GROUP MEMBERS

● KUMAR SANATAN
ROLL NO – 2211AI24

● CHANDERA RAVI
ROLL NO – 2211AI16

CS 571
ARTIFICIAL INTELLIGENC LAB
ASSIGMENT 3: Hill Climbing

INDIAN INSTITUTE OF TECHNOLOGY
PATNA

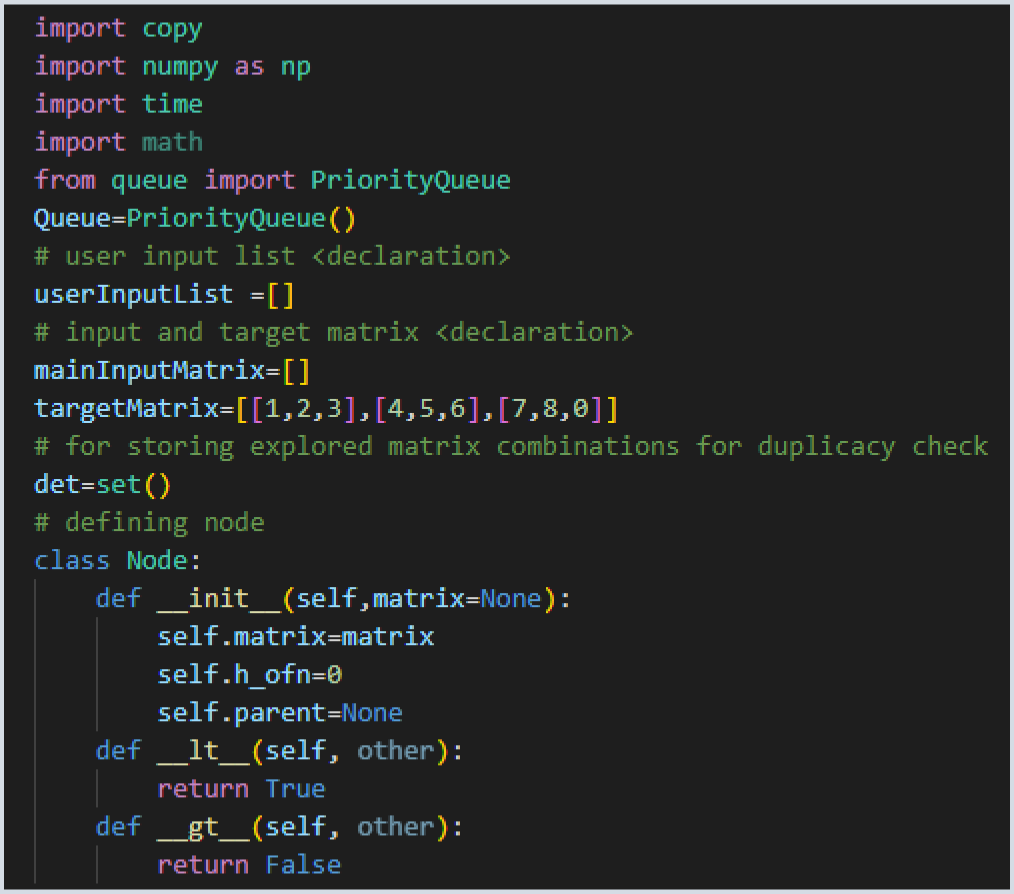
Date: 05th Sept. 2022 **Deadline:** 11th Sept. 2022

OBJECTIVE

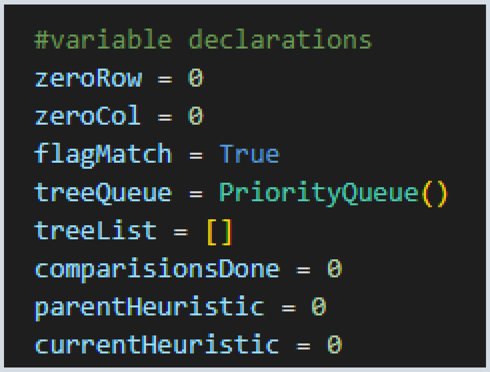
Implement the Hill Climbing Search Algorithm for solving the 8-puzzle problem

CODE BASE :

Importing the required libraries and defining the target matrix. We have defined the set where we are soring the matrices that have been traversed to check for duplicity. We have defined priority queue so as to store the matrix and corresponding heuristic serving as the priority. The initial node is defined to store the matrix, the heuristic value and the parent node.

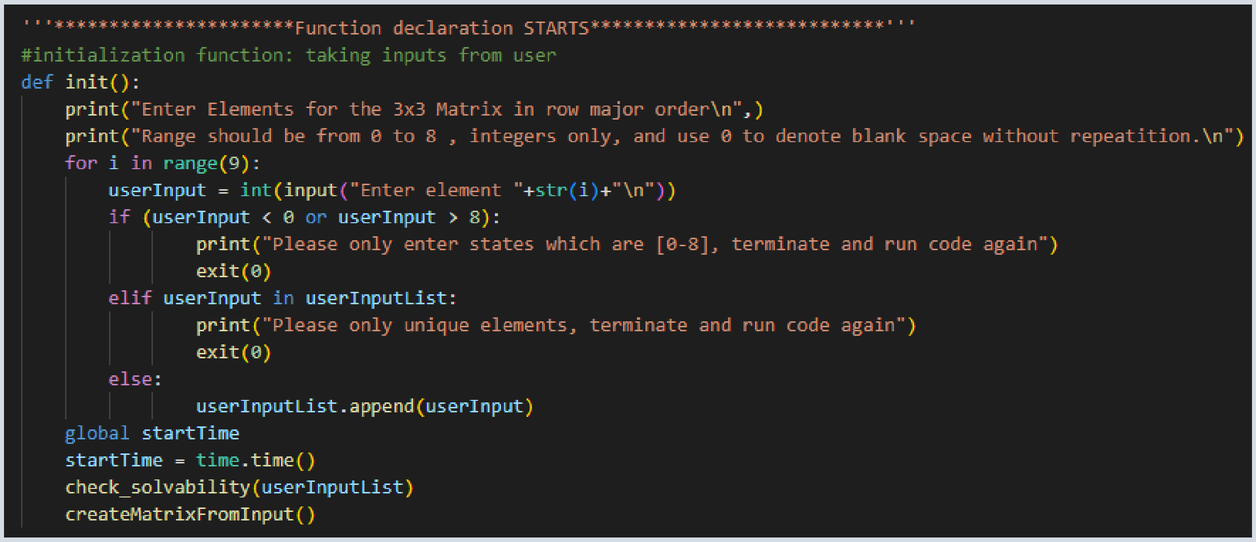


```
import copy
import numpy as np
import time
import math
from queue import PriorityQueue
Queue=PriorityQueue()
# user input list <declaration>
userInputList =[]
# input and target matrix <declaration>
mainInputMatrix=[]
targetMatrix=[[1,2,3],[4,5,6],[7,8,0]]
# for storing explored matrix combinations for duplicacy check
det=set()
# defining node
class Node:
    def __init__(self,matrix=None):
        self.matrix=matrix
        self.h_ofn=0
        self.parent=None
    def __lt__(self, other):
        return True
    def __gt__(self, other):
        return False
```

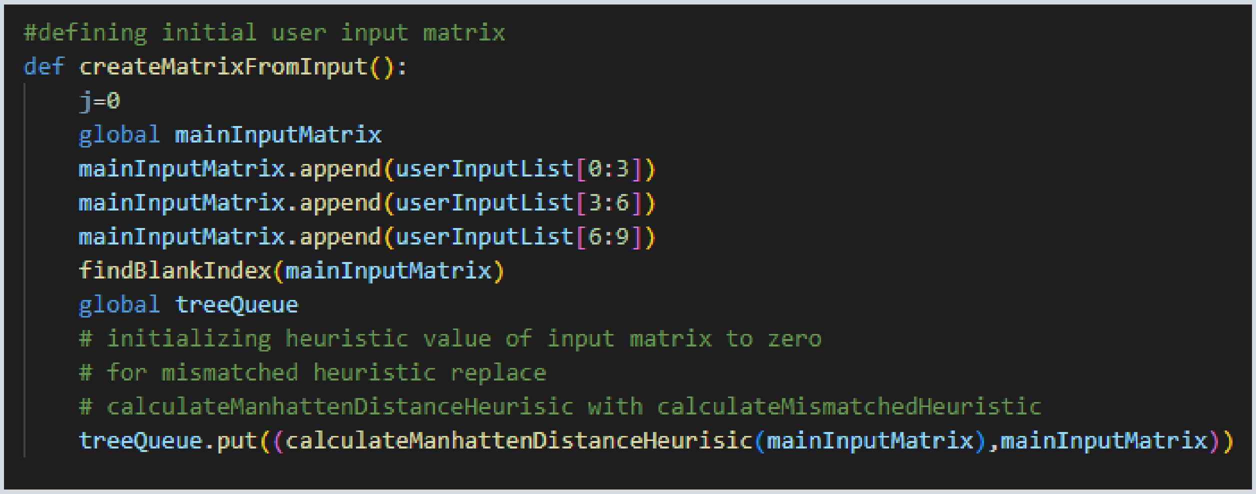


```
#variable declarations
zeroRow = 0
zeroCol = 0
flagMatch = True
treeQueue = PriorityQueue()
treeList = []
comparisionsDone = 0
parentHeuristic = 0
currentHeuristic = 0
```

We are taking random input from the user and performing all the check and balances so as to verify the correct input is being fed by the user and thereby creating the matrix from the input and storing it in the queue for performing further operations.

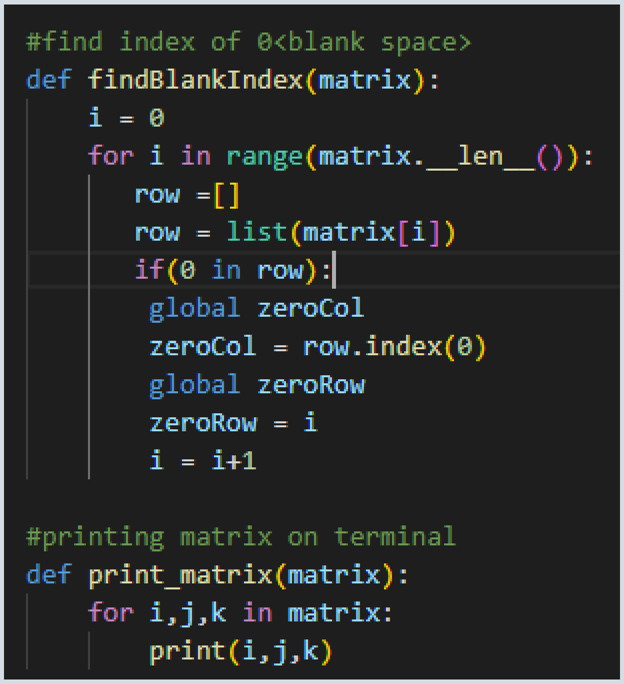


```python
'''**********************Function declaration STARTS***************************'''
#initialization function: taking inputs from user
def init():
    print("Enter Elements for the 3x3 Matrix in row major order\n",)
    print("Range should be from 0 to 8 , integers only, and use 0 to denote blank space without repeatition.\n")
    for i in range(9):
        userInput = int(input("Enter element "+str(i)+"\n"))
        if (userInput < 0 or userInput > 8):
                print("Please only enter states which are [0-8], terminate and run code again")
                exit(0)
        elif userInput in userInputList:
                print("Please only unique elements, terminate and run code again")
                exit(0)
        else:
                userInputList.append(userInput)
    global startTime
    startTime = time.time()
    check_solvability(userInputList)
    createMatrixFromInput()
```

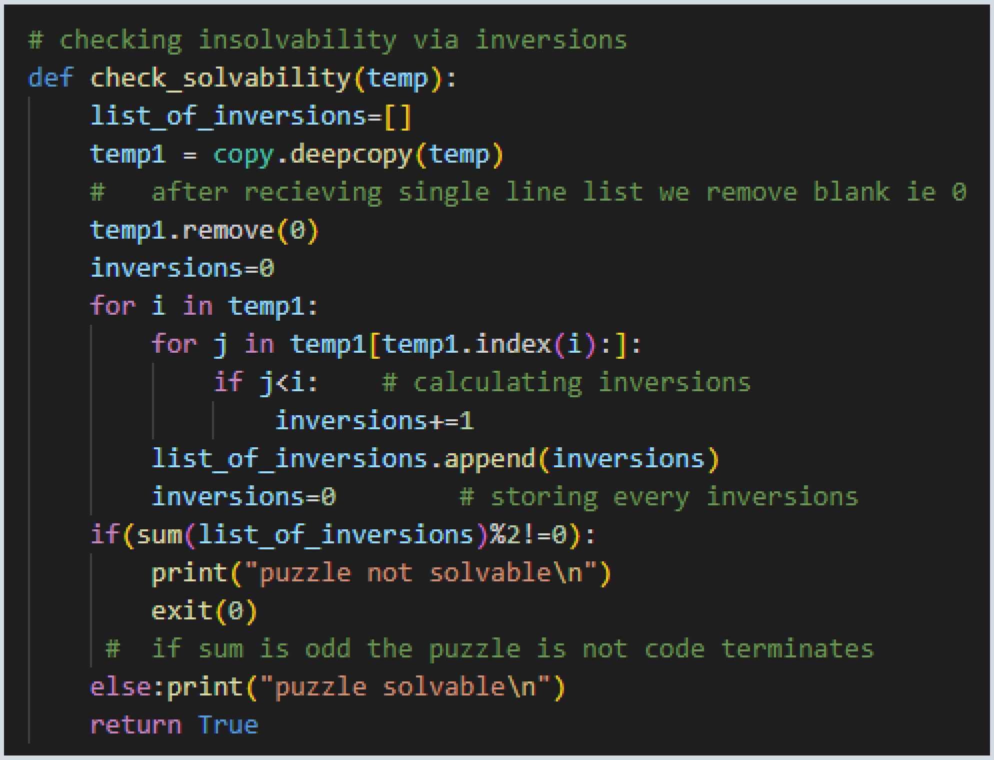


```python
#defining initial user input matrix
def createMatrixFromInput():
    j=0
    global mainInputMatrix
    mainInputMatrix.append(userInputList[0:3])
    mainInputMatrix.append(userInputList[3:6])
    mainInputMatrix.append(userInputList[6:9])
    findBlankIndex(mainInputMatrix)
    global treeQueue
    # initializing heuristic value of input matrix to zero
    # for mismatched heuristic replace
    # calculateManhattenDistanceHeurisic with calculateMismatchedHeuristic
    treeQueue.put((calculateManhattenDistanceHeurisic(mainInputMatrix),mainInputMatrix))
```

We check for solvability of the matrix via inversion and are finding the index of the blank position(depicted as 0) so as to perform all the feasible moves(left, right, up, down) of the blank element to reach to the goal state configuration.

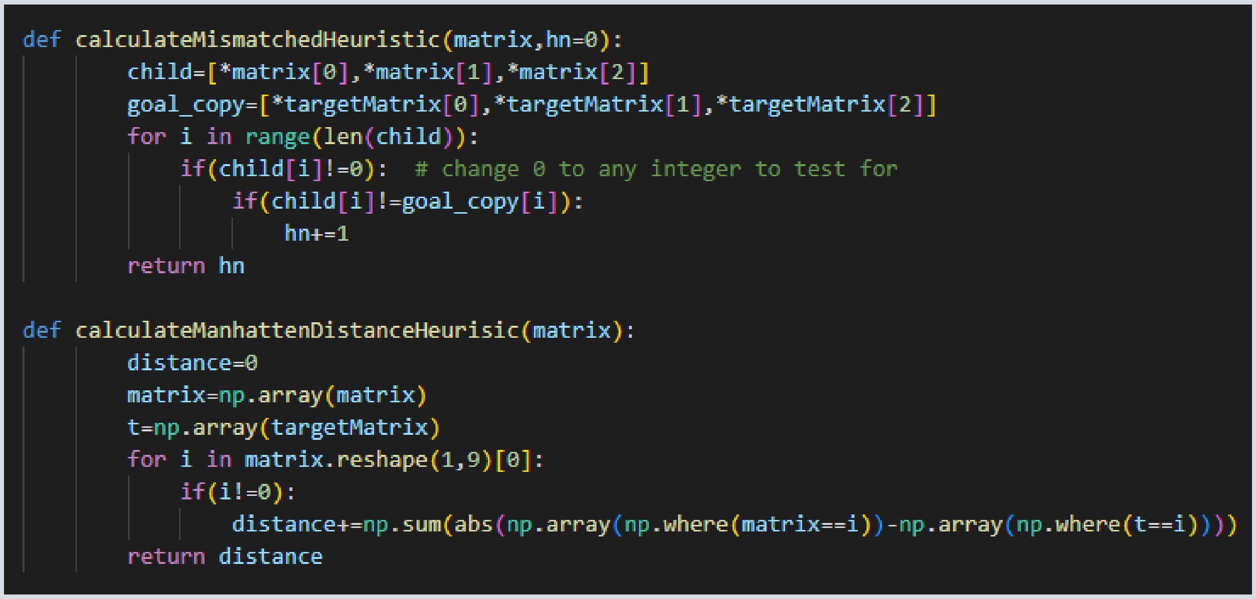


```
#find index of 0<blank space>
def findBlankIndex(matrix):
    i = 0
    for i in range(matrix.__len__()):
        row =[]
        row = list(matrix[i])
        if(0 in row):
         global zeroCol
         zeroCol = row.index(0)
         global zeroRow
         zeroRow = i
         i = i+1

#printing matrix on terminal
def print_matrix(matrix):
    for i,j,k in matrix:
        print(i,j,k)
```

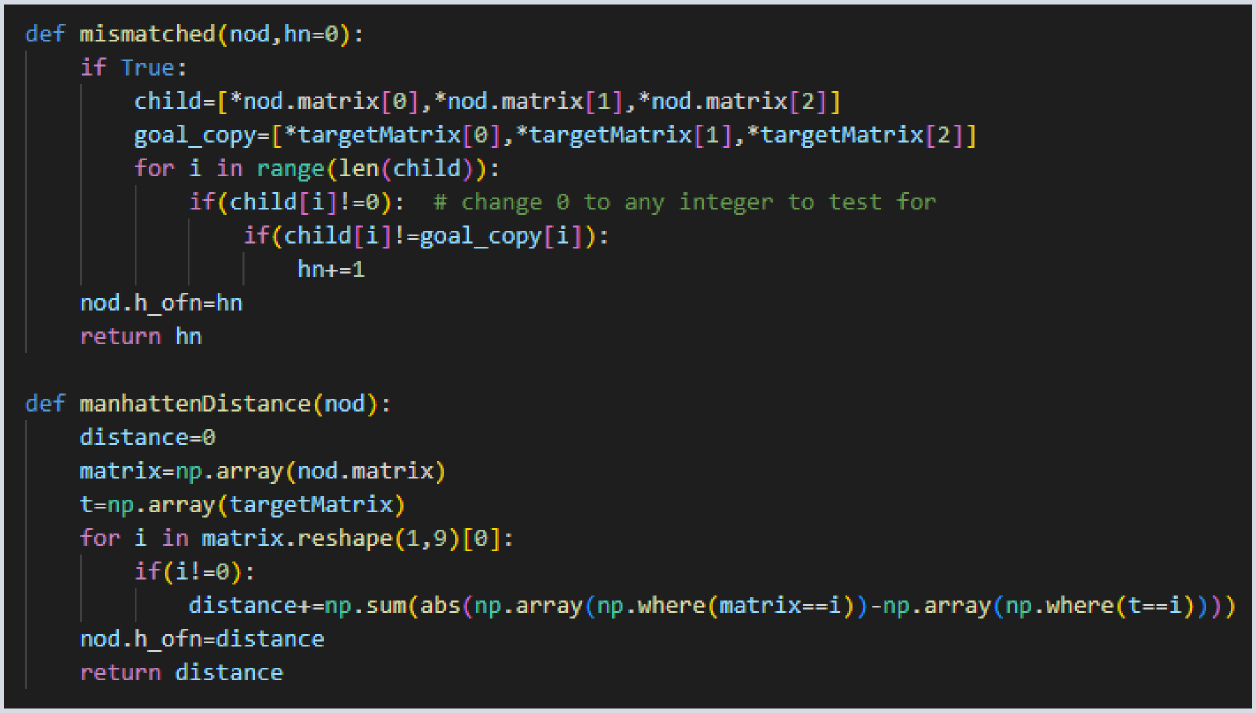


```
# checking insolvability via inversions
def check_solvability(temp):
    list_of_inversions=[]
    temp1 = copy.deepcopy(temp)
    #   after recieving single line list we remove blank ie 0
    temp1.remove(0)
    inversions=0
    for i in temp1:
        for j in temp1[temp1.index(i):]:
            if j<i:    # calculating inversions
                inversions+=1
        list_of_inversions.append(inversions)
        inversions=0        # storing every inversions
    if(sum(list_of_inversions)%2!=0):
        print("puzzle not solvable\n")
        exit(0)
     #  if sum is odd the puzzle is not code terminates
    else:print("puzzle solvable\n")
    return True
```

Here we have defined the 4 methods to calculate the heuristics (mismatched tiles and Manhattan distance)

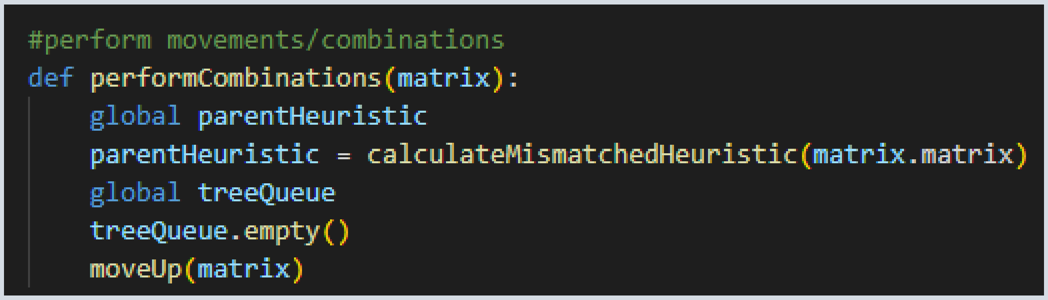


```python
def calculateMismatchedHeuristic(matrix,hn=0):
        child=[*matrix[0],*matrix[1],*matrix[2]]
        goal_copy=[*targetMatrix[0],*targetMatrix[1],*targetMatrix[2]]
        for i in range(len(child)):
            if(child[i]!=0):  # change 0 to any integer to test for
                if(child[i]!=goal_copy[i]):
                    hn+=1
        return hn

def calculateManhattenDistanceHeurisic(matrix):
        distance=0
        matrix=np.array(matrix)
        t=np.array(targetMatrix)
        for i in matrix.reshape(1,9)[0]:
            if(i!=0):
                distance+=np.sum(abs(np.array(np.where(matrix==i))-np.array(np.where(t==i))))
        return distance
```

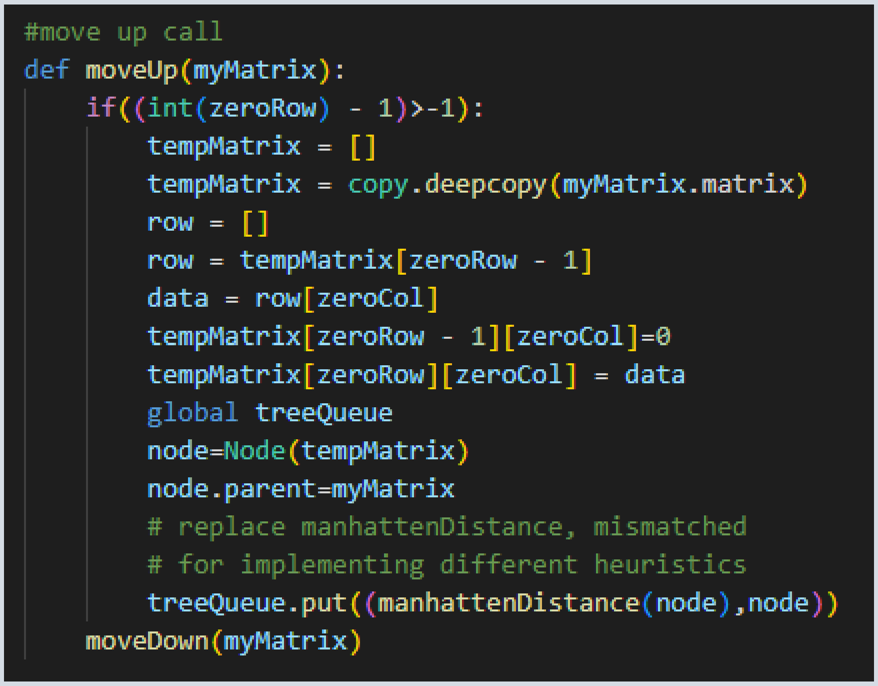


```python
def mismatched(nod,hn=0):
    if True:
        child=[*nod.matrix[0],*nod.matrix[1],*nod.matrix[2]]
        goal_copy=[*targetMatrix[0],*targetMatrix[1],*targetMatrix[2]]
        for i in range(len(child)):
            if(child[i]!=0):  # change 0 to any integer to test for
                if(child[i]!=goal_copy[i]):
                    hn+=1
    nod.h_ofn=hn
    return hn

def manhattenDistance(nod):
    distance=0
    matrix=np.array(nod.matrix)
    t=np.array(targetMatrix)
    for i in matrix.reshape(1,9)[0]:
        if(i!=0):
            distance+=np.sum(abs(np.array(np.where(matrix==i))-np.array(np.where(t==i))))
    nod.h_ofn=distance
    return distance
```

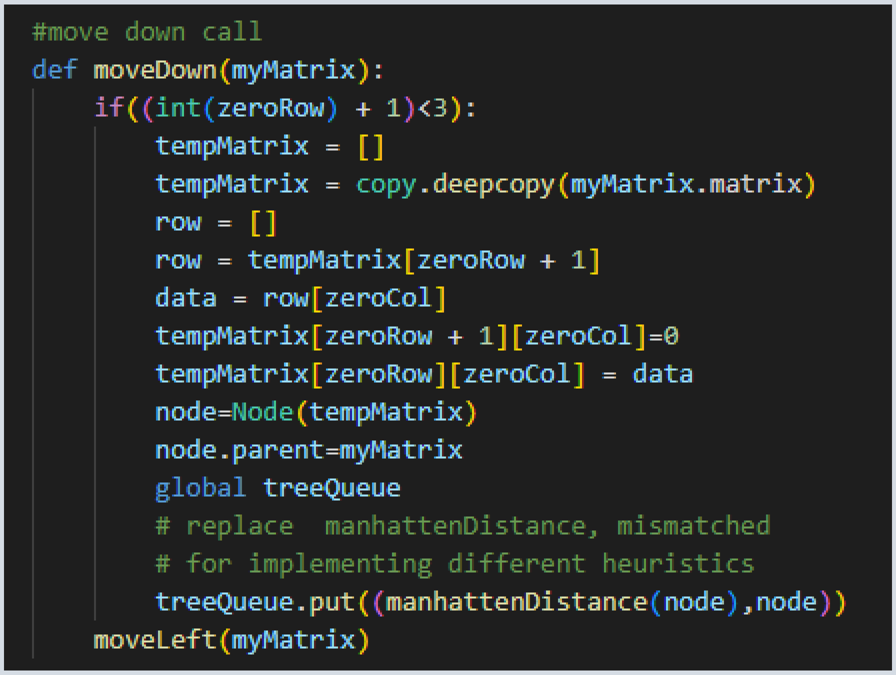
We are performing the left, right, up and down movements of the blank element here with respect to the corresponding heuristic use case.

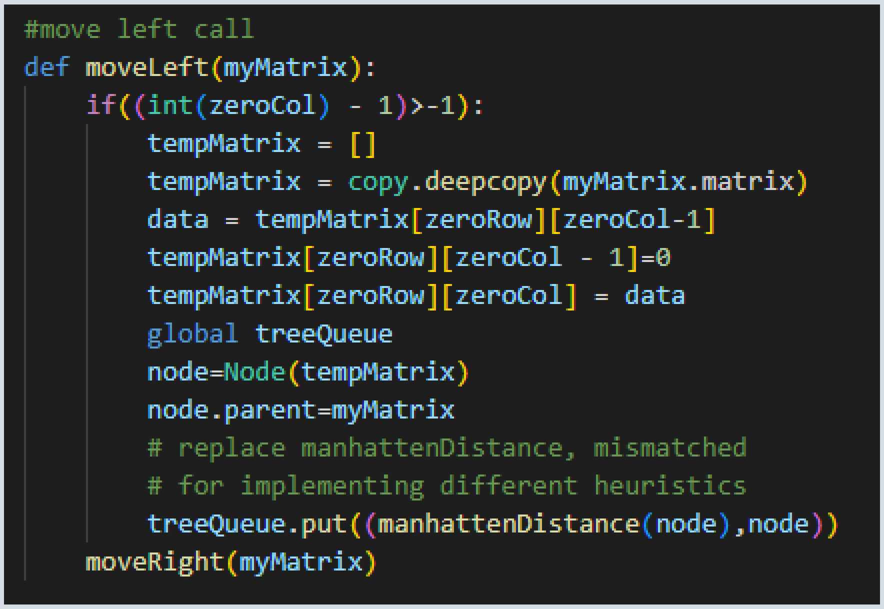


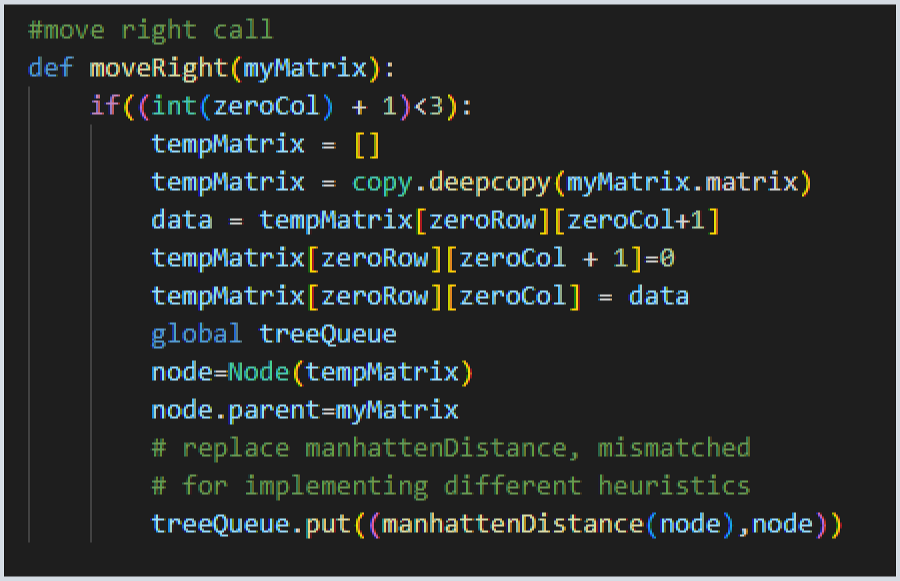
```
#perform movements/combinations
def performCombinations(matrix):
    global parentHeuristic
    parentHeuristic = calculateMismatchedHeuristic(matrix.matrix)
    global treeQueue
    treeQueue.empty()
    moveUp(matrix)
```



```
#move up call
def moveUp(myMatrix):
    if((int(zeroRow) - 1)>-1):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix.matrix)
        row = []
        row = tempMatrix[zeroRow - 1]
        data = row[zeroCol]
        tempMatrix[zeroRow - 1][zeroCol]=0
        tempMatrix[zeroRow][zeroCol] = data
        global treeQueue
        node=Node(tempMatrix)
        node.parent=myMatrix
        # replace manhattenDistance, mismatched
        # for implementing different heuristics
        treeQueue.put((manhattenDistance(node),node))
    moveDown(myMatrix)
```

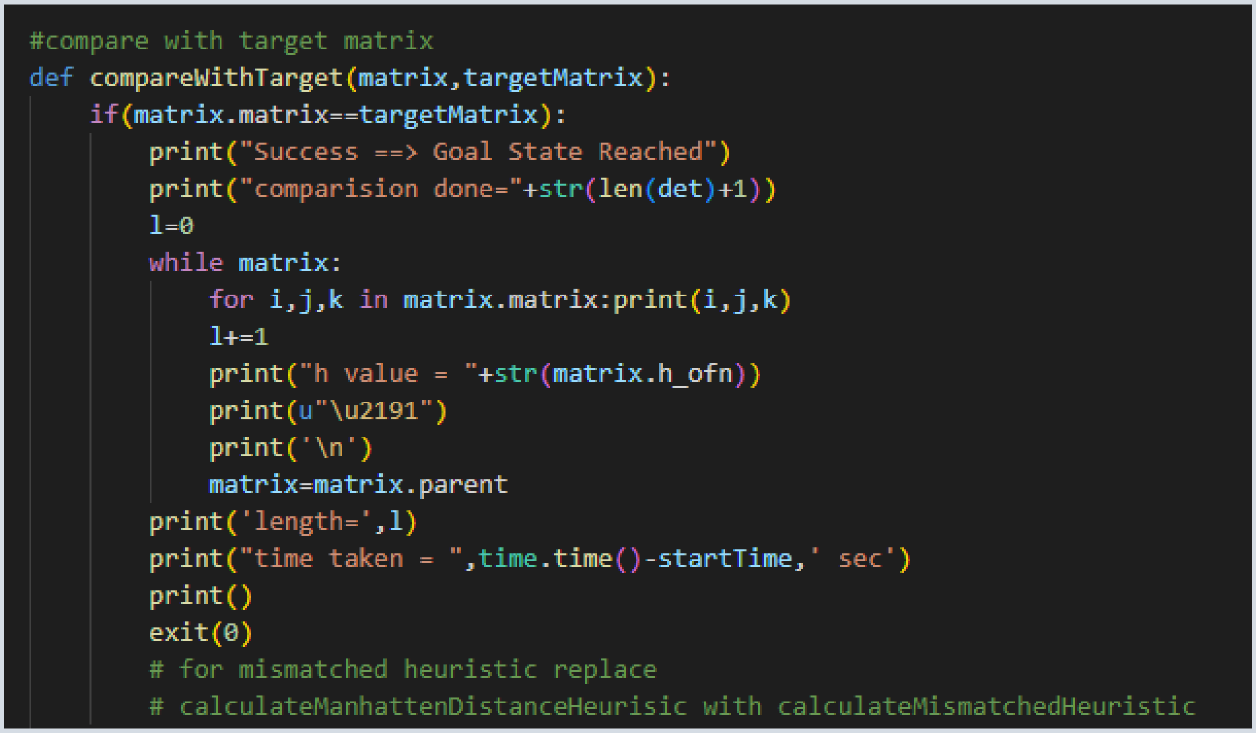

```python
#move down call
def moveDown(myMatrix):
    if((int(zeroRow) + 1)<3):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix.matrix)
        row = []
        row = tempMatrix[zeroRow + 1]
        data = row[zeroCol]
        tempMatrix[zeroRow + 1][zeroCol]=0
        tempMatrix[zeroRow][zeroCol] = data
        node=Node(tempMatrix)
        node.parent=myMatrix
        global treeQueue
        # replace  manhattenDistance, mismatched
        # for implementing different heuristics
        treeQueue.put((manhattenDistance(node),node))
    moveLeft(myMatrix)
```


```python
#move left call
def moveLeft(myMatrix):
    if((int(zeroCol) - 1)>-1):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix.matrix)
        data = tempMatrix[zeroRow][zeroCol-1]
        tempMatrix[zeroRow][zeroCol - 1]=0
        tempMatrix[zeroRow][zeroCol] = data
        global treeQueue
        node=Node(tempMatrix)
        node.parent=myMatrix
        # replace manhattenDistance, mismatched
        # for implementing different heuristics
        treeQueue.put((manhattenDistance(node),node))
    moveRight(myMatrix)
```

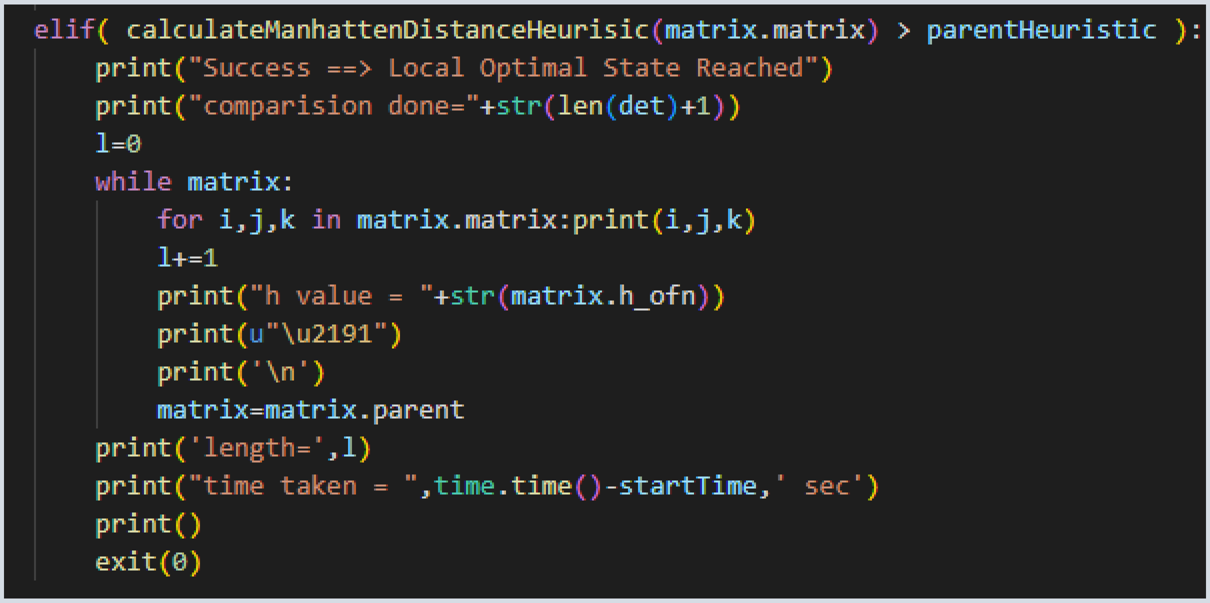


```
#move right call
def moveRight(myMatrix):
    if((int(zeroCol) + 1)<3):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix.matrix)
        data = tempMatrix[zeroRow][zeroCol+1]
        tempMatrix[zeroRow][zeroCol + 1]=0
        tempMatrix[zeroRow][zeroCol] = data
        global treeQueue
        node=Node(tempMatrix)
        node.parent=myMatrix
        # replace manhattenDistance, mismatched
        # for implementing different heuristics
        treeQueue.put((manhattenDistance(node),node))
```

We compare the matrix with the target matrix so as to check if the target matrix has been reached i.e the global maxima or if the local maxima is being reached

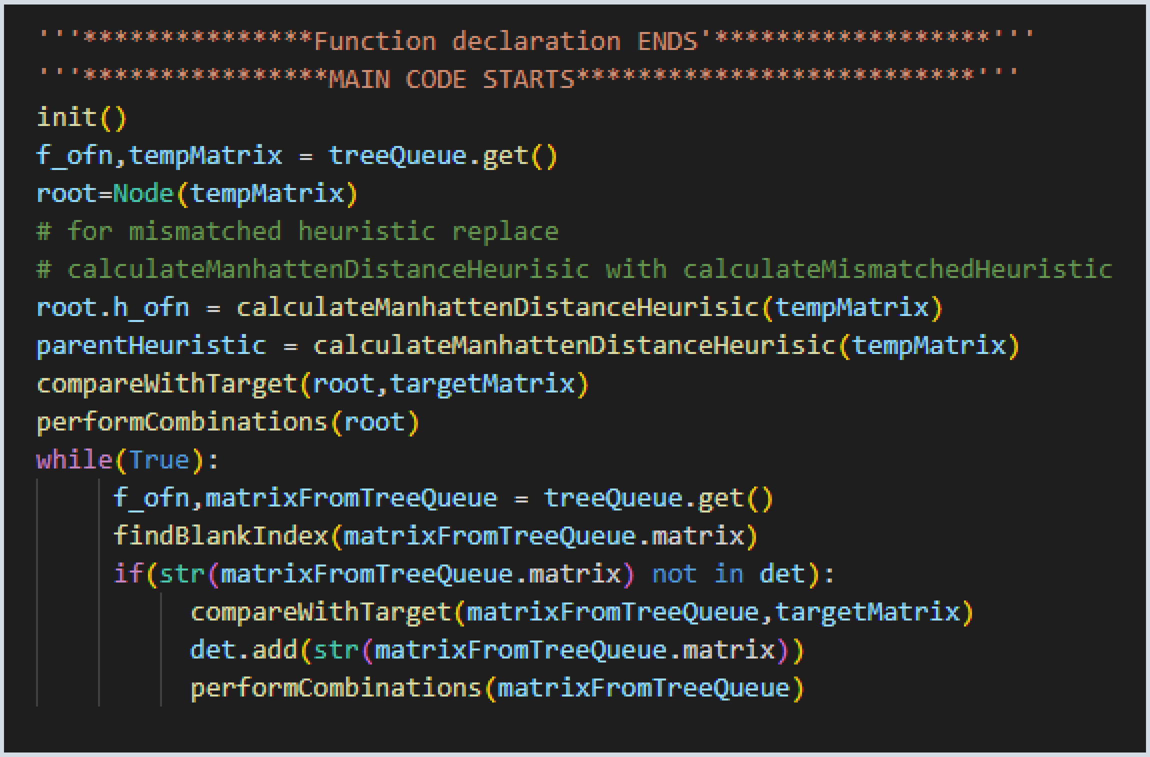


```
#compare with target matrix
def compareWithTarget(matrix,targetMatrix):
    if(matrix.matrix==targetMatrix):
        print("Success ==> Goal State Reached")
        print("comparision done="+str(len(det)+1))
        l=0
        while matrix:
            for i,j,k in matrix.matrix:print(i,j,k)
            l+=1
            print("h value = "+str(matrix.h_ofn))
            print(u"\u2191")
            print('\n')
            matrix=matrix.parent
        print('length=',l)
        print("time taken = ",time.time()-startTime,' sec')
        print()
        exit(0)
        # for mismatched heuristic replace
        # calculateManhattenDistanceHeurisic with calculateMismatchedHeuristic
```



```
elif( calculateManhattenDistanceHeurisic(matrix.matrix) > parentHeuristic ):
    print("Success ==> Local Optimal State Reached")
    print("comparision done="+str(len(det)+1))
    l=0
    while matrix:
        for i,j,k in matrix.matrix:print(i,j,k)
        l+=1
        print("h value = "+str(matrix.h_ofn))
        print(u"\u2191")
        print('\n')
        matrix=matrix.parent
    print('length=',l)
    print("time taken = ",time.time()-startTime,' sec')
    print()
    exit(0)
```

The main code execution calling all the respective functions starts here.



```
'''***************Function declaration ENDS'******************'''
'''****************MAIN CODE STARTS**************************'''
init()
f_ofn,tempMatrix = treeQueue.get()
root=Node(tempMatrix)
# for mismatched heuristic replace
# calculateManhattenDistanceHeurisic with calculateMismatchedHeuristic
root.h_ofn = calculateManhattenDistanceHeurisic(tempMatrix)
parentHeuristic = calculateManhattenDistanceHeurisic(tempMatrix)
compareWithTarget(root,targetMatrix)
performCombinations(root)
while(True):
    f_ofn,matrixFromTreeQueue = treeQueue.get()
    findBlankIndex(matrixFromTreeQueue.matrix)
    if(str(matrixFromTreeQueue.matrix) not in det):
        compareWithTarget(matrixFromTreeQueue,targetMatrix)
        det.add(str(matrixFromTreeQueue.matrix))
        performCombinations(matrixFromTreeQueue)
```

OUTPUT :

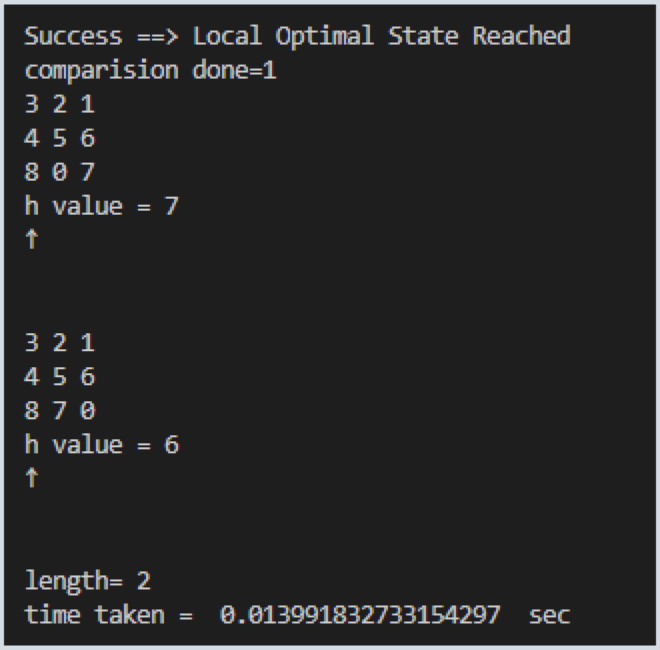
Input Matrix being considered:

3	2	1
4	5	6
8	7	0

Target Matrix :

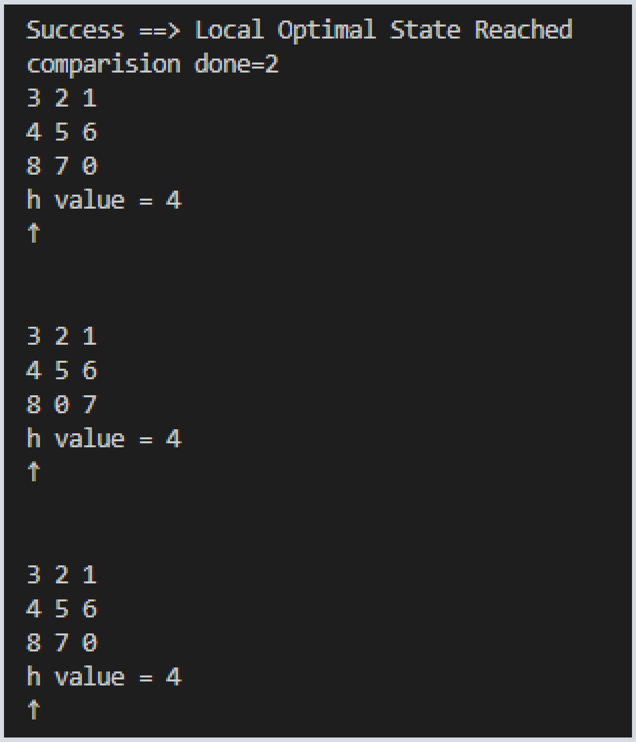
1	2	3
4	5	6
7	8	0

For Manhattan Distance:

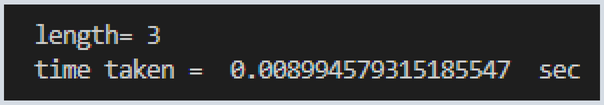


```
Success ==> Local Optimal State Reached
comparision done=1
3 2 1
4 5 6
8 0 7
h value = 7
↑


3 2 1
4 5 6
8 7 0
h value = 6
↑


length= 2
time taken =  0.013991832733154297  sec
```

For Mismatched tiles:



```
Success ==> Local Optimal State Reached
comparision done=2
3 2 1
4 5 6
8 7 0
h value = 4
↑


3 2 1
4 5 6
8 0 7
h value = 4
↑


3 2 1
4 5 6
8 7 0
h value = 4
↑
```



```
 length= 3
 time taken =  0.008994579315185547  sec
```