8 Puzzle Problem using A Star

Table of Contents

- 8 Puzzle Problem
    - Table of contents
    - Introduction
    - A Star Algorithm
    - Running the code
    - Problem Formualtion
    - Program Structure
    - Global Variable
    - Functions Used
    - Sample Input and Outputs
    - Source Code

Introduction

8 Puzzle or 8 Tile Problem is played on 3x3 grid with 9 tiles numbered from 0-8. The tile with the number 0 is considered as blank tile. The actual goal of this problem is to arrange the tiles in a sequential order. The blank tile can move through the 9 tiles in UP, DOWN, RIGHT, LEFT directions. The goal state of the 8 puzzle problem is shown below:

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

Sometimes the goal state vary according to the user requriements.

A Star Algorithm

Running the Code

The Programming Language used is Python 3.7. Running the Single file will execute the 8 Puzzle Problem. Running the python file will give the transition from initial state to the goal state and also gives the number of nodes generated and number of nodes expanded.

Problem Formulation

Goal: All the tiles end up in the state described by the goal state
States: All the permutation of states possible in the 8 Puzzle Problem
Action: The blank tile can move UP, DOWN, RIGHT, LEFT Performance: To reduce the number of transitions that take from initial to goal state.

Program Structure

The Program has mainly 3 classes node, ASTAR and Puzzle8. The node class has all the properties that node state will have i.e., whether the node is the child, parent or the intial node and that heuristic values it is associated with. It also parses from the parent node to the child node. The ASTAR class implements A star algorithm and calculates the f, g, h values to check whether the goal state has reached or not. The Puzzle8 class has all the variables and functions that help in solving the 8 Puzzle problem.

Global Variable

There are no implementations for Global Variables in this project

Functions Used

```
def __init__(self,child,parent):
def setp(self, p):
def getp(self, p):
```

The above functions are defined in the node class of the program. The `__init__` will have the properties related to the state. The g, h, f values are declared in this function. The child and parent members are used to find out whether the node is a child or a parent node.

```
def GoalState(self,state)
def G(self,parent):
def Path(self,n):
def generateChildrennodes(self,n):
def Nodes(self, n):
def LowerF(self,lists,n):
def EqualStates(self, state1, state2):
def F(self,n):
def AstarSearch(self):
```

The above functions are used in the ASTAR class of the program. `GoalState` , `G`, `Path`, `generateChildrennodes`, `Nodes`, `LowerF`, `EqualStates` are the functions which are overriden by the classes. `F` is used to the get the value of the heuristic function F. `AStarSearch` implements the A star search algorithm.

```
def nodecreation(self,state=None,parent=None):
def G(self, parent):
def GoalState(self,state):
def EqualStates(self,state1,state2):
def Index(self,state,number):
def H(self,state):
```

```
def printstate(self,state):
def Path(self,n):
def generateChildrennodes(self,n):
```

The above functions belong to the Puzzle8 class of the progarm. `nodecreation` creates a child node from the respective parent node. `G` function calculates the g value of the node at a particular state. `EqualStates` function is used to check if the tile positions of the states are exactly same as each other. `Index` function is used to give the (i,j) of a state. `H` calculates the heuristic value of the particular tile. `printstate` and `Path` are the functions which print the state and path to the goal state of the 8 puzzle respectively. `generateChildrennodes` function is used to generate child or the next states of the 8 puzzle problem.

Sample Input and Outputs

Sample 1:

Input state:

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

Goal state:

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

Path Trace:

1. Manhattan Distance Heuristic:

```
1 2 3
7 4 5
6 8

1 2 3
7 4
6 8 5

1 2 3
7   4
6 8 5

1 2 3
7 8 4
6   5

1 2 3
7 8 4
  6 5

1 2 3
  8 4
7 6 5

1 2 3
8   4
7 6 5

1 2 3
8 6 4
7   5

1 2 3
8 6 4
7 5
```

   The number of nodes generated: 27
   The number of nodes expanded: 9

2. Misplaced Tiles Heuristic:

```
1 2 3
7 4 5
6 8

1 2 3
7 4
6 8 5

1 2 3
7   4
6 8 5

1 2 3
7 8 4
6   5

1 2 3
7 8 4
  6 5

1 2 3
  8 4
7 6 5

1 2 3
```

```
8     4
7  6  5

1  2  3
8  6  4
7     5

1  2  3
8  6  4
7  5
```

The number of nodes generated: 63
The number of nodes expanded: 21

Sample 2:

Input State:

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

Goal State:

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

Path Trace:

1. Manhattan Distance Heuristic:

```
2  8  1
3  4  6
7  5

2  8  1
3  4
7  5  6

2  8  1
3     4
7  5  6

2     1
3  8  4
7  5  6

   2  1
3  8  4
7  5  6

3  2  1
   8  4
7  5  6

3  2  1
8     4
7  5  6
```

The number of nodes generated: 18
The number of nodes expanded: 6

2. Misplaced Tiles Heuristic

```
2  8  1
3  4  6
7  5

2  8  1
3  4
7  5  6

2  8  1
3     4
7  5  6

2     1
3  8  4
7  5  6

   2  1
3  8  4
7  5  6

3  2  1
   8  4
7  5  6

3  2  1
8     4
7  5  6
```

The number of nodes generated: 21
The number of nodes expanded: 7

Sample 3:

Input State:

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

Goal State:

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

Path Trace:

1. Manhattan Distance Heuristic:

```
1 2 3
8   4
7 6 5

1   3
8 2 4
7 6 5

  1 3
8 2 4
7 6 5

8 1 3
  2 4
7 6 5

8 1 3
2   4
7 6 5

8 1 3
2 4
7 6 5

8 1
2 4 3
7 6 5

8   1
2 4 3
7 6 5

  8 1
2 4 3
7 6 5

2 8 1
  4 3
7 6 5
```

The number of nodes generated: 40
The number of nodes expanded: 14

2. Misplaced Tiles Heuristic:

```
1 2 3
8   4
7 6 5

1   3
8 2 4
7 6 5

  1 3
8 2 4
7 6 5

8 1 3
  2 4
7 6 5

8 1 3
2   4
7 6 5

8 1 3
2 4
7 6 5

8 1
2 4 3
7 6 5

8   1
2 4 3
7 6 5

  8 1
2 4 3
7 6 5
```

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

The number of nodes generated: 109
The number of nodes expanded: 38

Sample 4: Initial State:

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

Goal State:

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

Path Trace: 1.Manhattan Distance Heuristic:

```
3   7
2 8 1
6 4 5

3 8 7
2   1
6 4 5

3 8 7
2 1
6 4 5

3 8
2 1 7
6 4 5

3   8
2 1 7
6 4 5

  3 8
2 1 7
6 4 5

2 3 8
  1 7
6 4 5

2 3 8
1   7
6 4 5

2 3 8
1 7
6 4 5

2 3
1 7 8
6 4 5

2   3
1 7 8
6 4 5

  2 3
1 7 8
6 4 5

1 2 3
  7 8
6 4 5

1 2 3
7   8
6 4 5

1 2 3
7 4 8
6   5

1 2 3
7 4 8
  6 5

1 2 3
  4 8
7 6 5

1 2 3
4   8
7 6 5

1 2 3
4 8
7 6 5

1 2 3
```

```
4 8 5
7 6

1 2 3
4 8 5
7   6

1 2 3
4   5
7 8 6

1 2 3
4 5
7 8 6

1 2 3
4 5 6
7 8
```

The number of nodes generated: 1704
The number of nodes expanded: 646

2. Misplaced Tiles Heuristic:

```
2 8 1
6 4 5

3 8 7
2   1
6 4 5

3 8 7
2 1
6 4 5

3 8
2 1 7
6 4 5

3   8
2 1 7
6 4 5

  3 8
2 1 7
6 4 5

2 3 8
  1 7
6 4 5

2 3 8
1   7
6 4 5

2 3 8
1 7
6 4 5

2 3
1 7 8
6 4 5

2   3
1 7 8
6 4 5

  2 3
1 7 8
6 4 5

1 2 3
  7 8
6 4 5

1 2 3
7   8
6 4 5

1 2 3
7 4 8
6   5

1 2 3
7 4 8
  6 5

1 2 3
  4 8
7 6 5

1 2 3
4   8
7 6 5

1 2 3
4 8
7 6 5
```

```
1 2 3
4 8 5
7 6

1 2 3
4 8 5
7   6

1 2 3
4   5
7 8 6

1 2 3
4 5
7 8 6

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

3 8 7
2   1
6 4 5

3 8 7
2 1
6 4 5

3 8
2 1 7
6 4 5

3   8
2 1 7
6 4 5

 3 8
2 1 7
6 4 5

2 3 8
 1 7
6 4 5

2 3 8
1   7
6 4 5

2 3 8
1 7
6 4 5

2 3
1 7 8
6 4 5

2   3
1 7 8
6 4 5

 2 3
1 7 8
6 4 5

1 2 3
 7 8
6 4 5

1 2 3
7   8
6 4 5

1 2 3
7 4 8
6   5

1 2 3
7 4 8
 6 5

1 2 3
 4 8
7 6 5

1 2 3
4   8
7 6 5

1 2 3
4 8
7 6 5

1 2 3
4 8 5
7 6

1 2 3
```

```
4 8 5
7   6

1 2 3
4   5
7 8 6

1 2 3
4 5
7 8 6

1 2 3
4 5 6
7 8
```

The number of nodes generated: 50765
The number of nodes expanded: 18724

Source Code:

```
import copy
import heapq



class node:
    def __init__(self,state,parent = None):             #At the initial state parent node will be none
        self.state = state                              #storing the state of node
        self.parent = parent                            #storing the parent node
        self.h = 0                                      #storing value of H
        self.f = 0                                      #storing value of f
        self.g = 0                                      #storing value of g

    def __lt__(self,ob1):
        return ob1.f

    def setP(self,p):
        self.parent = p

    def getP(self):
        return self.parent

class ASTAR:
    def __init__(self,inputState):
        self.Expandednodes = []                         #List of closed nodes
        self.Fringe = []                                #Priority Queue for open nodes
        self.inputState = inputState                    #Initial state entered by the user

    def AstarSearch(self):
        if(self.GoalState(self.inputState)):
            print('Input is Goal state')
            return None
        while len(self.Fringe) != 0:

            f, x = heapq.heappop(self.Fringe)            #poping nodes from the open node priority queue
            childs = self.generateChildrennodes(x)
            for child in childs:
                if(self.GoalState(child.state)):         #Goal achieved
                    print('Goal Reached ',child.state)   #Printing Goal
                    return child
                if self.LowerF(self.Fringe,child) ==False:
                    heapq.heappush(self.Fringe,(child.f,child))
                elif self.LowerF(self.Expandednodes,child) == False:
                    heapq.heappush(self.Fringe,(child.f,child))
            heapq.heappush(self.Expandednodes,(x.f,x))       #Adding nodes to the closed list.
        return None

    def GoalState(self,state):
        pass
    def LowerF(self,lists,x):   #samestatewithlowerF
        pass
    def G(self,parent):
        pass
    def Path(self,x):
        pass
    def Equalstates(self, state1, state2):
        pass
    def generateChildrennodes(self, x):
        pass
    def F(self,x):
        return x.g + x.h
    def Nodes(self, n):
        pass


class Puzzle8(ASTAR):
    def __init__(self,inputState,goalState,manhattandistance):
        ASTAR.__init__(self,inputState)
        self.goalState = goalState
        self.manhattandistance = manhattandistance
        self.noofNodesGenerated = 0
        x = self.nodecreation()
        heapq.heappush(self.Fringe,(x.f, x))

    def GoalState(self, state):
        for i in range(3):
            for j in range(3):
                if state[i][j] != self.goalState[i][j]:
```

```
                return False
        return True

    def nodecreation(self,state=None,parent=None):
        self.noofNodesGenerated += 1
        if state is None:
            state = self.inputState
            x = node(state)
            x.g = 0
            x.h = self.H(state)
            x.f = self.F(x)
            return x
        else:
            x = node(state, parent)
            x.g = self.G(parent)
            x.h = self.H(state)
            x.f = self.F(x)
            return x

    def Equalstates(self,state1,state2):    #Checking for equal states
        for i in range(3):
            for j in range(3):
                if state1[i][j] != state2[i][j]:
                    return False
        return True

    def Index(self,state,number):
        for i, row in enumerate(state):
            try:
                j = row.index(number)
            except ValueError:
                continue
            return i, j
    def G(self, parent):
        return parent.g+1

    def manhattanHvalue(self,state):         #Value of H when using Manhattan Distance as the Heuristic
        distance = 0
        for i in range(3):
            for j in range(3):
                if (state[i][j] != 0):
                    (goalRow,goalColumn) = self.Index(self.goalState,state[i][j])
                    distance += abs(goalColumn - j) + abs(goalRow - i)
        return distance

    def H(self,state):
        if(self.manhattandistance):
            return self.manhattanHvalue(state)
        else:
            return self.misplacedHvalue(state)
    def misplacedHvalue(self,state):           #Value of H when using Misplaced Tiles as the Heuristic
        distance = 0
        for i in range(3):
            for j in range(3):
                if ((state[i][j] !=0) and  (state[i][j] != self.goalState[i][j])):
                    distance +=1
        return distance

    def Path(self,x):
        path = []
        while(True):
            if(x is None):
                break
            else:
                path.append(x.state)
                x = x.parent
        print('The Path length is ',len(path))
        print('Path Trace')
        for state in reversed(path):
            self.printstate(state)

    def Nodes(self,x):
        print('Values are g = {}, f = {}, h = {}, and state = {}'.format(x.g,x.f,x.h,x.state))


    def printstate(self,state):

        for i in range(len(state)):
            for j in range(len(state[i])):
                if state[i][j] == 0:
                    print(' ',end=' ')
                else:
                    print(state[i][j], end=' ')
            print()
        print()


    def generateChildrennodes(self,x):

        abc = []
        children = []
        i,j = self.Index(x.state,0)
        properindexes = []     #Generating Indexes
        #UP LEFT DOWN RIGHT
        if i-1 >=0:
            properindexes.append((i-1,j))
            abc.append("Down")
        if j-1 >=0:
            properindexes.append((i,j-1))
            abc.append("Right")
```

```
        if i+1 <=2:
            properindexes.append((i+1,j))
            abc.append("Up")
        if j+1 <=2:
            properindexes.append((i,j+1))
            abc.append("Left")
        for k in abc:
            print(k)



        for index,(row,col) in enumerate(properindexes):
            state = copy.deepcopy(x.state)
            y = state[i][j]
            state[i][j] = state[row][col]
            state[row][col] = y
            children.append(self.nodecreation(state,x))
        return children







    def LowerF(self,list,x):
        for i,l in list:
            if self.Equalstates(l.state,x.state):
               if(x.f<l.f):
                    l = x
                    return True
               else:
                   return True
        return False

def parseInput():
    number1 = input()
    number2 = input()
    number3 = input()
    inputlist = [list(map(int, number1.split(' '))),list(map(int, number2.split(' '))),list(map(int, number3.split(' ')))]
    return inputlist


if __name__ == '__main__':
    print('Please enter the input state:')
    inputState = parseInput()
    print('Please enter the goal state:')
    goalState = parseInput()
    Inputlen = inputState[0]+inputState[1]+inputState[2]
    Goallen = goalState[0]+goalState[1]+goalState[2]
    if len(Inputlen) != 9 or len(Goallen) !=9:
        print('Invalid Input')
    elif len(set(Inputlen) - set([1,2,3,4,5,6,7,8,0])) > 0 or len(set(Goallen) - set([1,2,3,4,5,6,7,8,0])) > 0:
        print('Invalid Input')
    else:
        manhattandistance = False
        Heuristic = input('Heuristic function?\n1.Manhattan Distance \n2.Misplaced Tiles\n(1/2) :') #Taking the Heuristic Type Input
        Heuristic = int(Heuristic)
        if Heuristic == 1:
            manhattandistance = True     #If manhattandistance is true, it'll be considered as the heuristic. Else, Misplaced Tiles.
        print('Directions are -')
        P = Puzzle8(inputState,goalState,manhattandistance)
        goalNode = P.AstarSearch()                              #Implementing A* Algorithm
        P.Path(goalNode)                                        #The path from Input to Goal State
        print('Generated Nodes ', P.noofNodesGenerated)         #Printing no. of generated nodes
        print('Expanded Nodes ', len(P.Expandednodes)+1)        #Printing no. of expanded nodes
```