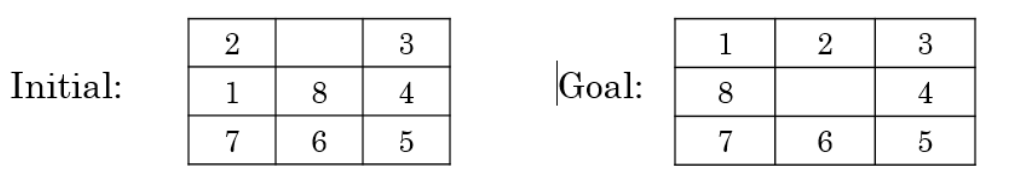
**Computer Science and Engineering Department**

**Artificial Intelligence (UCS-521)**

**Lab Assignment-3**

**Note**: As a data scientist, you have been assigned a job to solve the 8 puzzle problem. To generate the states of the search space, you need to define the rules/operators properly. As a solution, you need to print the intermediate steps of the solution as well as total number of moves used to achieve the goal state.

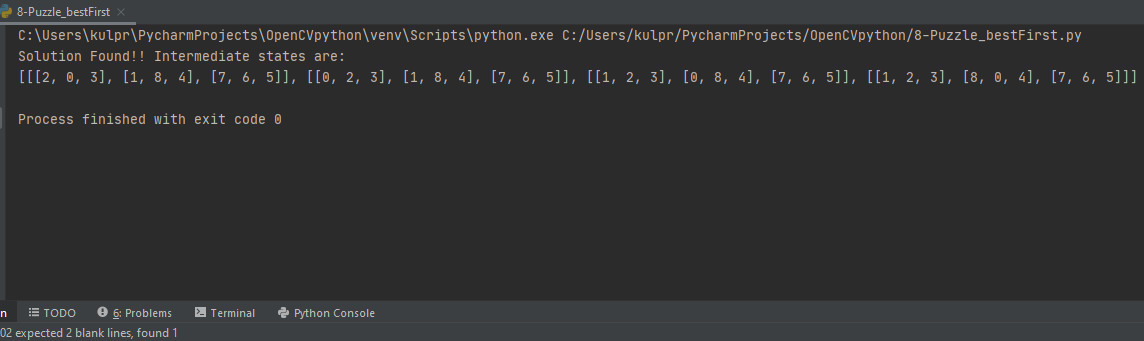
1. If the initial and final states are as below and H(n): number of misplaced tiles in the current state n as compared to the goal node need to be considered as the heuristic function. You need to use **Best First Search** algorithm.



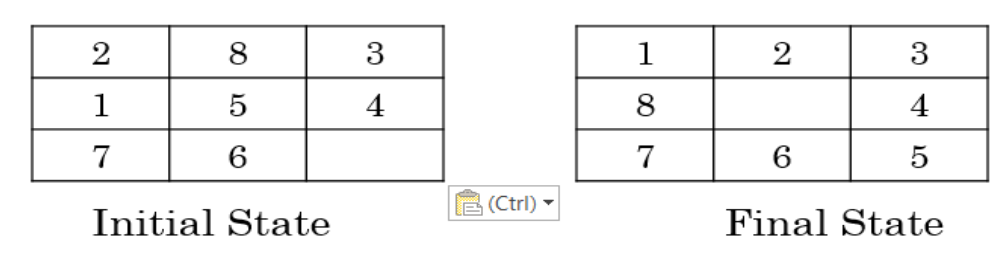
**CODE:**

import copy  
  
initial\_arr = [[2,0,3],[1,8,4],[7,6,5]]  
final\_arr = [[1,2,3],[8,0,4],[7,6,5]]  
  
#All possible moves  
# up = (-1,0)  
# down = (1,0)  
# left = (0,-1)  
# right = (0,1)  
  
moves = [(-1,0),(1,0),(0,-1),(0,1)]  
movesName = ['UP', 'DOWN', 'LEFT', 'RIGHT']  
  
#checking valid moves  
def isValidMove(initial\_arr, idx, move):  
 i = idx[0] + move[0]  
 j = idx[1] + move[1]  
 if i<len(initial\_arr) and i>=0 and j>=0 and j<len(initial\_arr):  
 return True  
 return False  
  
def performMove(initial\_arr, idx, move):  
 i = idx[0] + move[0]  
 j = idx[1] + move[1]  
 temp\_arr = copy.deepcopy(initial\_arr)  
 temp = temp\_arr[i][j]  
 temp\_arr[i][j] = temp\_arr[idx[0]][idx[1]]  
 temp\_arr[idx[0]][idx[1]] = temp  
 return temp\_arr  
  
def findZeroIndex(initial\_arr):  
 for i in range(0,len(initial\_arr)):  
 for j in range(0,len(initial\_arr[i])):  
 if initial\_arr[i][j] == 0:  
 return i,j  
  
def enqueue(s,val):  
 global q  
 q = q + [(val,s)]  
  
  
def dequeue():  
 global q  
 global visited  
  
 q.sort()  
 visited = visited + [q[0][1]]  
  
 temp = q[0][1]  
 del q[0]  
 return (temp)  
  
def heuristic(initial\_arr, final\_arr):  
 count = 0  
 for p in range(3):  
 for q in range(3):  
 if initial\_arr[p][q]!=0:  
 if initial\_arr[p][q]!=final\_arr[p][q]:  
 count = count+1  
 return count  
  
def findSol(initial\_arr, final\_arr):  
 global visited  
 global q  
 enqueue(initial\_arr, heuristic(initial\_arr, final\_arr))  
 if initial\_arr == final\_arr:  
 return  
 while True:  
 if len(q)>0:  
 curr\_state = dequeue()  
 else:  
 print('Not Found')  
 return  
 idx = findZeroIndex(curr\_state)  
 for move in moves:  
 if isValidMove(curr\_state, idx, move):  
 new\_arr = performMove(curr\_state, idx, move)  
 if new\_arr == final\_arr:  
 print('Solution Found!! Intermediate states are:')  
 print(visited+[new\_arr])  
 return  
 if new\_arr not in visited:  
 h = heuristic(new\_arr,final\_arr)  
 enqueue(new\_arr, h)  
  
  
def main():  
 global q  
 global visited  
 visited = []  
 q=[]  
 findSol(initial\_arr, final\_arr)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

**OUTPUT:**



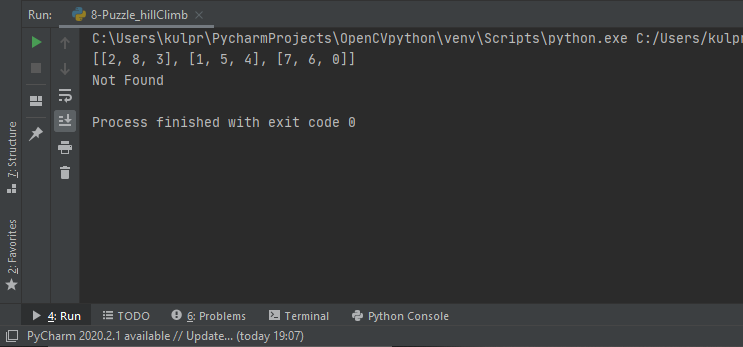
1. If the initial and final states have been changed as below and approach you need to use is **Hill Climbing searching algorithm**. H(n): number of misplaced tiles in the current state n as compared to the goal node as the heuristic function for the following states.



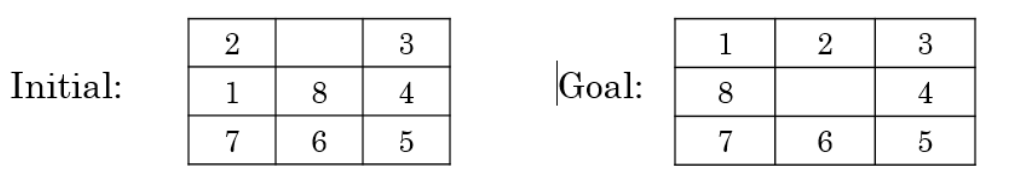
**CODE:**

import copy  
  
initial\_arr = [[2,8,3],[1,5,4],[7,6,0]]  
final\_arr = [[1,2,3],[8,0,4],[7,6,5]]  
  
#All possible moves  
# up = (-1,0)  
# down = (1,0)  
# left = (0,-1)  
# right = (0,1)  
  
moves = [(-1,0),(1,0),(0,-1),(0,1)]  
movesName = ['UP', 'DOWN', 'LEFT', 'RIGHT']  
  
#checking valid moves  
def isValidMove(initial\_arr, idx, move):  
 i = idx[0] + move[0]  
 j = idx[1] + move[1]  
 if i<len(initial\_arr) and i>=0 and j>=0 and j<len(initial\_arr):  
 return True  
 return False  
  
def performMove(initial\_arr, idx, move):  
 i = idx[0] + move[0]  
 j = idx[1] + move[1]  
 temp\_arr = copy.deepcopy(initial\_arr)  
 temp = temp\_arr[i][j]  
 temp\_arr[i][j] = temp\_arr[idx[0]][idx[1]]  
 temp\_arr[idx[0]][idx[1]] = temp  
 return temp\_arr  
  
def findZeroIndex(initial\_arr):  
 for i in range(0,len(initial\_arr)):  
 for j in range(0,len(initial\_arr[i])):  
 if initial\_arr[i][j] == 0:  
 return i,j  
  
  
def heuristic(initial\_arr, final\_arr):  
 count = 0  
 for p in range(3):  
 for q in range(3):  
 if initial\_arr[p][q]!=0:  
 if initial\_arr[p][q]!=final\_arr[p][q]:  
 count = count+1  
 return count  
  
def findSol(initial\_arr, final\_arr):  
 global visited  
 print(initial\_arr)  
 visited.append(initial\_arr)  
 H = heuristic(initial\_arr, final\_arr)  
 if H == 0:  
 return 0  
 best\_h = heuristic(initial\_arr, final\_arr)  
 best\_arr = initial\_arr  
 bestMove = (0,0)  
 idx = findZeroIndex(initial\_arr)  
 flag = 0  
 for move in moves:  
 if isValidMove(initial\_arr, idx, move):  
 new\_arr = performMove(initial\_arr, idx, move)  
 if new\_arr not in visited:  
 h = heuristic(new\_arr,final\_arr)  
 if h < best\_h:  
 flag = 1;  
 best\_h = h  
 bestMove = move  
 best\_arr = new\_arr  
  
 if flag == 0:  
 print('Not Found')  
 return -1000000  
 ind = moves.index(bestMove)  
 print(f'Move = {movesName[ind]}. best\_h = {best\_h}')  
 return findSol(best\_arr, final\_arr) + 1  
  
  
def main():  
 global visited  
 visited = []  
 noOfMoves = findSol(initial\_arr, final\_arr)  
 if noOfMoves >= 0:  
 print(f'Goal State!! Number of moves required = {noOfMoves}')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

**OUTPUT:**



1. Apply **A\* searching algorithm** by taking H(n): number of correctly placed tiles in the current state n as compared to the goal node. as the heuristic function.

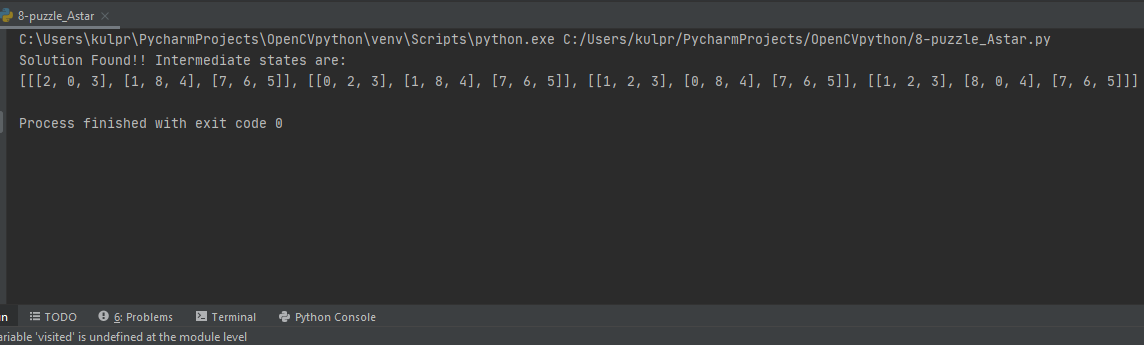


**CODE:**

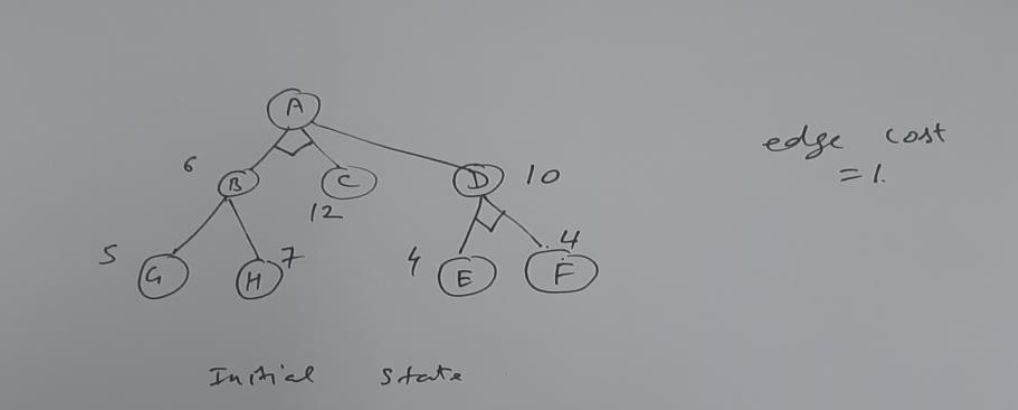
import copy  
  
initial\_arr = [[2,0,3],[1,8,4],[7,6,5]]  
final\_arr = [[1,2,3],[8,0,4],[7,6,5]]  
  
#All possible moves  
# up = (-1,0)  
# down = (1,0)  
# left = (0,-1)  
# right = (0,1)  
  
moves = [(-1,0),(1,0),(0,-1),(0,1)]  
movesName = ['UP', 'DOWN', 'LEFT', 'RIGHT']  
  
#checking valid moves  
def isValidMove(initial\_arr, idx, move):  
 i = idx[0] + move[0]  
 j = idx[1] + move[1]  
 if i<len(initial\_arr) and i>=0 and j>=0 and j<len(initial\_arr):  
 return True  
 return False  
  
def performMove(initial\_arr, idx, move):  
 i = idx[0] + move[0]  
 j = idx[1] + move[1]  
 temp\_arr = copy.deepcopy(initial\_arr)  
 temp = temp\_arr[i][j]  
 temp\_arr[i][j] = temp\_arr[idx[0]][idx[1]]  
 temp\_arr[idx[0]][idx[1]] = temp  
 return temp\_arr  
  
def findZeroIndex(initial\_arr):  
 for i in range(0,len(initial\_arr)):  
 for j in range(0,len(initial\_arr[i])):  
 if initial\_arr[i][j] == 0:  
 return i,j  
  
def enqueue(s,val):  
 global q  
 q = q + [(val,s)]  
  
  
def dequeue():  
 global q  
 global visited  
  
 q.sort(reverse = True)  
 visited = visited + [q[0][1]]  
  
 temp = q[0][1]  
 del q[0]  
 return (temp)  
  
def h\_val(curr\_state, final\_arr):  
 count = 0  
 for p in range(3):  
 for q in range(3):  
 if curr\_state[p][q]!=0 and curr\_state[p][q] == final\_arr[p][q]:  
 count = count + 1  
 return count  
  
def g\_val(curr\_state, initial\_arr):  
 count = 0  
 for p in range(3):  
 for q in range(3):  
 if curr\_state[p][q]!=0:

if curr\_state[p][q] == initial\_arr[p][q]:  
 count = count + 1  
 return count  
  
def heuristic(initial\_arr, curr\_state, final\_arr):  
 return (h\_val(curr\_state, final\_arr) + g\_val(curr\_state, initial\_arr))  
  
  
def findSol(initial\_arr, final\_arr):  
 global visited  
 global q  
 enqueue(initial\_arr, heuristic(initial\_arr, initial\_arr, final\_arr))  
 if initial\_arr == final\_arr:  
 return  
 while True:  
 if len(q)>0:  
 curr\_state = dequeue()  
 else:  
 print('Not Found')  
 return  
 idx = findZeroIndex(curr\_state)  
 for move in moves:  
 if isValidMove(curr\_state, idx, move):  
 new\_arr = performMove(curr\_state, idx, move)  
 if new\_arr == final\_arr:  
 print('Solution Found!! Intermediate states are:')  
 print(visited+[new\_arr])  
 return  
 if new\_arr not in visited:  
 h = heuristic(initial\_arr, new\_arr,final\_arr)  
 enqueue(new\_arr, h)  
  
  
def main():  
 global q  
 global visited  
 visited = []  
 q=[]  
 findSol(initial\_arr, final\_arr)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

**OUTPUT:**



1. Apply **AO\* searching algorithm** on the following search tree.



**CODE:**

def optimizePath(root):  
 global visited  
 global graph  
 q = []  
 # enqueue(q,[root],graph[root][1])  
 for children in graph[root][2]:  
 heu = 0  
 for child in children:  
 heu = heu + graph[child][1]  
 q = q + [(heu,children)]  
  
 if len(q)==0:  
 return graph[root][1]  
 graph[root][1] = 100  
 while True:  
 if len(q)>0:  
 q.sort()  
 visited.append(q[0][1])  
 curr\_children = q[0][1]  
 del q[0]  
 else:  
 return graph[root][1]  
 curr\_heu = 0  
 for curr\_child in curr\_children:  
 curr\_heu = curr\_heu + 1 + optimizePath(curr\_child)  
 if (curr\_heu < graph[root][1]):  
 graph[root][1] = curr\_heu  
  
def getPath(root):  
 global graph  
 q = []  
 for children in graph[root][2]:  
 heu = 0  
 for child in children:  
 heu = heu + graph[child][1]  
 q = q + [(heu, children)]  
 print(graph[root][0], end=' ')  
 if len(q) > 0:  
 q.sort()  
 curr\_children = q[0][1]  
 del q[0]  
 for curr\_child in curr\_children:  
 getPath(curr\_child)  
  
def AOstar(root):  
 global graph  
 minHue = optimizePath(root)  
 print(f'Optimized heuristic is {minHue} and optimised path is: ')  
 getPath(root)  
  
def main():  
 global visited  
 visited = []  
 global graph  
 # graph element: idx, heu, child  
 graph = [['A',100,[[1,2],[3]]],  
 ['B',6,[[6],[7]]],  
 ['C',12,[]],  
 ['D',10,[[4,5]]],  
 ['E',4,[]],  
 ['F',4,[]],  
 ['G',5,[]],  
 ['H',7,[]]]  
 root = 0  
 visited.append([root])  
 AOstar(root)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

**OUTPUT:**

