**EXP NO. 1 IMPLEMENTATION OF TOY PROBLEM – TICK-TAC-TOE**

AIM:

ALGORITHM:

**PROGRAM:**

import random

class TicTacToe:

def \_\_init\_\_(self):

self.board = []

def create\_board(self):

for i in range(3):

row = []

for j in range(3):

row.append('-')

self.board.append(row)

def get\_random\_first\_player(self):

return random.randint(0, 1)

def fix\_spot(self, row, col, player):

self.board[row][col] = player

def is\_player\_win(self, player):

win = None

n = len(self.board)

for i in range(n):

win = True

for j in range(n):

if self.board[i][j] != player:

win = False

break

if win:

return win

for i in range(n):

win = True

for j in range(n):

if self.board[j][i] != player:

win = False

break

if win: return win

win = True

for i in range(n):

if self.board[i][i] != player:

win = False

break

if win:

return win

win = True

for i in range(n):

if self.board[i][n - 1 - i] != player:

win = False

break

if win:

return win

return False

for row in self.board: for item in row:

if item == '-':

return False return True

def is\_board\_filled(self):

for row in self.board:

for item in row:

if item == '-':

return False

return True

def swap\_player\_turn(self, player):

return 'X' if player == 'O' else 'O'

def show\_board(self):

for row in self.board:

for item in row:

print(item, end=" ")

print()

def start(self):

self.create\_board()

player = 'X' if self.get\_random\_first\_player() == 1 else 'O'

while True:

print(f"Player {player} turn")

self.show\_board()

row, col = list(map(int, input("Enter row and column numbers to fix spot: ").split()))

print()

self.fix\_spot(row - 1, col - 1, player)t

if self.is\_player\_win(player):

print(f"Player {player} wins the game!") break

if self.is\_board\_filled():

print("Match Draw!")break

player = self.swap\_player\_turn(player) print()

self.show\_board()

tic\_tac\_toe = TicTacToe()

tic\_tac\_toe.start()

**EXP NO. 2 DEVELOPING AGENT PROGRAMS FOR REAL WORLD PROBLEM**

AIM:

ALGORITHM:

**PROGRAM:**

import copy

from heapq import heappush, heappop

n = 3

rows = [ 1, 0, -1, 0 ]

cols = [ 0, -1, 0, 1 ]

class priorityQueue: def \_init\_(self): self.heap = []

def push(self, key):

heappush(self.heap, key)

def pop(self):

return heappop(self.heap)

def empty(self):

if not self.heap: return True else:

return False

class nodes:

def \_init\_(self, parent, mats, empty\_tile\_posi, costs, levels):

self.parent = parent

self.mats = mats

self.empty\_tile\_posi = empty\_tile\_posi

self.costs = costs

self.levels = levels

def \_lt\_(self, nxt):

return self.costs < nxt.costs

def calculateCosts(mats, final) -> int: count = 0

for i in range(n):

for j in range(n):

if ((mats[i][j]) and

(mats[i][j] != final[i][j])):

count += 1 return count

def newNodes(mats, empty\_tile\_posi, new\_empty\_tile\_posi,

levels, parent, final) -> nodes:

new\_mats = copy.deepcopy(mats)

x1 = empty\_tile\_posi[0]

y1 = empty\_tile\_posi[1]

x2 = new\_empty\_tile\_posi[0]

y2 = new\_empty\_tile\_posi[1]

new\_mats[x1][y1], new\_mats[x2][y2] = new\_mats[x2][y2], new\_mats[x1][y1]

costs = calculateCosts(new\_mats, final)new\_nodes = nodes(parent, new\_mats, new\_empty\_tile\_posi,

costs, levels) return new\_nodes

def printMatsrix(mats):

for i in range(n):

for j in range(n):

print("%d " % (mats[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)

printMatsrix(root.mats)

print()

def solve(initial, empty\_tile\_posi, final):

pq = priorityQueue()

costs = calculateCosts(initial, final)

root = nodes(None, initial, empty\_tile\_posi, costs, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop()

if minimum.costs == 0:

printPath(minimum)

return

for i in range(n):

new\_tile\_posi = [

minimum.empty\_tile\_posi[0] + rows[i],

minimum.empty\_tile\_posi[1] + cols[i], ]

if isSafe(new\_tile\_posi[0], new\_tile\_posi[1]):

child = newNodes(minimum.mats,

minimum.empty\_tile\_posi,

new\_tile\_posi,

minimum.levels + 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\_posi = [ 1, 2 ]

solve(initial, empty\_tile\_posi, final)

**EXP NO. 3 IMPLEMENTATION OF CONSTRAINT SATISFACTION PROBLEM**

AIM:

ALGORITHM:

**PROGRAM:**

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

print (board[i][j],end=' ')

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, N, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1) == True:

return True

board[i][col] = 0

return False

def solveNQ():

board = [ [0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]

]

if solveNQUtil(board, 0) == False:

print ("Solution does not exist")

return False

printSolution(board)

return True

solveNQ()

**EXP NO. 4 IMPLEMENTATION AND ANALYSIS OF DFS AND BFS FOR**

**APPLICATION**

AIM:

ALGORITHM:

**PROGRAM:**

from collections import defaultdict

class Graph:

def \_\_init\_\_(self):

self.graph = defaultdict(list)

def addEdge(self,u,v):

self.graph[u].append(v)

def DFSUtil(self, v, visited):

visited[v]= True

print v,

for i in self.graph[v]:

if visited[i] == False:

self.DFSUtil(i, visited)

def DFS(self):V = len(self.graph)

visited =[False]\*(V)

for i in range(V):

if visited[i] == False:

self.DFSUtil(i, visited)

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 Depth First Traversal"

g.DFS()

**EXP NO. 5 DEVELOPING THE BEST FIRST SEARCH AND A\* ALGORITHM**

**FOR REAL WORLD**

AIM:

ALGORITHM:

**PROGRAM:**

graph = {

'A': {'B': 10, 'C': 20},

'B': {'A': 10, 'D': 5, 'E': 15},

'C': {'A': 20, 'F': 30},

'D': {'B': 5},

'E': {'B': 15, 'F': 5},

'F': {'C': 30, 'E': 5}

}

# Define the heuristic function for A\* algorithm

def heuristic(a, b):

return abs(ord(a) - ord(b))

# Define the BFS function

def bfs(graph, start, end):

queue = [(start, [start], 0)]

while queue:

(node, path, cost) = queue.pop(0)

for next\_node in graph[node]:

if next\_node == end:

return path + [next\_node], cost + graph[node][next\_node]

else:

queue.append((next\_node, path + [next\_node], cost + graph[node][next\_node]))

# Define the A\* function

def a\_star(graph, start, end):

queue = [(0, start, [start], 0)]

visited = set()

while queue:

(f\_cost, node, path, cost) = queue.pop(0)

if node in visited:

continue

visited.add(node)

if node == end:

return path, cost

for next\_node in graph[node]:

g\_cost = cost + graph[node][next\_node]

h\_cost = heuristic(next\_node, end)

queue.append((g\_cost + h\_cost, next\_node, path + [next\_node], g\_cost))

queue = sorted(queue, key=lambda x: x[0])

# Test the algorithms

print("BFS: ", bfs(graph, 'A', 'F'))

print("A\*: ", a\_star(graph, 'A', 'F'))