GROUP MEMBERS

● KUMAR SANATAN
ROLL NO – 2211AI24

● CHANDERA RAVI
ROLL NO – 2211AI16

CS 571
ARTIFICIAL INTELLIGENC LAB
ASSIGMENT 2: A* Search

INDIAN INSTITUTE OF TECHNOLOGY PATNA

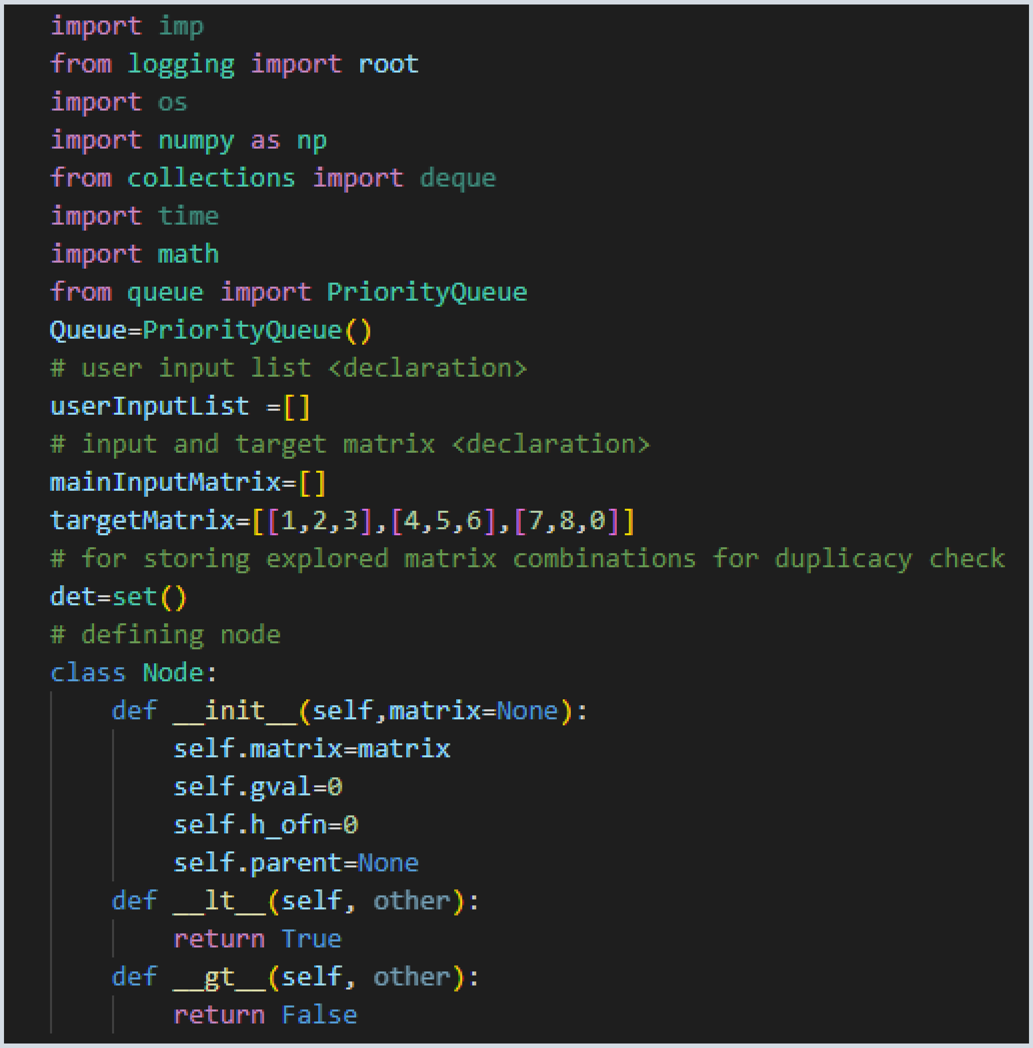
Date: 25th August 2022 **Deadline:** 04th September 2022

OBJECTIVE

Implement a search algorithm for solving the 8-puzzle problem with
respective assumptions.

CODE BASE :

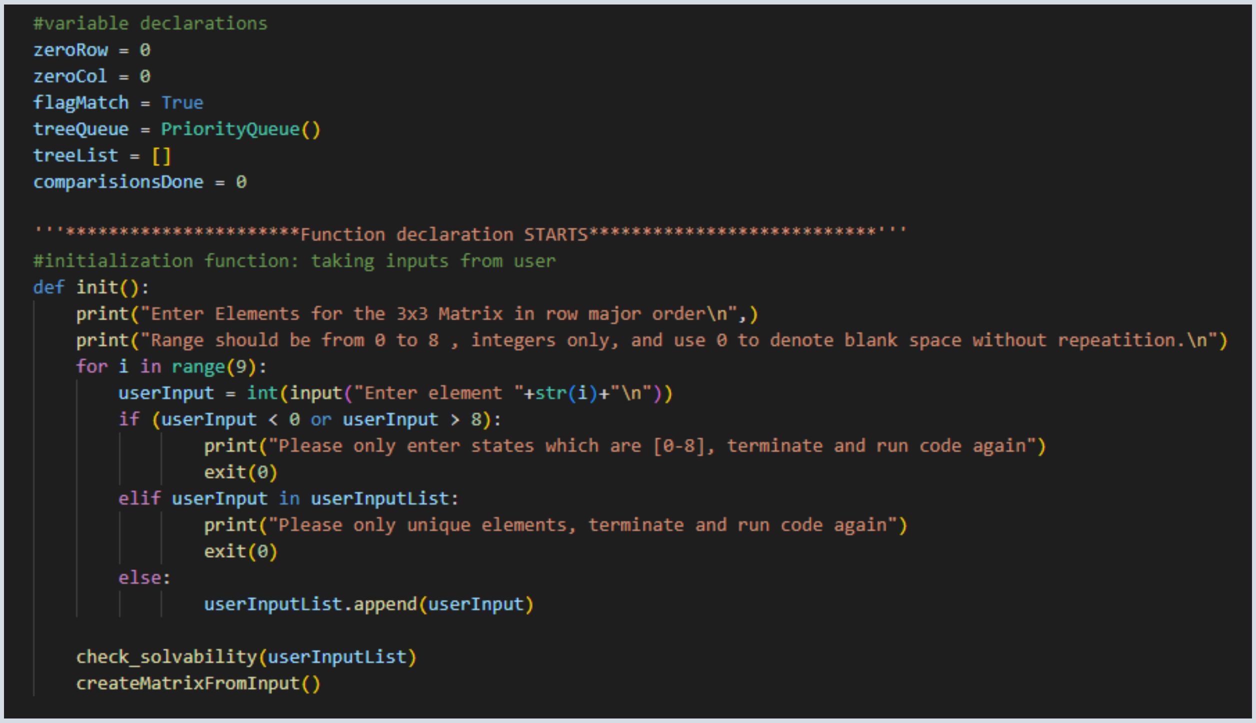
Importing libraries, defining target matrix and node that contains the matrix, g(n), h(n) and parent node.



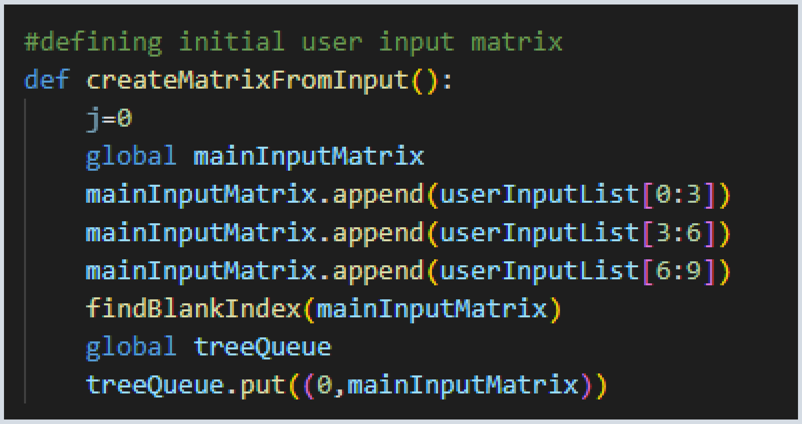
```
import imp
from logging import root
import os
import numpy as np
from collections import deque
import time
import math
from queue import PriorityQueue
Queue=PriorityQueue()
# user input list <declaration>
userInputList =[]
# input and target matrix <declaration>
mainInputMatrix=[]
targetMatrix=[[1,2,3],[4,5,6],[7,8,0]]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

We are defining variables and priority queue so as to store the matrix with respect to the corresponding priority based on the f(n) value (f(n)= g(n)+h(n)).

We are taking random input from the user in row major order and performing all the cross validtions and corner cases regaring the input fed by the user. Post that checking the solvability of the matrix and if true then proceeding towards matrix formation.

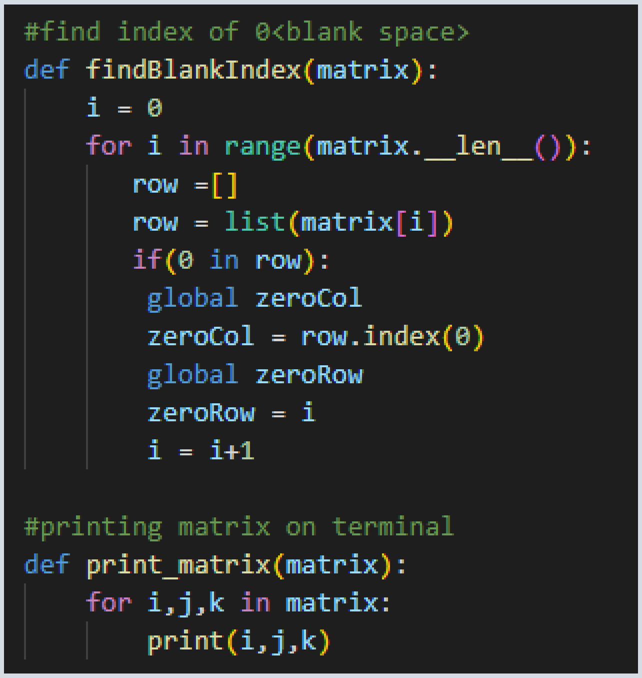


```python
#variable declarations
zeroRow = 0
zeroCol = 0
flagMatch = True
treeQueue = PriorityQueue()
treeList = []
comparisionsDone = 0

'''**********************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)

    check_solvability(userInputList)
    createMatrixFromInput()
```



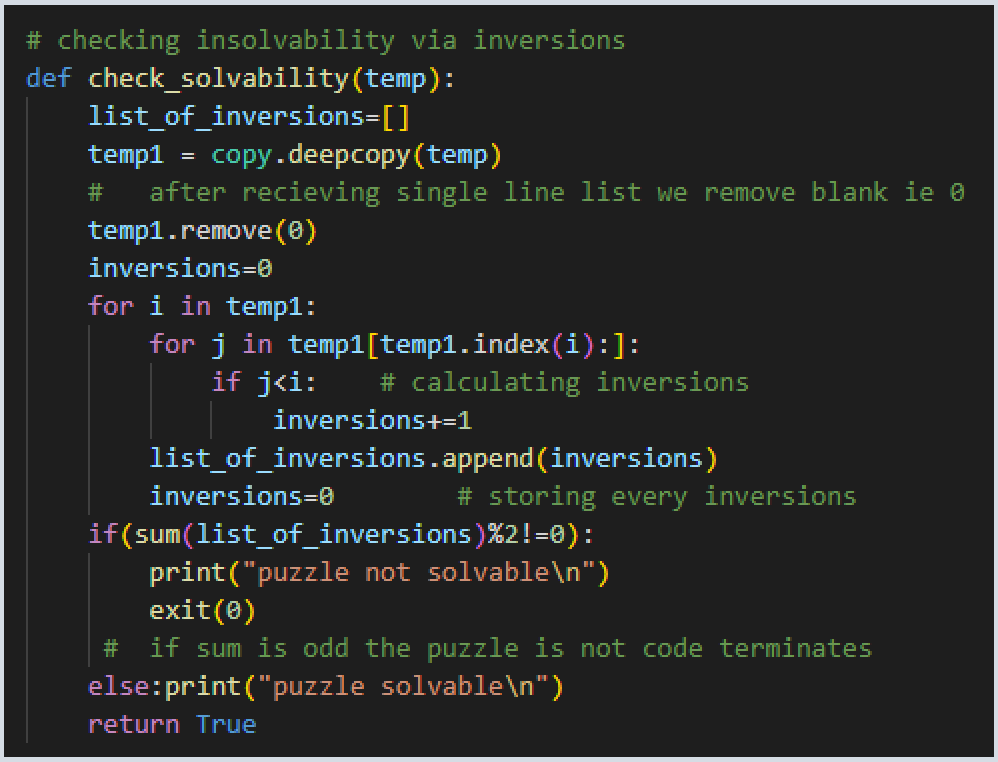
```python
#defining initial user input matrix
def createMatrixFromInput():
    j=0
    global mainInputMatrix
    mainInputMatrix.append(userInputList[0:3])
    mainInputMatrix.append(userInputList[3:6])
    mainInputMatrix.append(userInputList[6:9])
    findBlankIndex(mainInputMatrix)
    global treeQueue
    treeQueue.put((0,mainInputMatrix))
```

This function returns the index of the blank element (ie 0) so as to perform the shift moves.



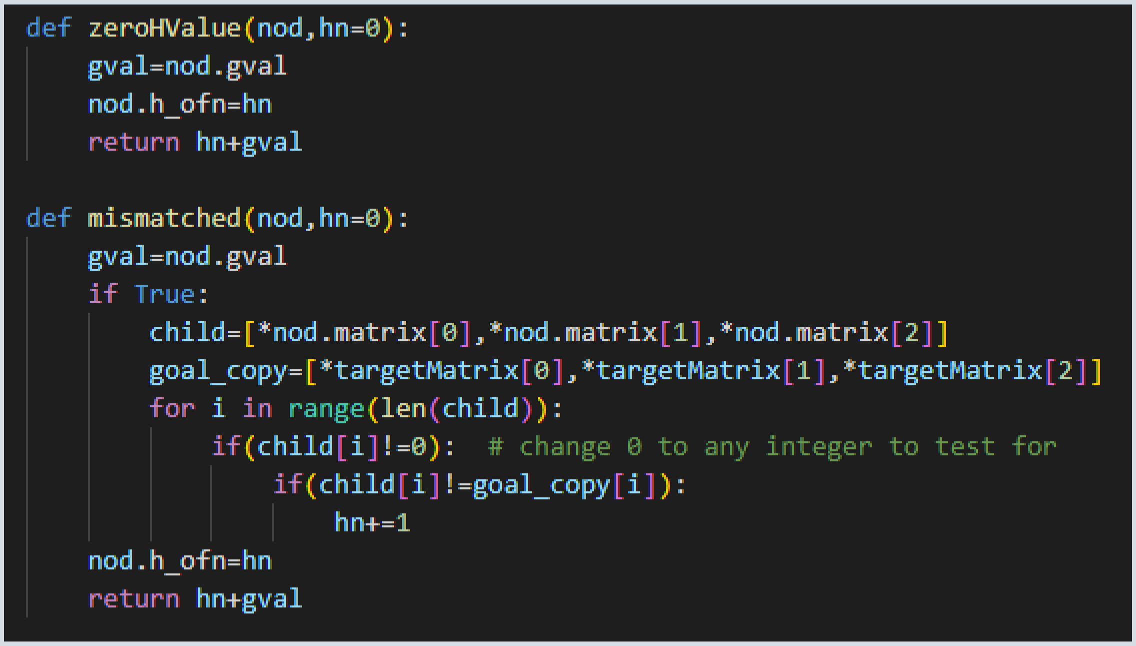
```
#find index of 0<blank space>
def findBlankIndex(matrix):
    i = 0
    for i in range(matrix.__len__()):
        row =[]
        row = list(matrix[i])
        if(0 in row):
         global zeroCol
         zeroCol = row.index(0)
         global zeroRow
         zeroRow = i
         i = i+1

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

check_solvability method incorporates inversions so as to check the for insolvability for a matrix thereby returning true if solvable or else exiting the code.

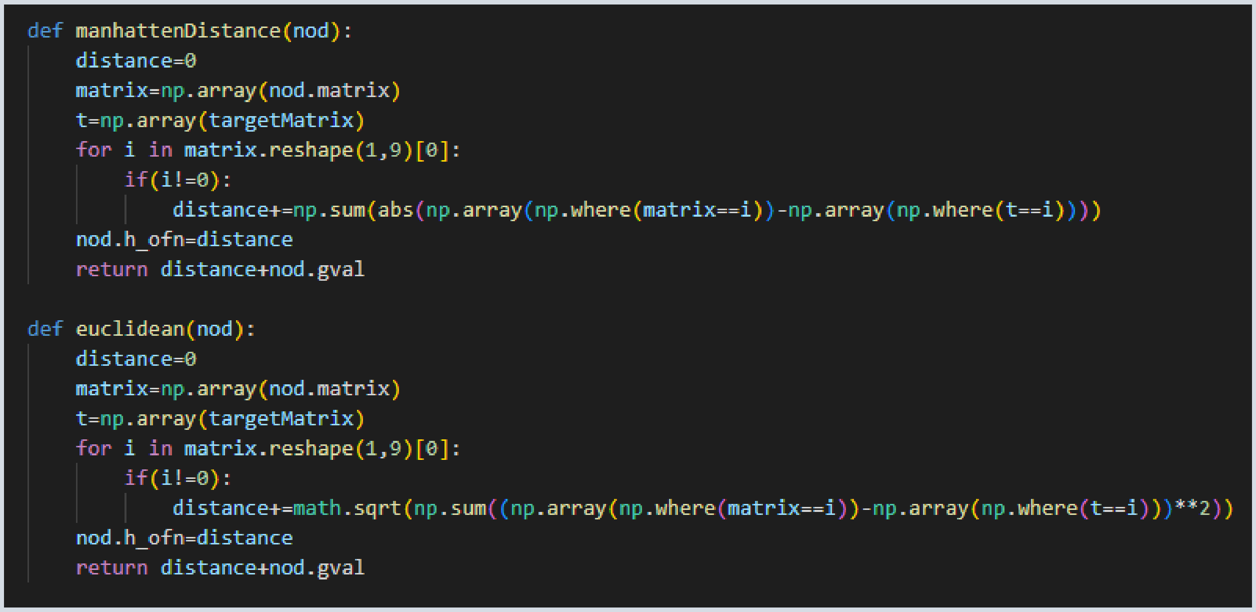


```
# checking insolvability via inversions
def check_solvability(temp):
    list_of_inversions=[]
    temp1 = copy.deepcopy(temp)
    #   after recieving single line list we remove blank ie 0
    temp1.remove(0)
    inversions=0
    for i in temp1:
        for j in temp1[temp1.index(i):]:
            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
```

We have defined functions zeroHvalue – to implement h1(n) = 0 case, mismatched – to implement mismatched tiles as h2(n) the heuristic, manhattenDistance h3(n) – to implement manhatten distacne as heuristic and euclidean – to implement euclidean and the h4(n) heuristic.

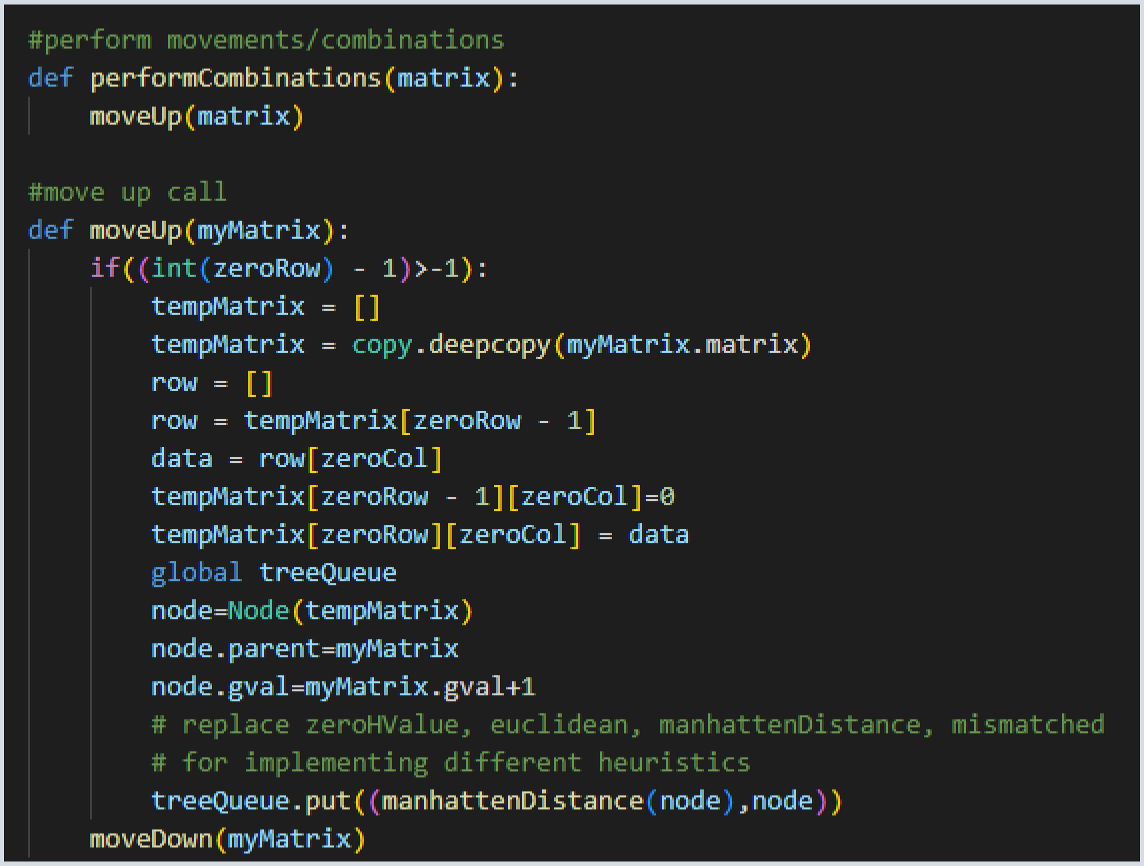


```python
def zeroHValue(nod,hn=0):
    gval=nod.gval
    nod.h_ofn=hn
    return hn+gval

def mismatched(nod,hn=0):
    gval=nod.gval
    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+gval
```

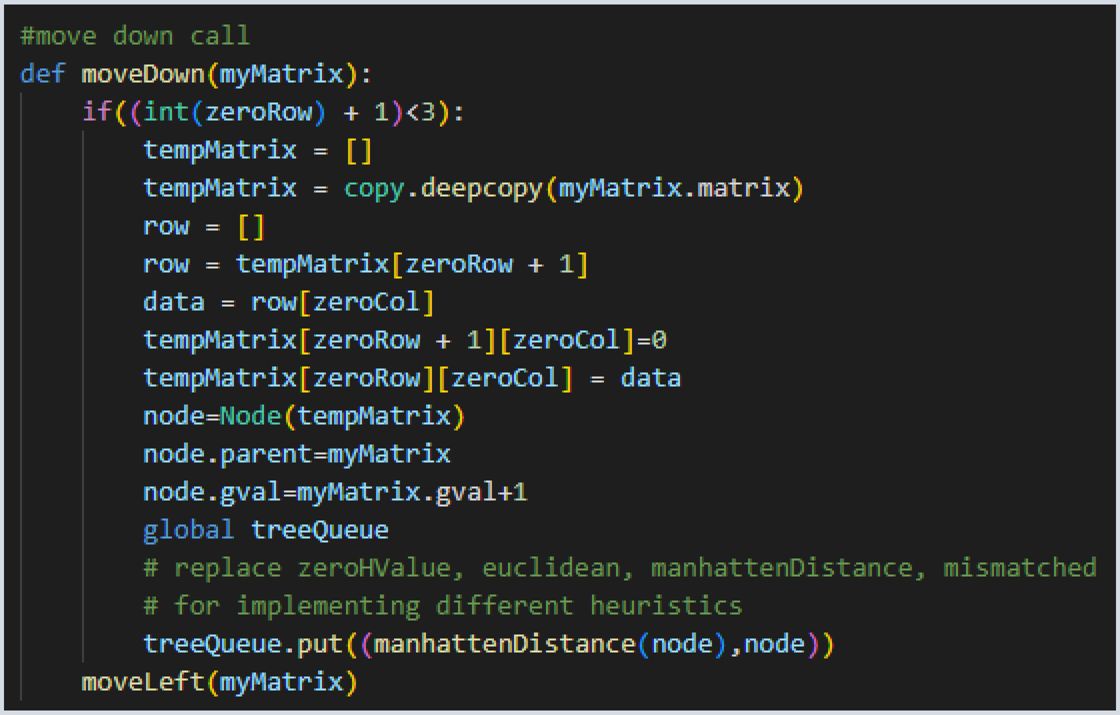


```python
def manhattenDistance(nod):
    distance=0
    matrix=np.array(nod.matrix)
    t=np.array(targetMatrix)
    for i in matrix.reshape(1,9)[0]:
        if(i!=0):
            distance+=np.sum(abs(np.array(np.where(matrix==i))-np.array(np.where(t==i))))
    nod.h_ofn=distance
    return distance+nod.gval

def euclidean(nod):
    distance=0
    matrix=np.array(nod.matrix)
    t=np.array(targetMatrix)
    for i in matrix.reshape(1,9)[0]:
        if(i!=0):
            distance+=math.sqrt(np.sum((np.array(np.where(matrix==i))-np.array(np.where(t==i)))**2))
    nod.h_ofn=distance
    return distance+nod.gval
```

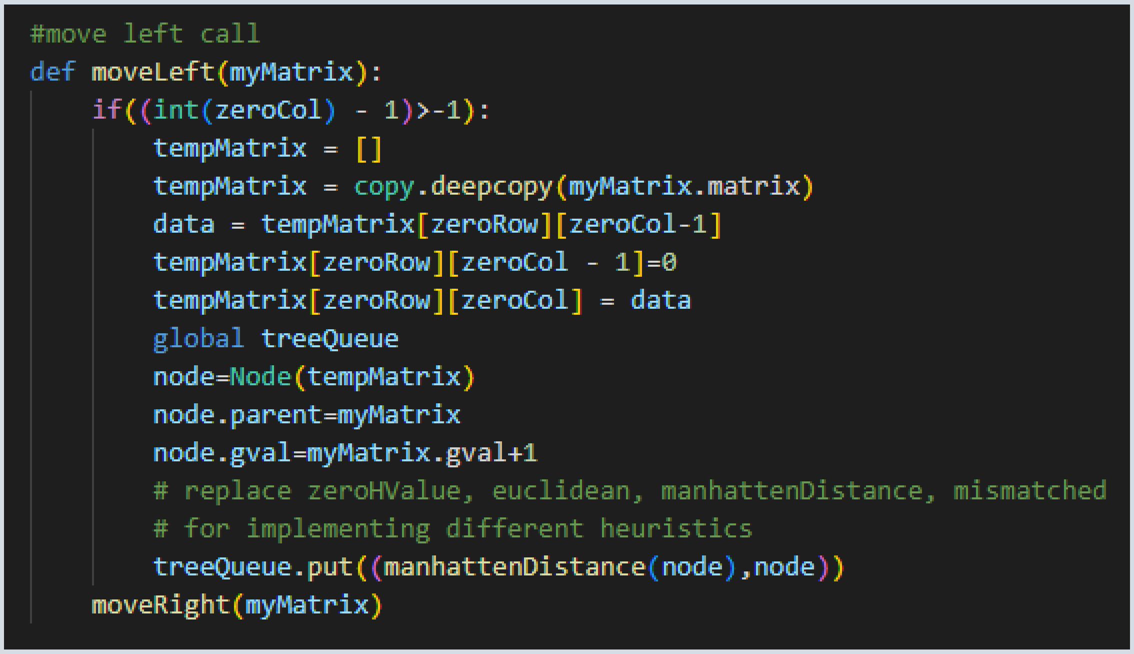
Here we have implemented the combinations of left, right, up and down movements of the blank space in either directions as feasible and putting it into the priority queue with priority being the objective function value that is sum of f(n) and g(n).



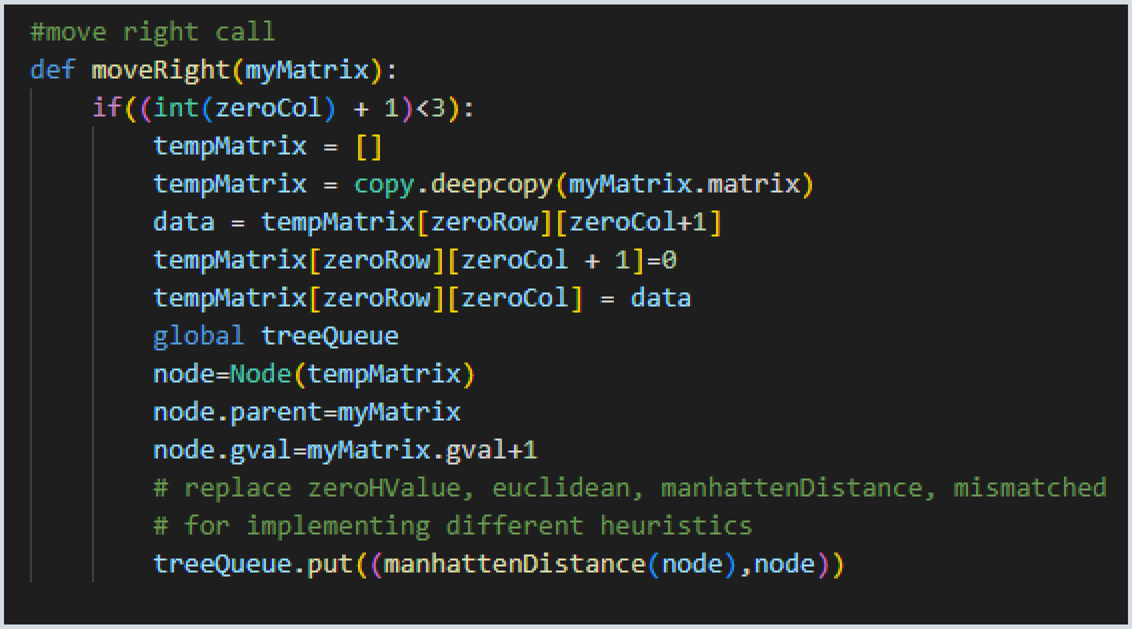
```
#perform movements/combinations
def performCombinations(matrix):
    moveUp(matrix)

#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
        node.gval=myMatrix.gval+1
        # replace zeroHValue, euclidean, 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
        node.gval=myMatrix.gval+1
        global treeQueue
        # replace zeroHValue, euclidean, manhattenDistance, mismatched
        # for implementing different heuristics
        treeQueue.put((manhattenDistance(node),node))
    moveLeft(myMatrix)
```

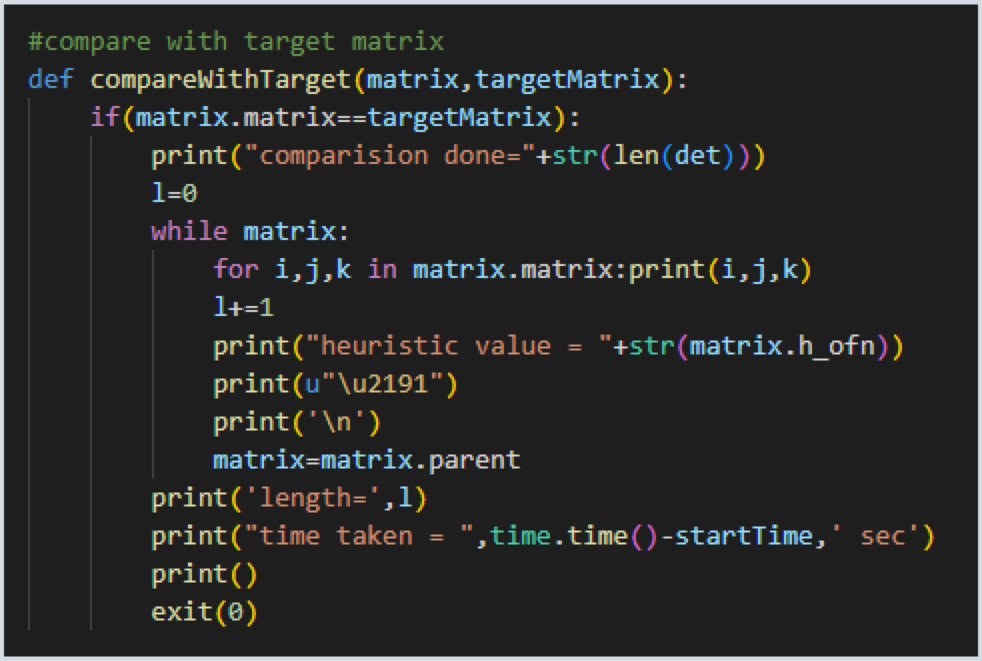


```python
#move left call
def moveLeft(myMatrix):
    if((int(zeroCol) - 1)>-1):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix.matrix)
        data = tempMatrix[zeroRow][zeroCol-1]
        tempMatrix[zeroRow][zeroCol - 1]=0
        tempMatrix[zeroRow][zeroCol] = data
        global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



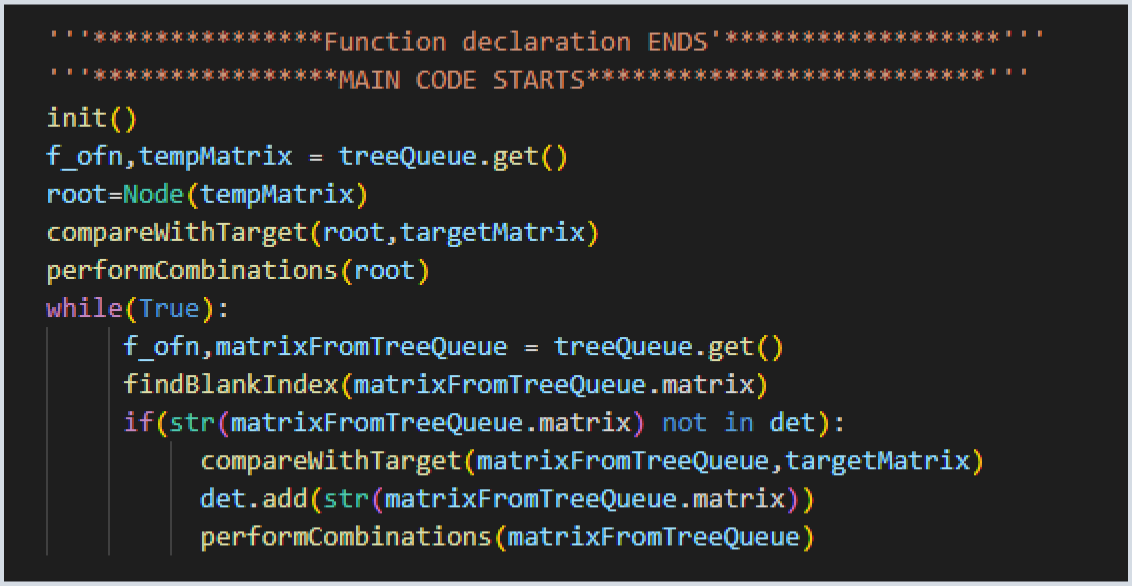
```python
#move right call
def moveRight(myMatrix):
    if((int(zeroCol) + 1)<3):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix.matrix)
        data = tempMatrix[zeroRow][zeroCol+1]
        tempMatrix[zeroRow][zeroCol + 1]=0
        tempMatrix[zeroRow][zeroCol] = data
        global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

Compare with target compares the matrix popped out with the target matrix and if matched with the parget prints back the trace from root to the target matrix thereby printing the entire path.



```python
#compare with target matrix
def compareWithTarget(matrix,targetMatrix):
    if(matrix.matrix==targetMatrix):
        print("comparision done="+str(len(det)))
        l=0
        while matrix:
            for i,j,k in matrix.matrix:print(i,j,k)
            l+=1
            print("heuristic 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)
```

Main function call starts here where the initial matrix is compared at first with the target matrix and if not matched then it is being stored to the priority queue.



```
'''***************Function declaration ENDS'******************'''
'''****************MAIN CODE STARTS**************************'''
init()
f_ofn,tempMatrix = treeQueue.get()
root=Node(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)
```

Results and Observations:

Input Matrix considered for observations and comparisons:

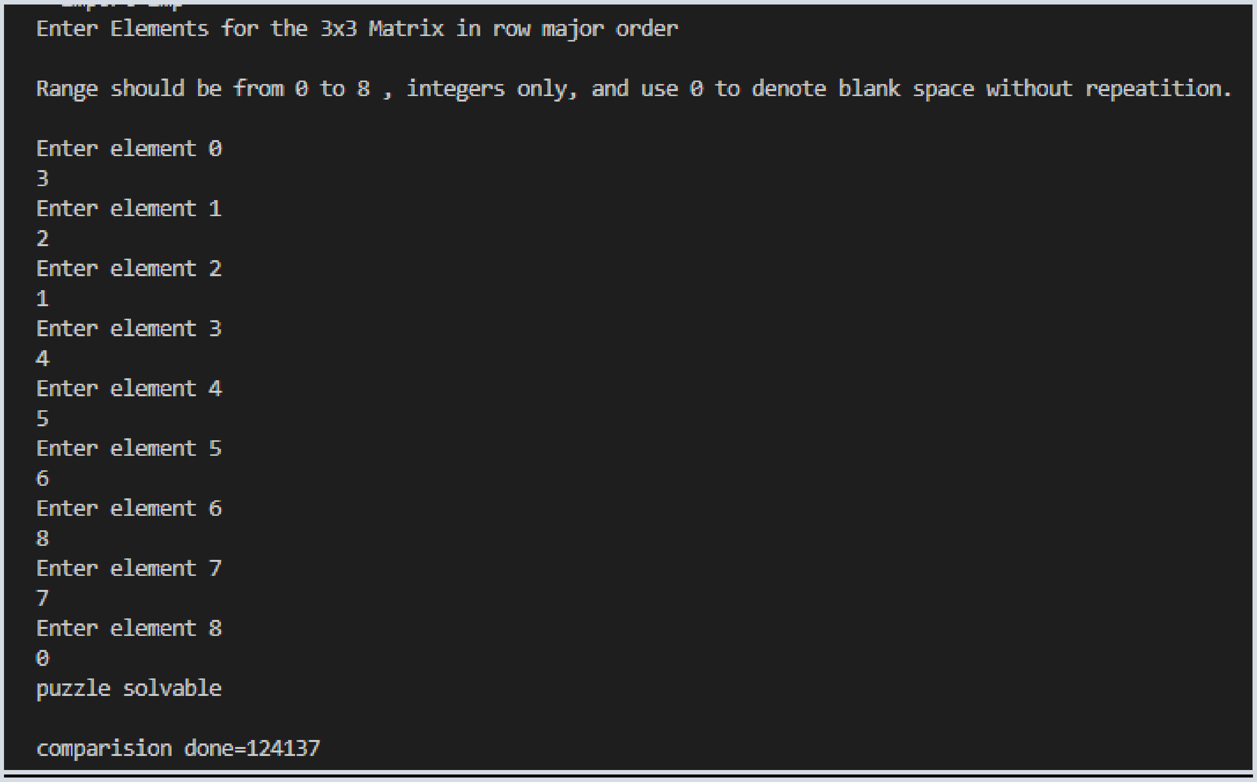
3	2	1
4	5	6
8	7	0

Target Matrix

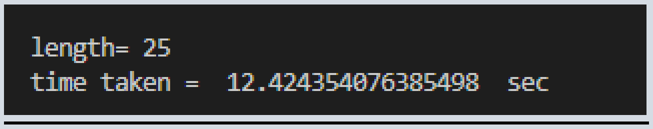
1	2	3
4	5	6
7	8	0

CODE Outputs:

For h1(n) = 0

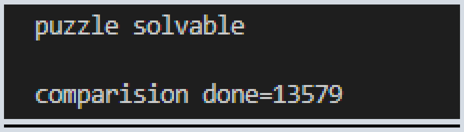


```
Enter Elements for the 3x3 Matrix in row major order

Range should be from 0 to 8 , integers only, and use 0 to denote blank space without repeatition.

Enter element 0
3
Enter element 1
2
Enter element 2
1
Enter element 3
4
Enter element 4
5
Enter element 5
6
Enter element 6
8
Enter element 7
7
Enter element 8
0
puzzle solvable

comparision done=124137
```

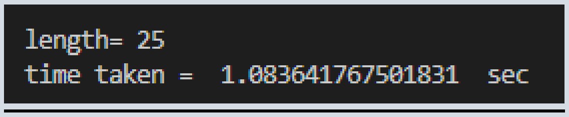


```
length= 25
time taken =  12.424354076385498  sec
```

For Displaced Tiles h2(n):

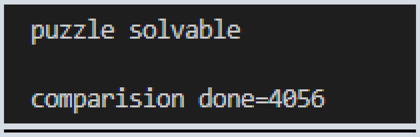


```
puzzle solvable

comparision done=13579
```

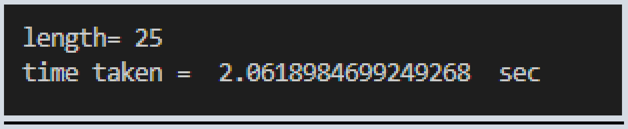


```
length= 25
time taken =  1.083641767501831  sec
```

For Manhattan Distance h3(n):

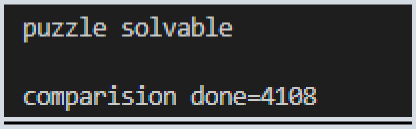


```
puzzle solvable

comparision done=4056
```

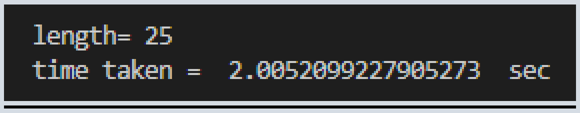


```
length= 25
time taken =  2.0618984699249268  sec
```

For Euclidean Distance h4(n):



```
puzzle solvable

comparision done=4108
```



```
length= 25
time taken =  2.0052099227905273  sec
```

Observations:

Parameter	**h1(n) = 0**	**Displaced Tiles h2(n)**	**Manhattan Distance h3(n)**	**Euclidean Distance h4(n)**
Time Taken	11.51 sec	1.109 sec	2.17 sec	2.04 sec
States Explored	124137	13579	4056	4108
Level Traversed	25	25	25	25

Points Observed:

- We have performed unreachability check via inversions as described above. If the sum is odd then the puzzle is unsolvable.
- From the above table we can verify that better the heuristic lesser is the states explored.
- All the states expanded by better heuristics is also expanded by the inferior heuristic for this we have printed the states explored in the code above.
- The monotone restriction being followed can be observed by the heuristic value printed alongside the matrix expanded.
- When we add the empty tile then this monotone restriction is violated that can be observed via the path traversed and the heuristic value printed alongside.