GROUP MEMBERS

KUMAR SANATAN
ROLL NO – 2211AI24

CHANDERA RAVI
ROLL NO – 2211AI16

CS 571
ARTIFICIAL INTELLIGENC LAB
ASSIGMENT 1: DFS, BFS

INDIAN INSTITUTE OF TECHNOLOGY
PATNA

Date: 08th August 2022 **Deadline:** 14th August 2022

OBJECTIVE

The task is to check if we can reach from any random start grid to the mentioned target grid by moving the Blank space ('B'). In one step, the Blank space can move either top or down or left or right.

CODE BASE (1) : KUMAR SANATAN - 2211AI24

Task : Generating any random input :

- For the above task we have taken random input from the user in row major order, with all checks and balances for range of inputs (from 0 to 8, 0 being the symbol for (‘B’) i.e blank space, also verifying any redundant data is not fed in this range by the user.)

1) Initialization and Variable declarations

(Defined the target matrix to be achieved as end result)

```
import copy
import os
from collections import deque
import time

#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()

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

2) Taking Inputs from user and matrix formation

```
'''**********************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)
    createMatrixFromInput()

#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.append(mainInputMatrix)
```

3) Function to find blank space index and printing matrix

```
#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)
```

4) Performing combinations (UP, DOWN, LEFT, RIGNT movements)

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

#move up call
def moveUp(myMatrix):
    if((int(zeroRow) - 1)>-1):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix)
        row = []
        row = tempMatrix[zeroRow - 1]
        data = row[zeroCol]
        tempMatrix[zeroRow - 1][zeroCol]=0
        tempMatrix[zeroRow][zeroCol] = data
        global treeQueue
        treeQueue.append(tempMatrix)
    moveDown(myMatrix)

#move down call
def moveDown(myMatrix):
    if((int(zeroRow) + 1)<3):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix)
        row = []
        row = tempMatrix[zeroRow + 1]
        data = row[zeroCol]
        tempMatrix[zeroRow + 1][zeroCol]=0
        tempMatrix[zeroRow][zeroCol] = data
        global treeQueue
        treeQueue.append(tempMatrix)
    moveLeft(myMatrix)
```

```
#move left call
def moveLeft(myMatrix):
    if((int(zeroCol) - 1)>-1):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix)
        data = tempMatrix[zeroRow][zeroCol-1]
        tempMatrix[zeroRow][zeroCol - 1]=0
        tempMatrix[zeroRow][zeroCol] = data
        global treeQueue
        treeQueue.append(tempMatrix)
    moveRight(myMatrix)

#move right call
def moveRight(myMatrix):
    if((int(zeroCol) + 1)<3):
        tempMatrix = []
        tempMatrix = copy.deepcopy(myMatrix)
        data = tempMatrix[zeroRow][zeroCol+1]
        tempMatrix[zeroRow][zeroCol + 1]=0
        tempMatrix[zeroRow][zeroCol] = data
        global treeQueue
        treeQueue.append(tempMatrix)
```

5) Comparing with target matrix

```
#compare with target matrix
def compareWithTarget(matrix,targetMatrix):
    global comparisionsDone
    comparisionsDone  = comparisionsDone + 1
    if(matrix==targetMatrix):
        print("comparision done="+str(comparisionsDone))
        print(len(det))
        exit(0)

'''***************Function declaration ENDS'*****************'''
```

6) Main code execution call for BFS

```
#main code execution BFS
init()
tempMatrix = treeQueue.popleft()
compareWithTarget(tempMatrix,targetMatrix)
performCombinations(tempMatrix)
while(True):
    matrixFromTreeQueue = treeQueue.popleft()
    findBlankIndex(matrixFromTreeQueue)
    if(str(matrixFromTreeQueue) not in det):
        compareWithTarget(matrixFromTreeQueue,targetMatrix)
        det.add(str(matrixFromTreeQueue))
        performCombinations(matrixFromTreeQueue)
```

Corresponding Output

```
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python310/python.exe c:/Users/HP/Desktop/PYTHON_LEARN/aiLabAssignment.py
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
comparision done=122269
122267
```

7) Main code execution call for DFS

```
#main code for DFS
init()
tempMatrix = treeQueue.pop()
compareWithTarget(tempMatrix,targetMatrix)
performCombinations(tempMatrix)
while(True):
    matrixFromTreeQueue = treeQueue.pop()
    findBlankIndex(matrixFromTreeQueue)
    if(str(matrixFromTreeQueue) not in det):
        compareWithTarget(matrixFromTreeQueue,targetMatrix)
        det.add(str(matrixFromTreeQueue))
        performCombinations(matrixFromTreeQueue)
```

Corresponding Output

```
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python310/python.exe c:/Users/HP/Desktop/PYTHON_LEARN/aiLabAssignment.py
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
comparision done=72986
72984
```

Code Overview:

- We are taking input from user in row major order element by element to preserve the randomness of the input as per prescribed in userInputList[].
- We are using det (set) to store the matrixes/combinations that gets popped out from the queue/stack to cross verify the duplicity while performing different further combinations(L,R,U,D movement of the blank space).
- treeQueue(a deque) is used to work as a stack/queue for DFS/BFS.
- findBlankIndex is used to find the index at which blank(0) is present to carry out further combinations of shifts.
- In the main code run we are storing the different combinations of matrices in queue and popping one at a time, comparing it with target matrix and if result matches we are exiting the code by exit(0) call and printing the comparisons performed. If it does not match then we push the different combinations of the matrix into the queue by cross checking if the same combination has been used before via the det (set).

For the given input as in assignment the number of comparisons are printed in the output as attached above after each BFS and DFS algorithm.

CODE BASE (2) : CHANDERA RAVI - 2211AI16

STEP1: Our code is asking users to enter the matrix as per their choice. So, we defined a function to take user input.

```
def matrix(m,n):
    print("Elements should be from 0 to 8 , integers only, and without repeatition.")
    print(" use 0 to denote blank space")
    user_input_matrix = [ ]
    for i in range(m):
        row = [ ]
        for j in range(n):
            element = int(input(f' enter element [{i} {j}] of your matrix \n'))
            if (element < 0 or element > 8):
                print("Only [0-8] are allowed, run code again")
                exit(0)
            elif element in row:
                print("Repeating elements are allowed, run code again")
                exit(0)
            else:
                row.append(element)
        user_input_matrix.append(row)
    return user_input_matrix
```

STEP2: Defining functions to perform Upward, Downward, left, and right operations on matrix.

```
def upword_move(randomA): # defined funtion to do upward operation
    for i in range(3):
        for j in range(3):
            if randomA[i][j] == 0:
                if i!=0: # checking whether operation is possible or not
                    randomA[i][j]= randomA[i-1][j]
                    randomA[i - 1][j]=0
                    return randomA
```

```
def downward_move(randomB): # defined funtion to do downaward operation
    for i in range(3):
        for j in range(3):
            if randomB[i][j] == 0:
                if i!=2:# checking whether operation is possible or not
                    randomB[i][j]= randomB[i+1][j]
                    randomB[i + 1][j]=0
                    return randomB
```

```
def right_move(randomC): # defined funtion to do right operation
    for i in range(3):
        for j in range(3):
            if randomC[i][j] == 0:
                if j!=2:# checking whether operation is possible or not
                    randomC[i][j]= randomC[i][j+1]
                    randomC[i][j+1]=0
                    return randomC
```

```
def left_move(randomD): # defined funtion to do left operation
    for i in range(3):
        for j in range(3):
            if randomD[i][j] == 0:
                if j!=0:# checking whether operation is possible or not
                    randomD[i][j]= randomD[i][j-1]
                    randomD[i ][j-1]=0
                    return randomD
```

STEP3: We generated a list to store all the possible matrix output we are getting by performing operations on it.

```
all_matrix_bfs = [ ] # made a list to store all the matrix
all_matrix_bfs.append(user_input_matrix_bfs)
avoid_repetition_bfs = set() # defined set to avoid repetition
start_time_bfs = time.time() # for calculating time to run code
```

STEP4: Using a while loop to ensure that loop runs till, we get our goal.

```
while user_input_matrix_bfs!=destination_matrix:
    A = upword_move(copy.deepcopy(user_input_matrix_bfs))
    if(A!=None):
        if str(A) not in avoid_repetition_bfs:
            all_matrix_bfs.append(A)
            avoid_repetition_bfs.add(str(A))
    B = downward_move(copy.deepcopy(user_input_matrix_bfs))
    if(B!=None):
        if str(B) not in avoid_repetition_bfs:
            all_matrix_bfs.append(B)
            avoid_repetition_bfs.add(str(B))
    C = right_move(copy.deepcopy(user_input_matrix_bfs))
    if(C!=None):
        if str(C) not in avoid_repetition_bfs:
            all_matrix_bfs.append(C)
            avoid_repetition_bfs.add(str(C))
    D = left_move(copy.deepcopy(user_input_matrix_bfs))
    if(D!=None):
        if str(D) not in avoid_repetition_bfs:
            all_matrix_bfs.append(D)
            avoid_repetition_bfs.add(str(D))
```

STEP5: we are using a set to store the popped matrix from our list so when we have performed all the possible operations on one matrix then we can check whether our matrix is the same as earlier or it is new. So, this way we are avoiding unnecessary repetition which may slow our program and increases unwanted comparisons.

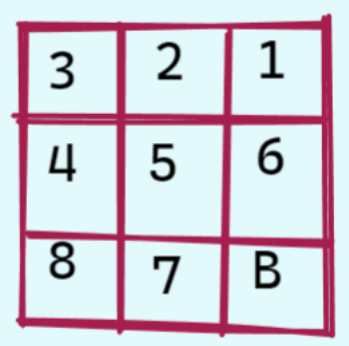
We are also using the time( ) function to track the running time of an algorithm.

```
    avoid_repetition_bfs.add(str(user_input_matrix_bfs))
    user_input_matrix_bfs = all_matrix_bfs.pop(0)
    total_comparision_bfs = total_comparision_bfs+1
end_time_bfs = time.time()
code_run_bfs = int(end_time_bfs - start_time_bfs)
```

STEP6: We are using the same method to write a program for DFS but we pop( ) the matrix from the end of the list whereas in BFS we were removing the matrix from the start of the list.

```
    avoid_repetition_dfs.add(str(user_input_matrix_dfs))
    user_input_matrix_dfs = all_matrix_dfs.pop(-1)
    total_comparision_dfs = total_comparision_dfs+1
```

STEP7: We are running code for a matrix which is given in the assignment as an example.



```
 use 0 to denote blank space
 enter element [0 0] of your matrix
3
 enter element [0 1] of your matrix
2
 enter element [0 2] of your matrix
1
 enter element [1 0] of your matrix
4
 enter element [1 1] of your matrix
5
 enter element [1 2] of your matrix
6
 enter element [2 0] of your matrix
8
 enter element [2 1] of your matrix
7
 enter element [2 2] of your matrix
0
User entered [[3, 2, 1], [4, 5, 6], [8, 7, 0]] matrix
please wait code is running
DFS gave our goal matrix    [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
BFS gave our goal matrix    [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
DFS had done total of  76153 comparisions
BFS had done total of  122563 comparisions
DFS took    3  seconds only to get our goal
BFS took    5  seconds only to get our goal
 Hence DFS is more efficient compared to BFS for given user input
PS C:\Users\Ravi\Documents\python-practice>
```

Conclusion:

For Code base (1) DFS took 72986 comparisons for the given input whereas BFS took 122269 comparisons for the same input to reach the desired output as prescribed.

For Code base (2) DFS had done a total of 76153 comparisons to reach the goal matrix. BFS had done a total of 122563 comparisons to reach the goal matrix.

DFS took 3 seconds Whereas BFS took 5 seconds for the same matrix.

Hence DFS is more efficient compared to BFS for given user input.

DFS is more efficient when the desired result is far away from the root node or the starting point of traversal whereas BFS is more viable for result which lies closer to the starting node of traversal as BFS computes level by level and DFS traverses sub tress from the starting point.

Example: **INPUT:**

If we had to search for node 3 which is closer to the starting node 1(assuming we started traversal from 1) then for BFS 1->2->3 and the goal will be reached whereas for DFS 1->2->4->3 then we will reach our goal thus node which is closer to the starting node for that BFS would serve as a better algorithm. Similarly if we searched for node 4 then DFS would serve better as it is away from the starting node due to the intuition as described above.