Name: Swapnil

Group: 2CS12

Roll Number: 102016108

Artificial Intelligence (AI)

UCS411

Assignment-3

Question-1: If the initial and final states are as below, find the value of Heuristic function, by taking

- **Euclidean Distance**

Initial:

2		3
1	8	4
7	6	5

Goal:

1	2	3
8		4
7	6	5

Code:

```
from copy import deepcopy

class Puzzle:
    def __init__(self, initial_state):
        self.initial = initial_state
        self.queue = []
        self.visited = []


    def _find_pos(self, state):
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    return i, j


    def _right(self, state, pos):
        i, j = pos


        if j != 2:
            t = deepcopy(state)
```

```
            t[i][j], t[i][j+1] = t[i][j+1], t[i][j]
            return t
        else:
            return state

    def _left(self, state, pos):
        i, j = pos

        if j != 0:
            t = deepcopy(state)
            t[i][j], t[i][j-1] = t[i][j-1], t[i][j]
            return t
        else:
            return state

    def _up(self, state, pos):
        i, j = pos

        if i != 0:
            t = deepcopy(state)
            t[i][j], t[i-1][j] = t[i-1][j], t[i][j]
```

```python
            return t
        else:
            return state

    def _down(self, state, pos):
        i, j = pos

        if i != 2:
            t = deepcopy(state)
            t[i][j], t[i+1][j] = t[i+1][j], t[i][j]
            return t
        else:
            return state

    def _enque(self, new_state):

        x = self._heu_el(new_state)

        if len(self.queue) == 0:
            self.queue.append(new_state)
```

```
        elif x < self._heu_el(self.queue[0]):
            self.queue.insert(0, new_state)
        else:
            for i in range(1, len(self.queue)):
                if self._heu_el(self.queue[i]) > x:
                    self.queue.insert(i-1, new_state)

    def _deque(self):
        self.visited.append(self.queue[0])

        ele = self.queue.pop(0)

        return ele

    def _heu_el(self, state):
        val = 0

        for x in range(3):
            for i in range(3):
```

```
                q = state[x][i]



                for j in range(3):

                    for k in range(3):

                        if q == self.goal[j][k] and not (x == j and i
== k):

                            val += pow(abs(x-j)+abs(i-k),2)

                            break

        return pow(val,1/2)



    def _print(self, vis):

        for k in range(len(vis)-1):

            x = vis[k]

            for i in range(3):

                for j in range(3):

                    print(x[i][j], end=" ")

                print("\n")

            print("  |")

            print("  |")

            print("  V")

        x = vis[-1]

        for i in range(3):
```

```
        for j in range(3):
            print(x[i][j], end=" ")
        print("\n")


def Solve(self, goal_state):


    current_state = deepcopy(self.initial)
    self.goal = goal_state
    if current_state == goal_state:
        return


    while 1:
        pos = self._find_pos(current_state)
        new_state = self._down(current_state, pos)
        if new_state != current_state:
            if new_state == goal_state:
                print("Goal State Reached!!")
                self.visited.append(new_state)
                self._print(self.visited)
                return
            else:
```

```
        if new_state not in self.visited:
            self._enque(new_state)


new_state = self._left(current_state, pos)
if new_state != current_state:
    if new_state == goal_state:
        print("Goal State Reached!!")
        self.visited.append(new_state)


        self._print(self.visited)
        return
    else:
        if new_state not in self.visited:
            self._enque(new_state)


new_state = self._right(current_state, pos)
if new_state != current_state:
    if new_state == goal_state:
        print("Goal State Reached!!")
        self.visited.append(new_state)
```

```
        self._print(self.visited)
        return
    else:
        if new_state not in self.visited:
            self._enque(new_state)


new_state = self._up(current_state, pos)
if new_state != current_state:
    if new_state == goal_state:
        print("Goal State Reached!!")
        self.visited.append(new_state)




        self._print(self.visited)
        return
    else:
        if new_state not in self.visited:
            self._enque(new_state)
```

```
        if len(self.queue) > 0:
            current_state = self._deque()
        else:
            print("Not Found!")
            return


if __name__ == '__main__':
    P = Puzzle([[2, 0, 3], [1, 8, 4], [7, 6, 5]])
    P.Solve([[1, 2, 3], [8, 0, 4], [7, 6, 5]])
```

Output:

```
Goal State Reached!!
2 8 3

1 0 4

7 6 5

  |
  |
  V
2 0 3

1 8 4

7 6 5

  |
  |
  V
0 2 3

1 8 4

7 6 5

  |
  |
  V
1 2 3

0 8 4

7 6 5

  |
  |
  V
1 2 3

8 0 4

7 6 5



...Program finished with exit code 0
Press ENTER to exit console.
```

- **Manhattan Distance**

Initial:

2		3
1	8	4
7	6	5

Goal:

1	2	3
8		4
7	6	5

Code:

```
from copy import deepcopy

class Puzzel:

    def __init__(self,initial_state):

        self.initial = initial_state

        self.queue = []

        self.visited = []



    def _find_pos(self,state):

        for i in range(3):

            for j in range(3):
```

```
            if state[i][j] == 0:
                return i,j

def _right(self,state,pos):
    i,j = pos


    if j != 2:
        t = deepcopy(state)
        t[i][j],t[i][j+1] = t[i][j+1],t[i][j]
        return t
    else:
        return state

def _left(self,state,pos):
    i,j = pos


    if j!=0:
        t= deepcopy(state)
        t[i][j],t[i][j-1] = t[i][j-1],t[i][j]
        return t
    else:
        return state
```

```
def _up(self,state,pos):
    i,j = pos


    if i != 0:
        t= deepcopy(state)
        t[i][j],t[i-1][j] = t[i-1][j],t[i][j]
        return t
    else:
        return state

def _down(self,state,pos):
    i,j = pos

    if i != 2:
        t=deepcopy(state)
        t[i][j],t[i+1][j] = t[i+1][j],t[i][j]
        return t
    else:
        return state


def _enque(self,new_state):
```

```
        x =self._heu_man(new_state)


        if len(self.queue) == 0:
            self.queue.append(new_state)


        elif  x < self._heu_man(self.queue[0]):
            self.queue.insert(0,new_state)
        else:
            for i in range(1,len(self.queue)):
                if self._heu_man(self.queue[i]) > x:
                    self.queue.insert(i-1,new_state)

    def _deque(self):
        self.visited.append(self.queue[0])


        ele = self.queue.pop(0)


        return ele
```

```
    def _heu_man(self,state):
        val = 0

        for x in range(3):
            for i in range(3):
                q = state[x][i]

                for j in range(3):
                    for k in range(3):
                        if q == self.goal[j][k] and not (x ==j and
i==k):
                            val += abs(x-j)+abs(i-k)
                            break
        return val


    def _print(self,vis):
        for k in range(len(vis)-1):
            x = vis[k]
            for i in range(3):
                for j in range(3):
                    print(x[i][j],end=" ")
```

```
            print("\n")
        print("  |")
        print("  |")
        print("  V")
    x = vis[-1]
    for i in range(3):
        for j in range(3):
            print(x[i][j],end=" ")
        print("\n")



def Solve(self,goal_state):


    current_state = deepcopy(self.initial)
    self.goal = goal_state
    if current_state == goal_state:
        return


    while 1:
        pos = self._find_pos(current_state)
        new_state = self._down(current_state,pos)
```

```
if new_state != current_state:
    if new_state == goal_state:
        print("Goal State Reached!!")
        self.visited.append(new_state)


        self._print(self.visited)
        return
    else:
        if new_state not in self.visited:
            self._enque(new_state)

new_state = self._left(current_state,pos)
if new_state != current_state:
    if new_state == goal_state:
        print("Goal State Reached!!")
        self.visited.append(new_state)



        self._print(self.visited)
        return
    else:
        if new_state not in self.visited:
```

```
            self._enque(new_state)


new_state = self._right(current_state,pos)
if new_state != current_state:
    if new_state == goal_state:
        print("Goal State Reached!!")
        self.visited.append(new_state)




        self._print(self.visited)
        return
    else:
        if new_state not in self.visited:
            self._enque(new_state)


new_state = self._up(current_state,pos)
if new_state != current_state:
    if new_state == goal_state:
        print("Goal State Reached!!")
        self.visited.append(new_state)


        self._print(self.visited)
```

```
                return
            else:
                if new_state not in self.visited:
                    self._enque(new_state)






        if len(self.queue) >0:
            current_state =self._deque()
        else:
            print("Not Found!")
            return




if __name__ == '__main__':
    P = Puzzel([[2,0,3],[1,8,4],[7,6,5]])
    P.Solve([[1,2,3],[8,0,4],[7,6,5]])
```

Output:

```
Goal State Reached!!
2 8 3

1 0 4

7 6 5

  |
  |
  V
2 0 3

1 8 4

7 6 5

  |
  |
  V
0 2 3

1 8 4

7 6 5

  |
  |
  V
1 2 3

0 8 4

7 6 5

  |
  |
  V
1 2 3

8 0 4

7 6 5



...Program finished with exit code 0
Press ENTER to exit console.
```

- **Minkowski Distance**

Initial:

2		3
1	8	4
7	6	5

Goal:

1	2	3
8		4
7	6	5

Code:

```
from copy import deepcopy

class Puzzel:

    def __init__(self, initial_state):

        self.initial = initial_state

        self.queue = []

        self.visited = []



    def _find_pos(self, state):

        for i in range(3):
```

```
        for j in range(3):
            if state[i][j] == 0:
                return i, j


def _right(self, state, pos):
    i, j = pos


    if j != 2:
        t = deepcopy(state)
        t[i][j], t[i][j+1] = t[i][j+1], t[i][j]
        return t
    else:
        return state


def _left(self, state, pos):
    i, j = pos


    if j != 0:
        t = deepcopy(state)
        t[i][j], t[i][j-1] = t[i][j-1], t[i][j]
        return t
```

```
        else:
            return state

    def _up(self, state, pos):
        i, j = pos

        if i != 0:
            t = deepcopy(state)
            t[i][j], t[i-1][j] = t[i-1][j], t[i][j]
            return t
        else:
            return state

    def _down(self, state, pos):
        i, j = pos

        if i != 2:
            t = deepcopy(state)
            t[i][j], t[i+1][j] = t[i+1][j], t[i][j]
            return t
        else:
```

```
        return state

def _enque(self, new_state):

    x = self._heu_mok(new_state)

    if len(self.queue) == 0:
        self.queue.append(new_state)

    elif x < self._heu_mok(self.queue[0]):
        self.queue.insert(0, new_state)
    else:
        for i in range(1, len(self.queue)):
            if self._heu_mok(self.queue[i]) > x:
                self.queue.insert(i-1, new_state)

def _deque(self):

    self.visited.append(self.queue[0])
```

```
        ele = self.queue.pop(0)

        return ele

    def _heu_mok(self, state):
        val = 0

        for x in range(3):
            for i in range(3):
                q = state[x][i]

                for j in range(3):
                    for k in range(3):
                        if q == self.goal[j][k] and not (x == j and i
== k):
                            val += pow(abs(x-j)+abs(i-k),3)
                            break
        return pow(val,1/3)

    def _print(self, vis):
        for k in range(len(vis)-1):
```

```
            x = vis[k]
            for i in range(3):
                for j in range(3):
                    print(x[i][j], end=" ")
                print("\n")
            print("  |")
            print("  |")
            print("  V")
        x = vis[-1]
        for i in range(3):
            for j in range(3):
                print(x[i][j], end=" ")
            print("\n")


    def Solve(self, goal_state):


        current_state = deepcopy(self.initial)
        self.goal = goal_state
        if current_state == goal_state:
            return
```

```
while 1:
    pos = self._find_pos(current_state)

    new_state = self._down(current_state, pos)

    if new_state != current_state:
        if new_state == goal_state:
            print("Goal State Reached!!")
            self.visited.append(new_state)



            self._print(self.visited)
            return
        else:
            if new_state not in self.visited:
                self._enque(new_state)


    new_state = self._left(current_state, pos)
    if new_state != current_state:
        if new_state == goal_state:
            print("Goal State Reached!!")
            self.visited.append(new_state)
```

```
            self._print(self.visited)
            return
        else:
            if new_state not in self.visited:
                self._enque(new_state)


    new_state = self._right(current_state, pos)
    if new_state != current_state:
        if new_state == goal_state:
            print("Goal State Reached!!")
            self.visited.append(new_state)



            self._print(self.visited)
            return
        else:
            if new_state not in self.visited:
                self._enque(new_state)
```

```
            new_state = self._up(current_state, pos)

            if new_state != current_state:

                if new_state == goal_state:

                    print("Goal State Reached!!")

                    self.visited.append(new_state)




                    self._print(self.visited)

                    return

                else:

                    if new_state not in self.visited:

                        self._enque(new_state)




            if len(self.queue) > 0:

                current_state = self._deque()

            else:

                print("Not Found!")

                return



if __name__ == '__main__':
```

```
P = Puzzel([[2, 0, 3], [1, 8, 4], [7, 6, 5]])

P.Solve([[1, 2, 3], [8, 0, 4], [7, 6, 5]])
```

Output:

```
Goal State Reached!!
2 8 3

1 0 4

7 6 5

  |
  |
  V
2 0 3

1 8 4

7 6 5

  |
  |
  V
0 2 3

1 8 4

7 6 5

  |
  |
  V
1 2 3

0 8 4

7 6 5

  |
  |
  V
1 2 3

8 0 4

7 6 5



...Program finished with exit code 0
Press ENTER to exit console.
```

Question-2: 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 BestFirst Search algorithm.

Initial:

2		3
1	8	4
7	6	5

Goal:

1	2	3
8		4
7	6	5

Code:

```
import copy



class MyEightPuzzle:

    def __init__(self, startState, goalState):

        self.currentState=startState

        self.goalState=goalState

        self.emptyIndex=self.emptyTileIndex()

        self.prevState=None



    def up(self):

        if self.emptyIndex==6 or self.emptyIndex==7 or self.emptyIndex==8:

            return False

        else:

            self.prevState=copy.deepcopy(self)
```

```
self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex+3
]

            self.currentState[self.emptyIndex+3]=0

            self.emptyIndex=self.emptyIndex+3

            return True



    def down(self):

        if self.emptyIndex==0 or self.emptyIndex==1 or
self.emptyIndex==2:

            return False

        else:

            self.prevState=copy.deepcopy(self)

self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex-
3]

            self.currentState[self.emptyIndex-3]=0

            self.emptyIndex=self.emptyIndex-3

            return True



    def left(self):

        if self.emptyIndex==2 or self.emptyIndex==5 or
self.emptyIndex==8:

            return False

        else:
```

```
            self.prevState=copy.deepcopy(self)

self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex+1
]
            self.currentState[self.emptyIndex+1]=0
            self.emptyIndex=self.emptyIndex+1
            return True

    def right(self):
        if self.emptyIndex==0 or self.emptyIndex==3 or
self.emptyIndex==6:
            #print("Cannot move")
            return False
        else:
            self.prevState=copy.deepcopy(self)

self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex-
1]
            self.currentState[self.emptyIndex-1]=0
            self.emptyIndex=self.emptyIndex-1
            return True

    def displayState(self):
        print("-------------------------------------")
```

```
        for i in range(0, 8, 3):
            print(self.currentState[i], self.currentState[i+1],
self.currentState[i+2])


    def emptyTileIndex(self):
        for i in range(0, 9):
            if self.currentState[i]==0:
                return i


    def isGoalReached(self):
        if self.currentState==self.goalState:
            return True
        else:
            return False



    def _eq_(self, other):
        return self.currentState==other.currentState



    def possibleNextStates(self):
        stateList=[]


        up_state=copy.deepcopy(self)
```

```
        if up_state.up():
            stateList.append(up_state)

        down_state=copy.deepcopy(self)
        if down_state.down():
            stateList.append(down_state)

        left_state=copy.deepcopy(self)
        if left_state.left():
            stateList.append(left_state)

        right_state=copy.deepcopy(self)
        if right_state.right():
            stateList.append(right_state)

        return stateList

def heuristic(self):
    count=0
    for i in range(0, 9):
```

```
        if self.goalState[i]!=self.currentState[i] and 
self.goalState[i]!=0:
            count=count+1
    return count



def constructPath(goalState):
    print("The solution path from Goal to Start")
    while goalState is not None:
        goalState.displayState()
        goalState=goalState.prevState



def BestFirstSearch(startState):
    open=[]
    closed=[]



    open.append(startState)



    while open:
```

```
        thisState=open.pop(0)

        thisState.displayState()

        closed.append(thisState)


        if thisState.isGoalReached():

            print("Goal state found.. stopping search")

            constructPath(thisState)

            break


        nextStates=thisState.possibleNextStates()


        for eachState in nextStates:

            if eachState not in open and eachState not in closed:

                open.append(eachState)

                open.sort(key=heuristic)


start=[2, 0, 3, 1, 8, 4, 7, 6, 5]

goal= [1, 2, 3, 8, 0, 4, 7, 6, 5]

problem=MyEightPuzzle(start, goal)

BestFirstSearch(problem)
```

Output:

```
-------------------------------------
2 0 3
1 8 4
7 6 5
-------------------------------------
0 2 3
1 8 4
7 6 5
-------------------------------------
1 2 3
0 8 4
7 6 5
-------------------------------------
1 2 3
8 0 4
7 6 5
Goal state found.. stopping search
The solution path from Goal to Start
-------------------------------------
1 2 3
8 0 4
7 6 5
-------------------------------------
1 2 3
0 8 4
7 6 5
-------------------------------------
0 2 3
1 8 4
7 6 5
-------------------------------------
2 0 3
1 8 4
7 6 5


...Program finished with exit code 0
Press ENTER to exit console.
```

Question-3: 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 Hill Climbing algorithm.

Initial:

2		3
1	8	4
7	6	5

Goal:

1	2	3
8		4
7	6	5

Code:

```
import copy

class MyEightPuzzle:

    def __init__(self, startState, goalState):

        self.currentState=startState

        self.goalState=goalState

        self.emptyIndex=self.emptyTileIndex()

        self.prevState=None



    def up(self):
```

```
        if self.emptyIndex==6 or self.emptyIndex==7 or
self.emptyIndex==8:

            return False

        else:

            self.prevState=copy.deepcopy(self)


self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex+3
]

            self.currentState[self.emptyIndex+3]=0

            self.emptyIndex=self.emptyIndex+3

            return True



    def down(self):

        if self.emptyIndex==0 or self.emptyIndex==1 or
self.emptyIndex==2:

            return False

        else:

            self.prevState=copy.deepcopy(self)


self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex-
3]

            self.currentState[self.emptyIndex-3]=0

            self.emptyIndex=self.emptyIndex-3

            return True
```

```
    def left(self):

        if self.emptyIndex==2 or self.emptyIndex==5 or 
self.emptyIndex==8:

            return False

        else:

            self.prevState=copy.deepcopy(self)


self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex+1
]

            self.currentState[self.emptyIndex+1]=0

            self.emptyIndex=self.emptyIndex+1

            return True



    def right(self):

        if self.emptyIndex==0 or self.emptyIndex==3 or 
self.emptyIndex==6:

            return False

        else:

            self.prevState=copy.deepcopy(self)


self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex-
1]

            self.currentState[self.emptyIndex-1]=0

            self.emptyIndex=self.emptyIndex-1

            return True
```

```
    def displayState(self):

        print("-------------------------------------")

        for i in range(0, 8, 3):

            print(self.currentState[i], self.currentState[i+1],
self.currentState[i+2])



    def emptyTileIndex(self):

        for i in range(0, 9):

            if self.currentState[i]==0:

                return i



    def isGoalReached(self):

        if self.currentState==self.goalState:

            return True

        else:

            return False




    def _eq_(self, other):

        return self.currentState==other.currentState




    def possibleNextStates(self):
```

```
        stateList=[]

        up_state=copy.deepcopy(self)
        if up_state.up():
            stateList.append(up_state)

        down_state=copy.deepcopy(self)
        if down_state.down():
            stateList.append(down_state)

        left_state=copy.deepcopy(self)
        if left_state.left():
            stateList.append(left_state)

        right_state=copy.deepcopy(self)
        if right_state.right():
            stateList.append(right_state)

        return stateList

    def heuristic(self):
```

```
        count=0

        for i in range(0, 9):

            if self.goalState[i]!=self.currentState[i] and
self.goalState[i]!=0:

                count=count+1

        return count



def constructPath(goalState):

    print("The solution path from Goal to Start")

    while goalState is not None:

        goalState.displayState()

        goalState=goalState.prevState



def HillClimbing(startState):

    open=[]

    closed=[]



    open.append(startState)




    while open:
```

```
thisState=open.pop(0)
thisState.displayState()



if thisState.isGoalReached():
    print("Goal state found.. stopping search")
    constructPath(thisState)
    break



nextStates=thisState.possibleNextStates()



for eachState in nextStates:
    if eachState not in open and eachState not in closed:
        if eachState.heuristic() < thisState.heuristic():
            open.append(eachState)
            closed.append(thisState)
```

```
start=[2, 0, 3, 1, 8, 4, 7, 6, 5]

goal= [1, 2, 3, 8, 0, 4, 7, 6, 5]

problem=MyEightPuzzle(start, goal)

HillClimbing(problem)
```

Output:

```
--------------------------------------
2 0 3
1 8 4
7 6 5
--------------------------------------
0 2 3
1 8 4
7 6 5
--------------------------------------
1 2 3
0 8 4
7 6 5
--------------------------------------
1 2 3
8 0 4
7 6 5
Goal state found.. stopping search
The solution path from Goal to Start
--------------------------------------
1 2 3
8 0 4
7 6 5
--------------------------------------
1 2 3
0 8 4
7 6 5
--------------------------------------
0 2 3
1 8 4
7 6 5
--------------------------------------
2 0 3
1 8 4
7 6 5


...Program finished with exit code 0
Press ENTER to exit console.
```

Question-4: If the initial and final states are as below and H(n): Manhattan distance as the heuristic function. You need to use Best First Search algorithm.

2	8	3
1	5	4
7	6	

Initial State

1	2	3
8		4
7	6	5

Final State

Code:

```
import copy


class MyEightPuzzle:

    def __init__(self, startState, goalState):

        self.currentState=startState

        self.goalState=goalState

        self.emptyIndex=self.emptyTileIndex()

        self.prevState=None
```

```
    def up(self):

        if self.emptyIndex==6 or self.emptyIndex==7 or
self.emptyIndex==8:

            return False

        else:

            self.prevState=copy.deepcopy(self)


self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex+3
]

            self.currentState[self.emptyIndex+3]=0

            self.emptyIndex=self.emptyIndex+3

            return True




    def down(self):

        if self.emptyIndex==0 or self.emptyIndex==1 or
self.emptyIndex==2:



            return False

        else:

            self.prevState=copy.deepcopy(self)


self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex-
3]
```

```
            self.currentState[self.emptyIndex-3]=0

            self.emptyIndex=self.emptyIndex-3

            return True



    def left(self):

        if self.emptyIndex==2 or self.emptyIndex==5 or
self.emptyIndex==8:

            return False

        else:

            self.prevState=copy.deepcopy(self)

self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex+1
]

            self.currentState[self.emptyIndex+1]=0

            self.emptyIndex=self.emptyIndex+1

            return True



    def right(self):

        if self.emptyIndex==0 or self.emptyIndex==3 or
self.emptyIndex==6:

            return False

        else:

            self.prevState=copy.deepcopy(self)
```

```
self.currentState[self.emptyIndex]=self.currentState[self.emptyIndex-
1]

            self.currentState[self.emptyIndex-1]=0

            self.emptyIndex=self.emptyIndex-1

            return True



    def displayState(self):

        print("-------------------------------------")

        for i in range(0, 8, 3):

            print(self.currentState[i], self.currentState[i+1],
self.currentState[i+2])


    def emptyTileIndex(self):

        for i in range(0, 9):

            if self.currentState[i]==0:

                return i


    def isGoalReached(self):

        if self.currentState==self.goalState:

            return True

        else:

            return False
```

```
def _eq_(self, other):
    return self.currentState==other.currentState


def possibleNextStates(self):
    stateList=[]

    up_state=copy.deepcopy(self)
    if up_state.up():
        stateList.append(up_state)


    down_state=copy.deepcopy(self)
    if down_state.down():
        stateList.append(down_state)

    left_state=copy.deepcopy(self)
    if left_state.left():
        stateList.append(left_state)


    right_state=copy.deepcopy(self)
    if right_state.right():
```

```
                stateList.append(right_state)

            return stateList

    def heuristic(self):
        sum=0
        for i in range(0, 9):

            goalNode=self.goalState[i]
            if goalNode==0:
                continue
            goalIndex=i


            for j in range(0, 9):
                currentNode=self.currentState[j]
                if currentNode==goalNode:
                    currentIndex=j
                    break

            difference=abs(goalIndex-currentIndex)
            if difference<3:
```

```
            moves=difference
        elif difference>=3 and difference<6:
            moves=difference%3 + 1
        elif difference>=6 and difference<8:
            moves=difference%3 + 2


        sum=sum+moves
        return sum


def constructPath(goalState):
    print("The solution path from Goal to Start")
    while goalState is not None:
        goalState.displayState()
        goalState=goalState.prevState


def BestFirstSearch(startState):
    open=[]
    closed=[]



    open.append(startState)
```

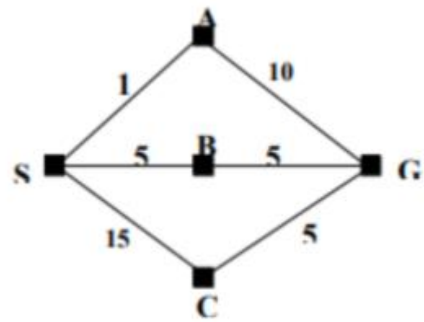
```
while open:

    thisState=open.pop(0)
    thisState.displayState()
    closed.append(thisState)

    if thisState.isGoalReached():
        print("Goal state found.. stopping search")
        constructPath(thisState)
        break

    nextStates=thisState.possibleNextStates()

    for eachState in nextStates:
        if eachState not in open and eachState not in closed:
```

```
                open.append(eachState)

                open.sort(key=heuristic)



start=[2, 0, 3, 1, 8, 4, 7, 6, 5]

goal= [1, 2, 3, 8, 0, 4, 7, 6, 5]

problem=MyEightPuzzle(start, goal)

BestFirstSearch(problem)
```

Output:

```
-------------------------------------
2 0 3
1 8 4
7 6 5
-------------------------------------
2 8 3
1 0 4
7 6 5
-------------------------------------
2 3 0
1 8 4
7 6 5
-------------------------------------
0 2 3
1 8 4
7 6 5
-------------------------------------
1 2 3
0 8 4
7 6 5
-------------------------------------
1 2 3
7 8 4
0 6 5
-------------------------------------
1 2 3
8 0 4
7 6 5
Goal state found.. stopping search
The solution path from Goal to Start
-------------------------------------
1 2 3
8 0 4
7 6 5
-------------------------------------
1 2 3
0 8 4
7 6 5
-------------------------------------
0 2 3
1 8 4
7 6 5
-------------------------------------
2 0 3
1 8 4
7 6 5


...Program finished with exit code 0
Press ENTER to exit console.
```

Question-5: Solve this given problem using Uniform Cost search. A is the initial state and G is the goal state



Code:

```
import copy


class MyShortestPath:

    def __init__(self, map, startCity, goalCity):

        MyShortestPath.map=map

        self.currentCity=startCity

        self.goalCity=goalCity

        self.visitedList=[]

        self.cost=0

        self.visitedList=[]

        self.visitedList.append(self.currentCity)
```

```python
        self.prevState=None

    def displayState(self):
        print("--------------------------------")
        print(f"Current city:{self.currentCity}     Visited cities={self.visitedList}     Cost={self.cost}")

    def __gt__(self, other):
        return self.cost>other.cost

    def __lt__(self, other):
        return self.cost<other.cost

    def __eq__(self, other):
        return self.visitedList==other.visitedList

    def isGoalReached(self):
        if self.goalCity in self.visitedList:
            return True
        else:
            return False
```

```
    def move(self, city):

        if city!=self.currentCity and city not in self.visitedList and
MyShortestPath.map[self.currentCity][city]!=0:

            print(f"Moving from city {self.currentCity} to {city}")

            self.cost+=MyShortestPath.map[self.currentCity][city]

            self.currentCity=city

            self.visitedList.append(self.currentCity)

            return True

        else:

            print("Already visited")

            return False



    def possibleNextStates(self):

        stateList=[]

        for i in range(0, len(MyShortestPath.map[0])):

            state=copy.deepcopy(self)

            if state.move(i):

                self.prevState=copy.deepcopy(self)

                stateList.append(state)

        return stateList
```

```
def constructPath(goalState):
    print("The solution path from Goal to Start")
    while goalState is not None:
        goalState.displayState()
        goalState=goalState.prevState


open=[]
closed=[]
def UCS(state):
    open.append(state)
    while(open):
        thisState=open.pop(0)
        thisState.displayState()
        if thisState not in closed:
            closed.append(thisState)
            if thisState.isGoalReached():
                print("Goal state found.. stopping search")
                constructPath(thisState)
                break
            else:
                nextStates=thisState.possibleNextStates()
                for eachState in nextStates:
```

```
                    if eachState not in open and eachState not in
closed:
                        open.append(eachState)
                        open.sort()
                    elif eachState in open:
                        index=open.index(eachState)
                        if open[index].cost>eachState.cost:
                            open.pop(index)
                            open.append(eachState)
                            open.sort()
                    elif eachState in closed:
                        index=closed.index(eachState)
                        if closed[index].cost>eachState.cost:
                            closed.pop(index)
                            closed.append(eachState)
                            propogateImprovement(eachState)


def propogateImprovement(state):
    nextStates=state.possibleNextStates()
    for eachState in nextStates:
        if eachState in open:
            index=open.index[eachState]
            if open[index].cost>eachState.cost:
```

```
                open.pop(index)

                open.append(eachState)

                open.sort()

            if eachState in closed:

                index=closed.index(eachState)

                if closed[index].cost>eachState.cost:

                    closed.pop(index)

                    closed.append(eachState)

                    propogateImprovement(eachState)



map=[[0, 1, 5, 15, 0], [1, 0, 0, 0, 10], [5, 0, 0, 0, 5], [15, 0, 0,
0, 5], [0, 10, 5, 5, 0]]

start=0

goal=4

problem=MyShortestPath(map, start, goal)

UCS(problem)
```

Output:

```
--------------------------------
Current city:0     Visited cities=[0]     Cost=0
Already visited
Moving from city 0 to 1
Moving from city 0 to 2
Moving from city 0 to 3
Already visited
--------------------------------
Current city:1     Visited cities=[0, 1]     Cost=1
Already visited
Already visited
Already visited
Already visited
Moving from city 1 to 4
--------------------------------
Current city:2     Visited cities=[0, 2]     Cost=5
Already visited
Already visited
Already visited
Already visited
Moving from city 2 to 4
--------------------------------
Current city:4     Visited cities=[0, 2, 4]     Cost=10
Goal state found.. stopping search
The solution path from Goal to Start
--------------------------------
Current city:4     Visited cities=[0, 2, 4]     Cost=10
--------------------------------
Current city:0     Visited cities=[0]     Cost=0


...Program finished with exit code 0
Press ENTER to exit console.
```