them. 1 for a movable spot and 0 for the unreachable spots, i.e, the walls of the maze.

We will start from the default left top corner & traverse towards the destination point. As we reach a dead end we will revert back & traverse in a different path