MAHARISHI DAYANAND UNIVERSITY

Delhi Global Institute of Technology

Artificial Intelligence Lab

Submitted By: Bazgha Razi

Subject Code : LC-CSE-326G

Subject Name: Artificial Intelligence Lab using Python

Registration Number :*191380214*

INDEX

Program 1: Write a program to implement Breadth First Search using python.

Code:

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def BFS(self, s):
        visited = [False] * (len(self.graph))
        queue = []
        queue.append(s)
        visited[s] = True
        while queue:
            s = queue.pop(0)
            print (s, end = " ")
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print ("Following is Breadth First Traversal"" (starting from vertex 1)")
g.BFS(1)
```

Output:

```
Following is Breadth First Traversal (starting from vertex 1)
1 2 0 3
```

Program 2: Write a program to implement Depth First Search using python.

Code:

```
def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = set()
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output:

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

Program 3: Write a program to implement Tic-Tac-Toe using python.

Code:

```
import numpy as np
import random
from time import sleep
def create_board():
	return(np.array([
					[0, 0, 0],
					[0, 0, 0],
					[0, 0, 0]
				]))
def possibilities(board):
	l = []
	for i in range(len(board)):
		for j in range(len(board)):
			if board[i][j] == 0:
				l.append((i, j))
	return(l)
def random_place(board, player):
	selection = possibilities(board)
	current_loc = random.choice(selection)
	board[current_loc] = player
	return(board)
def row_win(board, player):
	for x in range(len(board)):
		win = True
		for y in range(len(board)):
			if board[x, y] != player:
				win = False
				continue
		if win == True:
			return(win)
	return(win)
def col_win(board, player):
	for x in range(len(board)):
		win = True
		for y in range(len(board)):
			if board[y][x] != player:
				win = False
				continue
		if win == True:
			return(win)
```

```
    return(win)
def diag_win(board, player):
    win = True
    y = 0
    for x in range(len(board)):
        if board[x, x] != player:
            win = False
    if win:
        return win
    win = True
    if win:
        for x in range(len(board)):
            y = len(board) - 1 - x
            if board[x, y] != player:
                win = False
    return win
def evaluate(board):
    winner = 0
    for player in [1, 2]:
        if (row_win(board, player) or
            col_win(board,player) or
            diag_win(board,player)):
            winner = player
    if np.all(board != 0) and winner == 0:
        winner = -1
    return winner
def play_game():
    board, winner, counter = create_board(), 0, 1
    print(board)
    sleep(2)
    while winner == 0:
        for player in [1, 2]:
            board = random_place(board, player)
            print("Board after " + str(counter) + " move")
            print(board)
            sleep(2)
            counter += 1
            winner = evaluate(board)
            if winner != 0:
                break
    return(winner)
print("Winner is: " + str(play_game()))
```

Output:

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[0 0 0]
 [0 0 0]
 [0 0 1]]
Board after 2 move
[[0 0 0]
 [2 0 0]
 [0 0 1]]
Board after 3 move
[[0 1 0]
 [2 0 0]
 [0 0 1]]
Board after 4 move
[[0 1 0]
 [2 0 0]
 [2 0 1]]
Board after 5 move
[[0 1 1]
 [2 0 0]
 [2 0 1]]
Board after 6 move
[[0 1 1]
 [2 0 0]
 [2 2 1]]
Board after 7 move
[[0 1 1]
 [2 0 1]
 [2 2 1]]
Winner is: 1
```

Program 4: Write a program to implement 8-Puzzle problem using python.

Code:

```
import copy
from heapq import heappush, heappop
n = 3
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]
class priorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, k):
        heappush(self.heap, k)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

class node:

    def __init__(self, parent, mat, empty_tile_pos,
                 cost, level):
        self.parent = parent
        self.mat = mat
        self.empty_tile_pos = empty_tile_pos
        self.cost = cost
        self.level = level
    def __lt__(self, nxt):
        return self.cost < nxt.cost
def calculateCost(mat, final) -> int:

    count = 0
    for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                (mat[i][j] != final[i][j])):
                count += 1

    return count
```

```
def newNode(mat, empty_tile_pos, new_empty_tile_pos,
                        level, parent, final) -> node:
        new_mat = copy.deepcopy(mat)
        x1 = empty_tile_pos[0]
        y1 = empty_tile_pos[1]
        x2 = new_empty_tile_pos[0]
        y2 = new_empty_tile_pos[1]
        new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2],
new_mat[x1][y1]
        cost = calculateCost(new_mat, final)
        new_node = node(parent, new_mat, new_empty_tile_pos,
                                        cost, level)
        return new_node
def printMatrix(mat):
        for i in range(n):
                for j in range(n):
                        print("%d " % (mat[i][j]), end = " ")
                print()
def isSafe(x, y):
        return x >= 0 and x < n and y >= 0 and y < n
def printPath(root):
        if root == None:
                return
        printPath(root.parent)
        printMatrix(root.mat)
        print()
def solve(initial, empty_tile_pos, final):
        pq = priorityQueue()
        cost = calculateCost(initial, final)
        root = node(None, initial,
                                empty_tile_pos, cost, 0)
        pq.push(root)
        while not pq.empty():
                minimum = pq.pop()
                if minimum.cost == 0:
                        printPath(minimum)
                        return
                for i in range(n):
                        new_tile_pos = [
                                minimum.empty_tile_pos[0] + row[i],
                                minimum.empty_tile_pos[1] + col[i], ]
                        if isSafe(new_tile_pos[0], new_tile_pos[1]):
                                child = newNode(minimum.mat,
```

```
                                        minimum.empty_tile_pos,
                                        new_tile_pos,
                                        minimum.level + 1,
                                        minimum, final,)
                        pq.push(child)
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],
          [ 5, 8, 6 ],
          [ 0, 7, 4 ] ]

empty_tile_pos = [ 1, 2 ]

solve(initial, empty_tile_pos, final)
```

Output:

```
1   2   3
5   6   0
7   8   4

1   2   3
5   0   6
7   8   4

1   2   3
5   8   6
7   0   4

1   2   3
5   8   6
0   7   4
```