In [3]:

```
from copy import deepcopy
import numpy as np
import time

# takes the input of current states and evaluvates the best path to goal state
def bestsolution(state):
    bestsol = np.array([], int).reshape(-1, 9)
    count = len(state) - 1
    while count != -1:
        bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)
        count = (state[count]['parent'])
    return bestsol.reshape(-1, 3, 3)


# this function checks for the uniqueness of the iteration(it) state, weather it has been previous
ly traversed or not.
def all(checkarray):
    set=[]
    for it in set:
        for checkarray in it:
            return 1
        else:
            return 0


# calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state
def manhattan(puzzle, goal):
    a = abs(puzzle // 3 - goal // 3)
    b = abs(puzzle % 3 - goal % 3)
    mhcost = a + b
    return sum(mhcost[1:])



# will calcuates the number of misplaced tiles in the current state as compared to the goal state
def misplaced_tiles(puzzle,goal):
    mscost = np.sum(puzzle != goal) - 1
    return mscost if mscost > 0 else 0



#3[on_true] if [expression] else [on_false]


# will indentify the coordinates of each of goal or initial state values
def coordinates(puzzle):
    pos = np.array(range(9))
    for p, q in enumerate(puzzle):
        pos[q] = p
    return pos



# start of 8 puzzle evaluvation, using Manhattan heuristics
def evaluate(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right'
, [2, 5, 8],  1)],
                 dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

     # initializing the parent, gn and hn, where hn is manhattan distance function call
    costg = coordinates(goal)
    parent = -1
    gn = 0
    hn = manhattan(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]
    priority = np.array( [(0, hn)], dtpriority)
```

```
    while 1:
        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
        position, fn = priority[0]
        priority = np.delete(priority, 0, 0)
        # sort priority queue using merge sort,the first element is picked for exploring remove
from queue what we are exploring
        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)
        # Identify the blank square in input
        blank = int(np.where(puzzle == 0)[0])
        gn = gn + 1
        c = 1
        start_time = time.time()
        for s in steps:
            c = c + 1
            if blank not in s['position']:
                # generate new state as copy of current
                openstates = deepcopy(puzzle)
                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], op
enstates[blank]
                # The all function is called, if the node has been previously explored or not
                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable ! \n")
                        exit
                    # calls the manhattan function to calcuate the cost
                    hn = manhattan(coordinates(openstates), costg)
                     # generate and add new state in the list
                    q = np.array([(openstates, position, gn, hn)], dtstate)
                    state = np.append(state, q, 0)
                    # f(n) is the sum of cost to reach node and the cost to rech fromt he node to t
he goal state
                    fn = gn + hn

                    q = np.array([(len(state) - 1, fn)], dtpriority)
                    priority = np.append(priority, q, 0)
                      # Checking if the node in openstates are matching the goal state.
                    if np.array_equal(openstates, goal):
                        print(' The 8 puzzle is solvable ! \n')
                        return state, len(priority)


    return state, len(priority)


# start of 8 puzzle evaluvation, using Misplaced tiles heuristics
def evaluvate_misplaced(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right'
, [2, 5, 8],  1)],
                dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

    costg = coordinates(goal)
    # initializing the parent, gn and hn, where hn is misplaced_tiles  function call
    parent = -1
    gn = 0
    hn = misplaced_tiles(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

   # We make use of priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]

    priority = np.array([(0, hn)], dtpriority)

    while 1:
        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
        position, fn = priority[0]
        # sort priority queue using merge sort,the first element is picked for exploring.

        priority = np.delete(priority, 0, 0)
        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)
```

```
         # Identify the blank square in input
        blank = int(np.where(puzzle == 0)[0])
        # Increase cost g(n) by 1
        gn = gn + 1
        c = 1
        start_time = time.time()
        for s in steps:
            c = c + 1
            if blank not in s['position']:
                 # generate new state as copy of current
                openstates = deepcopy(puzzle)
                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], op
enstates[blank]
                # The check function is called, if the node has been previously explored or not.
                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable \n")
                        break
                    # calls the Misplaced_tiles function to calcuate the cost
                    hn = misplaced_tiles(coordinates(openstates), costg)
                    # generate and add new state in the list
                    q = np.array([(openstates, position, gn, hn)], dtstate)
                    state = np.append(state, q, 0)
                    # f(n) is the sum of cost to reach node and the cost to rech fromt he node to t
he goal state
                    fn = gn + hn

                    q = np.array([(len(state) - 1, fn)], dtpriority)
                    priority = np.append(priority, q, 0)
                    # Checking if the node in openstates are matching the goal state.
                    if np.array_equal(openstates, goal):
                        print(' The 8 puzzle is solvable \n')
                        return state, len(priority)

    return state, len(priority)



# ----------  Program start -----------------


 # User input for initial state
puzzle = []
print(" Input vals from 0-8 for start state ")
for i in range(0,9):
    x = int(input("enter vals :"))
    puzzle.append(x)

 # User input of goal state
goal = []
print(" Input vals from 0-8 for goal state ")
for i in range(0,9):
    x = int(input("Enter vals :"))
    goal.append(x)



n = int(input("1. Manhattan distance \n2. Misplaced tiles"))

if(n ==1 ):
    state, visited = evaluvate(puzzle, goal)
    bestpath = bestsolution(state)
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
    print('Total nodes visited: ',visit, "\n")
    print('Total generated:', len(state))

if(n == 2):
    state, visited = evaluvate_misplaced(puzzle, goal)
    bestpath = bestsolution(state)
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
```

```
    print('Total nodes visited: ',visit, "\n")
    print('Total generated:', len(state))
```

```
 Input vals from 0-8 for start state
enter vals :1
enter vals :2
enter vals :0
enter vals :4
enter vals :5
enter vals :3
enter vals :7
enter vals :8
enter vals :6
 Input vals from 0-8 for goal state
Enter vals :1
Enter vals :2
Enter vals :3
Enter vals :4
Enter vals :5
Enter vals :6
Enter vals :7
Enter vals :8
Enter vals :0
1. Manhattan distance
2. Misplaced tiles2
 The 8 puzzle is solvable

   1 2 0
   4 5 3
   7 8 6

   1 2 3
   4 5 0
   7 8 6

   1 2 3
   4 5 6
   7 8 0
Steps to reach goal: 2
Total nodes visited:  2

Total generated: 4
```

In [ ]: