LAB – 01:

1. Implement Tic –Tac –Toe Game.

```
board={1:' ',2:' ',3:' ',
      4:' ',5:' ',6:' ',
      7:' ',8:' ',9:' '

}

def printBoard(board):
   print(board[1]+'|'+board[2]+'|'+board[3])
   print('-+-+-')
   print(board[4] + '|' + board[5] + '|' + board[6])
   print('-+-+-')
   print(board[7] + '|' + board[8] + '|' + board[9])
   print('\n')

def spaceFree(pos):
   if(board[pos]==' '):
      return True
   else:
      return False

def checkWin():
   if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
      return True
   elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
      return True
   elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
      return True
   elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
      return True
   elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
      return True
   elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
      return True
   elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
      return True
   elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
      return True
   else:
      return False

def checkMoveForWin(move):
   if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
      return True
   elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
      return True
   elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
      return True
   elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
      return True
   elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
```

```
        return True
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if (board[key]==' '):
            return False
    return True


def insertLetter(letter, position):
    if (spaceFree(position)):
        board[position] = letter
        printBoard(board)

        if (checkDraw()):
            print('Draw!')
        elif (checkWin()):
            if (letter == 'X'):
                print('Bot wins!')
            else:
                print('You win!')
        return

    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        insertLetter(letter, position)
        return

player = 'O'
bot ='X'

def playerMove():
    position=int(input('Enter position for O:'))
    insertLetter(player, position)
    return

def compMove():
    bestScore=-1000
    bestMove=0
    for key in board.keys():
        if (board[key]==' '):
            board[key]=bot
            score = minimax(board, False)
            board[key] = ' '
            if (score > bestScore):
                bestScore = score
```

```
                bestMove = key

    insertLetter(bot, bestMove)
    return
def minimax(board, isMaximizing):
    if (checkMoveForWin(bot)):
        return 1
    elif (checkMoveForWin(player)):
        return -1
    elif (checkDraw()):
        return 0
    if isMaximizing:
        bestScore = -1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                if (score > bestScore):
                    bestScore = score
        return bestScore
    else:
        bestScore = 1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                if (score < bestScore):
                    bestScore = score
        return bestScore

while not checkWin():
    compMove()
    playerMove()
```

Output:

```
X| |
-+-+-
 | |
-+-+-
 | |


Enter position for O: 3
X| |O
-+-+-
 | |
-+-+-
 | |


X| |O
-+-+-
X| |
-+-+-
 | |


Enter position for O: 7
X| |O
-+-+-
X| |
-+-+-
O| |
X| |O
-+-+-
X|X|
-+-+-
O| |


Enter position for O: 9
X| |O
-+-+-
X|X|
-+-+-
O| |O


X| |O
-+-+-
X|X|X
-+-+-
O| |O


Bot wins!
```

2. Implement vacuum cleaner agent.

```
def vacuum_world():
goal_state = {'A': '0', 'B': '0'}
cost = 0
location_input = input("Enter Location of Vacuum (A or B): ")
status_input = input(f"Enter status of {location_input} (0 for clean, 1 for dirty): ")
status_input_complement = input("Enter status of the other room (0 for clean, 1 for dirty): ")
print("Initial Location Condition:", goal_state)
if location_input == 'A':
    print("Vacuum is placed in Location A."
    if status_input == '1':
        print("Location A is Dirty.")
        goal_state['A'] = '0'
        cost += 1
        print("Cost for CLEANING A:", cost)
        print("Location A has been Cleaned.")

    if status_input_complement == '1':
        print("Location B is Dirty.")
        print("Moving right to Location B.")
        cost += 1  # Cost for moving
        print("COST for moving RIGHT:", cost)
        goal_state['B'] = '0'
        cost += 1  # Cost for sucking
        print("COST for SUCK:", cost)
        print("Location B has been Cleaned.")
    else:
        print("Location B is already clean.")

elif location_input == 'B':
    print("Vacuum is placed in Location B.")

    if status_input == '1':
```

```python
        print("Location B is Dirty.")
        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING B:", cost)
        print("Location B has been Cleaned.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving left to Location A.")
            cost += 1  # Cost for moving
            print("COST for moving LEFT:", cost)
            goal_state['A'] = '0'
            cost += 1  # Cost for sucking
            print("COST for SUCK:", cost)
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")
    else:
        print("Location B is already clean.")
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving left to Location A.")
            cost += 1  # Cost for moving
            print("COST for moving LEFT:", cost)
            goal_state['A'] = '0'
            cost += 1  # Cost for sucking
            print("COST for SUCK:", cost)
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")

else:
    print("Invalid location input. Please enter A or B.")
```

```
    print("GOAL STATE:", goal_state)
    print("Performance Measurement:", cost)


# To run the function:
vacuum_world()
```

Output:

```
Enter Location of Vacuum (A or B):  A
Enter status of A (0 for clean, 1 for dirty):  0
Enter status of the other room (0 for clean, 1 for dirty):  1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A.
Location B is Dirty.
Moving right to Location B.
COST for moving RIGHT: 1
COST for SUCK: 2
Location B has been Cleaned.
GOAL STATE: {'A': '0', 'B': '0'}
Performance Measurement: 2
```

LAB – 02:

3. Solve 8 Puzzle problem using DFS and BFS.

BFS:

```
from collections import deque
class EightPuzzleBFS:
    def __init__(self, start_state, goal_state):
        self.start_state = start_state
        self.goal_state = goal_state
        self.visited = set()
    def get_neighbors(self, state):
        zero_idx = state.index(0)
        neighbors = []
  row, col = divmod(zero_idx, 3)
   directions = {
            'up': (-1, 0),
            'down': (1, 0),
            'left': (0, -1),
```

```
            'right': (0, 1)
        }
        for dr, dc in directions.values():
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_idx = new_row * 3 + new_col
                new_state = list(state)
 new_state[zero_idx], new_state[new_idx] = new_state[new_idx], new_state[zero_idx]
                neighbors.append(new_state)
    def solve(self):
        queue = deque([(self.start_state, [self.start_state])])
        self.visited.add(tuple(self.start_state))
        while queue:
            current_state, path = queue.popleft()
            if current_state == self.goal_state:
                return path  # Return the full path of states
            for neighbor in self.get_neighbors(current_state):
                if tuple(neighbor) not in self.visited:
                    self.visited.add(tuple(neighbor))
                    queue.append((neighbor, path + [neighbor]))
        return "No solution found"
def print_puzzle(state):
    """Print the puzzle state in a 3x3 grid format."""
    print("\n".join([" ".join(map(str, state[i:i + 3])) for i in range(0, 9, 3)]))
start_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]  # Initial configuration
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]   # Goal configuration
solver = EightPuzzleBFS(start_state, goal_state)
solution = solver.solve()
if solution != "No solution found":
    for idx, state in enumerate(solution):
        print(f"Step {idx}:\n")
        print_puzzle(state)
        print()  # Add a blank line for better separation
else:
```

```
    print(solution)
```

Output:

```
Step 0:

1 2 3
4 0 5
6 7 8

Step 1:

1 2 3
4 5 0
6 7 8

Step 2:

1 2 3
4 5 8
6 7 0

Step 3:

1 2 3
4 5 8
6 0 7

Step 4:

1 2 3
4 5 8
0 6 7

Step 5:

1 2 3
0 5 8
4 6 7

Step 6:

1 2 3
5 0 8
4 6 7

Step 7:

1 2 3
5 6 8
4 0 7

Step 8:

1 2 3
5 6 8
4 7 0

Step 9:

1 2 3
5 6 0
4 7 8

Step 10:

1 2 3
5 0 6
4 7 8

Step 11:

1 2 3
0 5 6
4 7 8

Step 12:

1 2 3
4 5 6
0 7 8

Step 13:

1 2 3
4 5 6
7 0 8

Step 14:

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

DFS:

```
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)
```

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
```