ARTIFICIAL INTELLIGENCE -ASSIGNMENT1

Name: - SuryaPraveen Adivi (001038932)

Purna Sarovar (001078181)

In this program, we use the A* algorithm with the Manhattan distance as the heuristic to solve the famous 8-puzzle problem. You're presented with a 3x3 grid with eight numbers and a blank space as the test. Rearranging the tiles such that the free space is at a predetermined spot on the board is a requirement for victory. The approach grows nodes and calculates their costs to determine the cheapest path from the origin to the destination, using the Manhattan distance heuristic.

To keep track of where a puzzle is in its evolution, the code defines a Node class with the attributes node State to record the current puzzle state, super Node to store the parent node, and each Node to record the activity that led to this node. The add Nodes class, which represents a queue of nodes, has operations to add nodes to, delete nodes from, and check the presence of nodes. The queofStack class is an extension of the add Nodes class that implements a queue by treating a list as a stack.

The Puzzle class encapsulates the problem by defining properties such as start Point (the problem's initial condition), results (its end state), and resultantReq (a location to keep track of the steps required to go from start Point to results), amongst others. The node's successors, or the states that may be reached from the current state by rearranging tiles, are generated by the node Relatives function. The missedNodeState process checks whether a state has been visited before, cutting down on back-and-forth traffic. As part of its solution, the A* algorithm employs the Manhattan distance heuristic to attribute costs to individual nodes. The final path used by the search and the total number of nodes visited may be shown by using display Result.

Code: -

```
import numpy as np


class Node:
    def __init__(self, nodeState, superNode, eachNode):
        self.nodeState = nodeState
        self.superNode = superNode
        self.eachNode = eachNode


class addNodes:
```

```
    def __init__(self):
        self.nodeListFront = []

    def add(self, node):

        self.nodeListFront.append(node)

    def nodeState(self, nodeState):

        i = 0
        while i < len(self.nodeListFront):
            node = self.nodeListFront[i]
            if (node.nodeState[0] == nodeState[0]).all():
                return True
            i += 1
        return False

    def checkingIsNull(self):

        if self.nodeListFront:
            return False
        else:
            return True

    def deleteNode(self):

        if self.checkingIsNull():
            raise Exception("empty")
        else:
            node = self.nodeListFront[-1]
            self.nodeListFront = self.nodeListFront[:-1]
            return node


class queofStack(addNodes):
    def deleteNode(self):

        if self.checkingIsNull():
            raise Exception("empty nodeListFront")
        else:
            node = self.nodeListFront[0]
            self.nodeListFront = self.nodeListFront[1:]
            return node


class Puzzle:
    def __init__(self, startPoint, startIndex, results, resultsIndex):
        self.startPoint = [startPoint, startIndex]
        self.results = [results, resultsIndex]
        self.resultantReq = None

    def nodeRelatives(self, nodeState):

        puzzleMatririx, (rowws, cols) = nodeState
        resultant = []

        if rowws > 0:
```

```
            puzzleMatririx1 = np.copy(puzzleMatririx)
            temp = puzzleMatririx1[rowws][cols]
            puzzleMatririx1[rowws][cols] = puzzleMatririx1[rowws - 1][cols]
            puzzleMatririx1[rowws - 1][cols] = temp
            resultant.append(('up', [puzzleMatririx1, (rowws - 1, cols)]))

        if cols > 0:
            puzzleMatririx1 = np.copy(puzzleMatririx)
            temp = puzzleMatririx1[rowws][cols]
            puzzleMatririx1[rowws][cols] = puzzleMatririx1[rowws][cols - 1]
            puzzleMatririx1[rowws][cols - 1] = temp
            resultant.append(('left', [puzzleMatririx1, (rowws, cols - 1)]))

        if rowws < 2:
            puzzleMatririx1 = np.copy(puzzleMatririx)
            temp = puzzleMatririx1[rowws][cols]
            puzzleMatririx1[rowws][cols] = puzzleMatririx1[rowws + 1][cols]
            puzzleMatririx1[rowws + 1][cols] = temp
            resultant.append(('down', [puzzleMatririx1, (rowws + 1, cols)]))

        if cols < 2:
            puzzleMatririx1 = np.copy(puzzleMatririx)
            temp = puzzleMatririx1[rowws][cols]
            puzzleMatririx1[rowws][cols] = puzzleMatririx1[rowws][cols + 1]
            puzzleMatririx1[rowws][cols + 1] = temp
            resultant.append(('right', [puzzleMatririx1, (rowws, cols + 1)]))

        return resultant

    def displayResult(self):
        resultantReq = self.resultantReq if self.resultantReq is not None
else None
        print("starting state of the node:\n", self.startPoint[0], "\n")
        print("result state of node:\n", self.results[0], "\n")
        print("\n visited nodes: ", self.visitedNodes, "\n")
        print("results :\n ")
        resultPath=[]
        resultPath = [eachNode for eachNode, everyCell in
zip(resultantReq[0], resultantReq[1])]

        i = 0
        while i < len(resultantReq[0]):
            eachNode = resultantReq[0][i]
            everyCell = resultantReq[1][i]
            print(" move--->: ", eachNode, "\n", everyCell[0], "\n")
            i += 1

        print("resultant path is ", resultPath)


    def missedNodeState(self, nodeState):

        return not any((st[0] == nodeState[0]).all() for st in
self.explored_nodes)


    def solving(self):
```

```
        self.visitedNodes = 0

        startPoint = Node(nodeState=self.startPoint, superNode=None,
eachNode=None)
        nodeListFront = queofStack()
        nodeListFront.add(startPoint)

        self.explored_nodes = []

        while not nodeListFront.checkingIsNull():
            node = nodeListFront.deleteNode()
            self.visitedNodes += 1

            if (node.nodeState[0] == self.results[0]).all():
                resultantReq = []
                while node.superNode is not None:
                    resultantReq.append((node.eachNode, node.nodeState))
                    node = node.superNode
                resultantReq.reverse()
                each_node_of_action, every_cell = zip(*resultantReq)
                self.resultantReq = (list(each_node_of_action),
list(every_cell))
                return

            self.explored_nodes.append(node.nodeState)

            new_nodeStates = [(eachNode, nodeState) for eachNode, nodeState
in self.nodeRelatives(node.nodeState)
                              if not nodeListFront.nodeState(nodeState) and
self.missedNodeState(nodeState)]

            [nodeListFront.add(Node(nodeState=nodeState, superNode=node,
eachNode=eachNode)) for eachNode, nodeState in
             new_nodeStates]


print("-----------------------Start Matrix----------------------------------
")

rows = int(input("Enter the number of rows of start matrix: "))
cols = int(input("Enter the number of columns of start matrix: "))

startMatrix = np.array([[int(input(f"Enter element [{i}][{j}]: ")) for j in
range(cols)] for i in range(rows)])
startMatrix = np.array(startMatrix)
print("start matrix :",startMatrix)

print("-----------------------Result Matrix---------------------------------
")

print("Give input for result matrix :")
rows = int(input("Enter the number of rows of result matrix: "))
cols = int(input("Enter the number of columns of result matrix: "))

resultMatrix = [[int(input(f"Enter element [{i}][{j}]: ")) for j in
range(cols)] for i in range(rows)]
resultMatrix = np.array(resultMatrix)
```

```
print("resultmatrix is ", resultMatrix)
startMatrixIndices = (1, 1)
resultmatrixIndices = (1, 0)

p = Puzzle(startMatrix, startMatrixIndices, resultMatrix,
resultmatrixIndices)
p.solving()
p.displayResult()
```