```
import copy
import time
nodeNumbers =0

# Calculates the heuristic adding both Manhattan and misplaced tiles
def bothHeuristic(node,hue):
    manhattanDistance= calculateHeuristic(node, 1)
    misplacedTileDistance=calculateHeuristic(node, 2)
    return (manhattanDistance + misplacedTileDistance)



#Calculates the heuristic based on the choice hue=1 Manhattan, hue = 2
Misplaced tiles
#Not considering blank
def calculateHeuristic_noBlankTile(node, hue):
    distance = 0

    #Manhattan distance
    if hue == 1:
        for current, target in enumerate(node):
            if(target!=0):
                target = finalNode.index(target)
                currentRow = int(current / 3)
                currentColumn = current % 3
                targetRow = int(target / 3)
                targetColumn = target % 3
                distance += abs(currentRow - targetRow) +
abs(currentColumn - targetColumn)

    #Misplaced tiles
    else:
        for i in range(9):
            if node[i] != finalNode[i] and node[i] != 0:
                distance += 1
    return distance


#Calculates the heuristic based on the choice hue=1 Manhattan, hue = 2
Misplaced tiles
#Considering blank
def calculateHeuristic(node, hue):
    distance = 0

    #Manhattan distance
    if hue == 1:
        for current, target in enumerate(node):
            target = finalNode.index(target)
            currentRow = int(current / 3)
            currentColumn = current % 3
```

```
            targetRow = int(target / 3)
            targetColumn = target % 3
            distance += abs(currentRow - targetRow) +
abs(currentColumn - targetColumn)

    #Misplaced tiles
    else:
        for i in range(9):
            if node[i] != finalNode[i]:
                distance += 1
    return distance

def checkFinal(node, visitedList):
    if node[:9] == finalNode:
        printNode(node)
        print("Hill climbing is Success")
        return True
    if node[:9] not in visitedList:
        printNode(node)
        nodeList.append(node)
    return False



def printNode(node):
    print(node[0], node[1], node[2])
    print(node[3], node[4], node[5])
    print(node[6], node[7], node[8])
    global nodeNumbers
    print('Node:', nodeNumbers)
    print('Depth:', len(node[9:]))
    print('Moves:', node[9:])
    print('------')
    nodeNumbers += 1


def printOnlyNodeValues(node):
    print(node[0], node[1], node[2])
    print(node[3], node[4], node[5])
    print(node[6], node[7], node[8])
    print('------')


# start hill climbing search with
def hillClimbing_bothHeurristic(choice):
    hue =choice
    nodeNumber = 0
    hValues = []
    visitedList = []
    nodeList = []
    global nodeNumbers
```

```
    nodeNumbers=0
    nodeList.append(startNode)

    printNode(startNode)
    found = False
    previous = 100

    t0 = time.time()
    while (found == False and len(nodeList) > 0):
        for node in nodeList:
            h= bothHeuristic(node[:9], hue)
            hValues.append(h)
        currentNode = nodeList[hValues.index(min(hValues))]
        print(hValues)
        if previous >= min(hValues):
            previous = min(hValues)
        else:
            print("Hill climbing Failure")
            break
        hValues = []
        nodeList = []
        visitedList.append(currentNode[:9])
        blankIndex = currentNode.index(0)
        if blankIndex != 0 and blankIndex != 1 and blankIndex != 2:
            upNode = copy.deepcopy(currentNode)
            upNode[blankIndex] = upNode[blankIndex - 3]
            upNode[blankIndex - 3] = 0
            upNode.append('up')
            found = checkFinal(upNode, visitedList)
        if blankIndex != 0 and blankIndex != 3 and blankIndex != 6 and
found == False:
            leftNode = copy.deepcopy(currentNode)
            leftNode[blankIndex] = leftNode[blankIndex - 1]
            leftNode[blankIndex - 1] = 0
            leftNode.append('left')
            found = checkFinal(leftNode, visitedList)
        if blankIndex != 6 and blankIndex != 7 and blankIndex != 8 and
found == False:
            downNode = copy.deepcopy(currentNode)
            downNode[blankIndex] = downNode[blankIndex + 3]
            downNode[blankIndex + 3] = 0
            downNode.append('down')
            found = checkFinal(downNode, visitedList)
        if blankIndex != 2 and blankIndex != 5 and blankIndex != 8 and
found == False:
            rightNode = copy.deepcopy(currentNode)
            rightNode[blankIndex] = rightNode[blankIndex + 1]
            rightNode[blankIndex + 1] = 0
            rightNode.append('right')
            found = checkFinal(rightNode, visitedList)
```

```
    t1 = time.time()
    print("Total no of state Explored:",(nodeNumbers-1))
    print('Time:', t1 - t0)
    print('------')

# start hill climbing search
def hillClimbing(choice,blankTile):
    hue =choice
    nodeNumber = 0
    hValues = []
    visitedList = []
    nodeList = []
    global nodeNumbers
    nodeNumbers=0
    nodeList.append(startNode)

    printNode(startNode)
    found = False
    previous = 100

    t0 = time.time()
    while (found == False and len(nodeList) > 0):
        for node in nodeList:
            if blankTile==0:
                h= calculateHeuristic_noBlankTile(node[:9], hue)
            else:
                h = calculateHeuristic(node[:9], hue)
            hValues.append(h)
        currentNode = nodeList[hValues.index(min(hValues))]
        print(hValues)
        if previous >= min(hValues):
            previous = min(hValues)
        else:
            print("Hill climbing Failure")
            break
        hValues = []
        nodeList = []
        visitedList.append(currentNode[:9])
        blankIndex = currentNode.index(0)
        if blankIndex != 0 and blankIndex != 1 and blankIndex != 2:
            upNode = copy.deepcopy(currentNode)
            upNode[blankIndex] = upNode[blankIndex - 3]
            upNode[blankIndex - 3] = 0
            upNode.append('up')
            found = checkFinal(upNode, visitedList)
        if blankIndex != 0 and blankIndex != 3 and blankIndex != 6 and
found == False:
            leftNode = copy.deepcopy(currentNode)
            leftNode[blankIndex] = leftNode[blankIndex - 1]
```

```
            leftNode[blankIndex - 1] = 0
            leftNode.append('left')
            found = checkFinal(leftNode, visitedList)
        if blankIndex != 6 and blankIndex != 7 and blankIndex != 8 and
found == False:
            downNode = copy.deepcopy(currentNode)
            downNode[blankIndex] = downNode[blankIndex + 3]
            downNode[blankIndex + 3] = 0
            downNode.append('down')
            found = checkFinal(downNode, visitedList)
        if blankIndex != 2 and blankIndex != 5 and blankIndex != 8 and
found == False:
            rightNode = copy.deepcopy(currentNode)
            rightNode[blankIndex] = rightNode[blankIndex + 1]
            rightNode[blankIndex + 1] = 0
            rightNode.append('right')
            found = checkFinal(rightNode, visitedList)

    t1 = time.time()
    print("Total no of state Explored:",(nodeNumbers-1))
    print('Time:', t1 - t0)
    print('------')


 # read the input from the file
startNode = []
finalNode = []
with open('MyFile.txt') as f:
    array = []
    for line in f:
            array.append([int(x) for x in line.split()])
    for i in range(3):
        for j in range(3):
            startNode.append(array[i][j])
    for i in range(4, 7):
        for j in range(3):
            finalNode.append(array[i][j])
```

i. The success or failure message ii. Heuristics chosen, Start state and Goal state iii. (Sub)Optimal Path (on success) iv. Total number of states explored v. Total amount of time taken All the above outputs are printed below

```
    # print the start node and Goal node
    print("Start Node:")
    printOnlyNodeValues(startNode)
    print("Goal Node:")
    printOnlyNodeValues(finalNode)
    print("Considering blank tile")
    print("Heuristic choosen:Manhattan:\n")
```

```
    hillClimbing(1,1)
    print("Considering blank tile")
    print("Heuristic choosen:Misplaced tiles:\n")
    hillClimbing(2,1)

    print("------------------------------")
    print("Start Node:")
    printOnlyNodeValues(startNode)
    print("Goal Node:")
    printOnlyNodeValues(finalNode)
    print("Not considering blank tile")
    print("Heuristic choosen:Manhattan:\n")
    hillClimbing(1,0)
    print("Not considering blank tile")
    print("Heuristic choosen:Misplaced tiles:\n")
    hillClimbing(2,0)

    print("------------------------------")
    print("Considering both Heuristic (h1 (n) + h2 (n))")
    print("Start Node:")
    printOnlyNodeValues(startNode)
    print("Goal Node:")
    printOnlyNodeValues(finalNode)
    hillClimbing_bothHeurristic(1)
```

```
Start Node:
1 2 3
4 5 6
7 0 8
------
Goal Node:
1 2 3
4 5 6
7 8 0
------
Considering blank tile
Heuristic choosen:Manhattan:

1 2 3
4 5 6
7 0 8
Node: 0
Depth: 0
Moves: []
------
[2]
1 2 3
4 0 6
7 5 8
```

```
Node: 1
Depth: 1
Moves: ['up']
------
1 2 3
4 5 6
0 7 8
Node: 2
Depth: 1
Moves: ['left']
------
1 2 3
4 5 6
7 8 0
Node: 3
Depth: 1
Moves: ['right']
------
Hill climbing is Success
Total no of state Explored: 3
Time: 0.0020689964294433594
------
Considering blank tile
Heuristic choosen:Misplaced tiles:

1 2 3
4 5 6
7 0 8
Node: 0
Depth: 0
Moves: []
------
[2]
1 2 3
4 0 6
7 5 8
Node: 1
Depth: 1
Moves: ['up']
------
1 2 3
4 5 6
0 7 8
Node: 2
Depth: 1
Moves: ['left']
------
1 2 3
4 5 6
7 8 0
```

```
Node: 3
Depth: 1
Moves: ['right']
------
Hill climbing is Success
Total no of state Explored: 3
Time: 0.0012249946594238281
------
------------------------------
Start Node:
1 2 3
4 5 6
7 0 8
------
Goal Node:
1 2 3
4 5 6
7 8 0
------
Not considering blank tile
Heuristic choosen:Manhattan:

1 2 3
4 5 6
7 0 8
Node: 0
Depth: 0
Moves: []
------
[1]
1 2 3
4 0 6
7 5 8
Node: 1
Depth: 1
Moves: ['up']
------
1 2 3
4 5 6
0 7 8
Node: 2
Depth: 1
Moves: ['left']
------
1 2 3
4 5 6
7 8 0
Node: 3
Depth: 1
Moves: ['right']
```

```
------
Hill climbing is Success
Total no of state Explored: 3
Time: 0.00146484375
------
Not considering blank tile
Heuristic choosen:Misplaced tiles:

1 2 3
4 5 6
7 0 8
Node: 0
Depth: 0
Moves: []
------
[1]
1 2 3
4 0 6
7 5 8
Node: 1
Depth: 1
Moves: ['up']
------
1 2 3
4 5 6
0 7 8
Node: 2
Depth: 1
Moves: ['left']
------
1 2 3
4 5 6
7 8 0
Node: 3
Depth: 1
Moves: ['right']
------
Hill climbing is Success
Total no of state Explored: 3
Time: 0.001459360122680664
------
------------------------------
Considering both Heuristic (h1 (n) + h2 (n))
Start Node:
1 2 3
4 5 6
7 0 8
------
Goal Node:
1 2 3
```

```
4 5 6
7 8 0
------
1 2 3
4 5 6
7 0 8
Node: 0
Depth: 0
Moves: []
------
Both: 4
[4]
1 2 3
4 0 6
7 5 8
Node: 1
Depth: 1
Moves: ['up']
------
1 2 3
4 5 6
0 7 8
Node: 2
Depth: 1
Moves: ['left']
------
1 2 3
4 5 6
7 8 0
Node: 3
Depth: 1
Moves: ['right']
------
Hill climbing is Success
Total no of state Explored: 3
Time: 0.0014798641204833984
------
```

(e). Constraints: i. Check whether the heuristics are admissible Ans :Yes, Heuristic are admissible.The Manhattan distance is an admissible heuristic in this case because every tile will have to be moved at least the number of spots in between itself and its correct position. The misplaced tiles also admissible heuristic.

(ii). What happens if we make a new heuristics h3(n)= h1 (n) + h2 (n)? Ans : h(n) = h1(n)+h2(n) is admissible, since given that h1(n) ≤ h ∗ (n) and h2(n) ≤ h ∗ (n) we deduce h1(n)+h2(n) 2 ≤ h ∗ (n). In the output added above , we can see there is no difference in the state explored.

(iii). What happens if you consider the blank tile as another tile? (for each heuristic as mentioned in d) Ans. The output is added above , it does not change anything if we

consider the blank tile or do not consider the tiles. (iv). What if the search algorithm got stuck into Local maximum? Is there any way to get out of this?

Ans. Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

(v). What happens when all the neighbours of the current state have the same value? How to get rid of this situation?

Ans : The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.