AI

ASSIGNMENT 3

Panshul Saxena

CS10

102196006

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

(i) Euclidean Distance

(ii) Manhattan Distance

(iii) Minkowski Distance



**CODE:**

import sys # It lets us access system-specific parameters and functions

import copy

import math

import numpy as np

def panshul\_saxena(s,elem):  #finds the position of a certain element in the state s

    for i in range(len(s)):

        for j in range(len(s[0])):

            if s[i][j] == elem:

                return [i,j]

    return -1

def eucledian(s,g):

    res\_mat = np.zeros(len(s)\*len(s[0]),dtype = float) #create a matrix of data tyoe float to hold decimal value

    res\_mat = res\_mat.reshape(len(s),len(s))    #reshape it to a size of n\*n where n is size of start state matrix

    for x1 in range(len(s)):

        for y1 in range(len(s[0])):

            elem = s[x1][y1]

            pos = panshul\_saxena(g,elem)

            x2 = pos[0]

            y2 = pos[1]

            res\_mat[x1][y1] = math.sqrt((x2-x1)\*\*2 + (y2-y1)\*\*2)  #calculates the Euclidean distance between the same element in start and goal matrix

    ps\_102016114 = 0

    for i in range(len(res\_mat)):

        ps\_102016114 += sum(res\_mat[i])

    return ps\_102016114  #return the sum of Euclidean distance for every element in the start state

def manhattan(s,g):

    res\_mat = np.zeros(len(s)\*len(s[0]),dtype = float) #create a matrix of data type float to hold decimal value

    res\_mat = res\_mat.reshape(len(s),len(s)) #reshape it to a size of n\*n where n is size of start state matrix

    for x1 in range(len(s)):

        for y1 in range(len(s[0])):

            elem = s[x1][y1]

            pos = panshul\_saxena(g,elem)

            x2 = pos[0]

            y2 = pos[1]

            res\_mat[x1][y1] = abs(x2-x1) + abs(y2-y1)  #calculates the Manhattan distance between the same element in start and goal matrix

    ps\_102016114 = 0

    for i in range(len(res\_mat)):

        ps\_102016114 += sum(res\_mat[i])

    return ps\_102016114 #return the sum of Manhattan distance for every element in the start state

def minkowiski(s,g,p):

    res\_mat = np.zeros(len(s)\*len(s[0]),dtype = float) #create a matrix of data type float to hold decimal value

    res\_mat = res\_mat.reshape(len(s),len(s)) #reshape it to a size of n\*n where n is size of start state matrix

    for x1 in range(len(s)):

        for y1 in range(len(s[0])):

            elem = s[x1][y1]

            pos = panshul\_saxena(g,elem)

            x2 = pos[0]

            y2 = pos[1]

            res\_mat[x1][y1] = ((abs(x2-x1)\*\*p) + (abs(y2-y1)\*\*p))\*\*(1./p) #calculates the Minkowiski distance between the same element in start and goal matrix

    ps\_102016114 = 0

    for i in range(len(res\_mat)):

        ps\_102016114 += sum(res\_mat[i])

    return ps\_102016114    #return the sum of Minkowiski distance for every element in the start state

def main():

    p\_val = 3

    s0 = [[2,0,3],[1,8,4],[7,6,5]] #start state

    g = [[1,2,3],[8,0,4],[7,6,5]]  #end state

    euc = eucledian(s0,g) #function to calculate the Euclidean distance

    man = manhattan(s0,g)  #function to calculate the Manhattan distance

    mink = minkowiski(s0,g,p\_val) #function to calculate Minkowiski distance

   #print the respective distances ->

    print(euc)

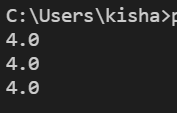
    print(man)

    print(mink)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**OUTPUT:**

****

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 **Best First Search** algorithm.



**CODE:**

import sys # It lets us access system-specific parameters and functions

import copy

q = []

ps\_102016114 = []

def compare(s,g):  #compares the current state with goal state

    if s==g:

        return(1)

    else:

        return(0)

def panshul\_saxena(s): #finds position of the empty cell

    for i in range(3):

        for j in range(3):

            if s[i][j] == 0:

                return([i,j])

def up(s,pos): #moves the empty cell up

    i = pos[0]

    j = pos[1]

    if i > 0:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i-1][j]

        temp[i-1][j] = 0

        return (temp)

    else:

        return (s)

def down(s,pos): #moves the empty cell down

    i = pos[0]

    j = pos[1]

    if i < 2:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i+1][j]

        temp[i+1][j] = 0

        return (temp)

    else:

        return (s)

def right(s,pos): #moves the empty cell right

    i = pos[0]

    j = pos[1]

    if j < 2:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i][j+1]

        temp[i][j+1] = 0

        return (temp)

    else:

        return (s)

def left(s,pos): #moves the empty cell left

    i = pos[0]

    j = pos[1]

    if j > 0:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i][j-1]

        temp[i][j-1] = 0

        return (temp)

    else:

        return (s)

def enqueue(s,val): #adds specific state in queue with its heuristic value

    global q

    q = q + [(val,s)]

def heuristic(s,g):   # the heuristic value  is calculated according to the given rule

    d = 0

    for i in range(3):

        for j in range(3):

            if s[i][j] != g[i][j]:

                d += 1

    return d

def dequeue(g):

    # this function actually works on a priority basis where we take the state with minimum cost as the next state to be explored

    global q

    global ps\_102016114

    q.sort()

    ps\_102016114 = ps\_102016114 + [q[0][1]]

    elem = q[0][1]

    del q[0]

    return (elem)

def search(s,g):

    curr\_state = copy.deepcopy(s)  #curr\_state created for a state s

    if s == g:   #if goal state reached

        return

    global ps\_102016114

    while(1):

        #Inside the while loop we perform same set of operations to reach the goal state by moving the zero up down left and right

        pos = panshul\_saxena(curr\_state) # checks for position of empty cell

        new = up(curr\_state,pos)

        if new != curr\_state: # checks if new state not equal to current state

            if new == g: #if new state is goal state

                print ("found!! The intermediate states are:")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114: #new state not in ps\_102016114 list then add it

                    enqueue(new,heuristic(new,g))  # new state added in queue with heuristic value

#performing the same operations for down, left and right as well

        new = down(curr\_state,pos)

        if new != curr\_state:

            if new == g:

                print ("found!! The intermediate states are:")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114:

                    enqueue(new,heuristic(new,g))

        new = right(curr\_state,pos)

        if new != curr\_state:

            if new == g:

                print ("found!! The intermediate states are:")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114:

                    enqueue(new,heuristic(new,g))

        new = left(curr\_state,pos)

        if new != curr\_state:

            if new == g:

                print ("found!! The intermediate states are:")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114:

                    enqueue(new,heuristic(new,g))

        if len(q) > 0:  # if q not empty then choose the current state as the state with minimum heuristic value present in q

            curr\_state = dequeue(g)

        else:  # if q is empty ad we do not reach the goal state => no solution exists..

            print ("not found")

            return

def main():

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

    g = [[1,2,3],[8,0,4],[7,6,5]]   #end state

    global ps\_102016114

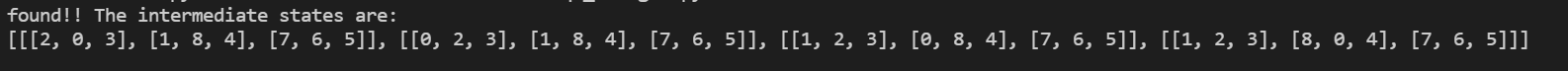
    ps\_102016114 = ps\_102016114 + [s] # global ps\_102016114 list updated with start state of the problem

    search(s,g)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**OUTPUT:**



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**.



**CODE:**

import sys # It lets us access system-specific parameters and functions

import copy

curr\_min = sys.maxsize # set minimum as the max value

q = []

ps\_102016114 = []

def compare(s,g): #compares the current state with goal state

    if s==g:

        return(1)

    else:

        return(0)

def panshul\_saxena(s): #finds position of the empty cell

    for i in range(len(s)):

        for j in range(len(s[0])):

            if s[i][j] == 0:

                return([i,j])

def up(s,pos): #moves the empty cell up

    i = pos[0]

    j = pos[1]

    if i > 0:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i-1][j]

        temp[i-1][j] = 0

        return (temp)

    else:

        return (s)

def down(s,pos): #moves the empty cell down

    i = pos[0]

    j = pos[1]

    if i < 2:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i+1][j]

        temp[i+1][j] = 0

        return (temp)

    else:

        return (s)

def right(s,pos): #moves the empty cell to its right

    i = pos[0]

    j = pos[1]

    if j < 2:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i][j+1]

        temp[i][j+1] = 0

        return (temp)

    else:

        return (s)

def left(s,pos): #moves the empty cell to its left

    i = pos[0]

    j = pos[1]

    if j > 0:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i][j-1]

        temp[i][j-1] = 0

        return (temp)

    else:

        return (s)

def enqueue(s): #adds specific state in queue

    global q

    q = q + [s]

def heuristic(s,g): #heuristic calculated as the number of misplaced cells

    d = 0

    for i in range(len(s)):

        for j in range(len(s[0])):

            if s[i][j] != g[i][j]:

                d += 1

    return d

def dequeue(g):  #we perform the dequeue (selection of the next state) with hill climb method

    h = []

    global q

    global ps\_102016114

    global curr\_min

    for i in range(len(q)):

        h = h + [heuristic(q[i],g)]  #add heuristic value of each state present in q in h

    if min(h) < curr\_min:  #check for the minimum value in h

        curr\_min = min(h)  #set the curr\_min to that minimum value

        index = h.index(min(h))  #get the index for the minimum value

        ps\_102016114 = ps\_102016114 + [q[index]] #add that state at index in ps\_102016114 list

    else:

        #if current minimum is the smallest

        print ("optimal solution found !! The intermediate states are: ")

        print (ps\_102016114)

        exit()

    elem = q[index]

    q = []

    return (elem)

def search(s,g):

    curr\_state = copy.deepcopy(s)

    if s == g:

        return

    global ps\_102016114

    while(1):

        pos = panshul\_saxena(curr\_state)

        new = up(curr\_state,pos)

        if new != curr\_state:# checks if new state not equal to current state

            if new == g:#if new state is goal state

                print ("Goal State found !! The intermediate States are :")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114: #new state not in ps\_102016114 list then add it

                    enqueue(new) # new state added in queue

            #Performing the same operations for down, left and right as well

        new = down(curr\_state,pos)

        if new != curr\_state:

            if new == g:

                print ("Goal State found !! The intermediate States are :")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114:

                    enqueue(new)

        new = right(curr\_state,pos)

        if new != curr\_state:

            if new == g:

                print ("Goal State found !! The intermediate States are :")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114:

                    enqueue(new)

        new = left(curr\_state,pos)

        if new != curr\_state:

            if new == g:

                print ("Goal State found !! The intermediate States are :")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114:

                    enqueue(new)

        if len(q) > 0: # if q not empty then choose the current state as the state with minimum heuristic value present in q

            curr\_state = dequeue(g)

        else: # if q is empty and we do not reach the goal state => no solution exists

            print ("not found")

            return

def main():

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

    g = [[1,2,7],[8,0,4],[7,6,5]] #end state

    global q

    global ps\_102016114

    q = q + [s] #added the start state in the q list

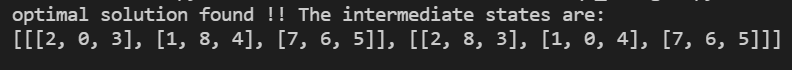
    ps\_102016114 = ps\_102016114 + [s] #added the start state in the ps\_102016114 list

    search(s,g)

if \_\_name\_\_ == "\_\_main\_\_":

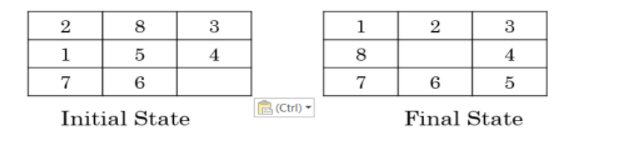
    main()

**OUTPUT:**



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**

****

**CODE:**

import copy

import sys # It lets us access system-specific parameters and functions

from heuristic import \*

q = []

ps\_102016114 = []

def compare(s,g): #compares the current state with goal state

    if s==g:

        return(1)

    else:

        return(0)

def panshul\_saxena(s): #finds position of the empty cell

    for i in range(3):

        for j in range(3):

            if s[i][j] == 0:

                return([i,j])

def up(s,pos): #moves the empty cell up

    i = pos[0]

    j = pos[1]

    if i > 0:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i-1][j]

        temp[i-1][j] = 0

        return (temp)

    else:

        return (s)

def down(s,pos): #moves the empty cell down

    i = pos[0]

    j = pos[1]

    if i < 2:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i+1][j]

        temp[i+1][j] = 0

        return (temp)

    else:

        return (s)

def right(s,pos): #moves the empty cell right

    i = pos[0]

    j = pos[1]

    if j < 2:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i][j+1]

        temp[i][j+1] = 0

        return (temp)

    else:

        return (s)

def left(s,pos): #moves the empty cell left

    i = pos[0]

    j = pos[1]

    if j > 0:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i][j-1]

        temp[i][j-1] = 0

        return (temp)

    else:

        return (s)

def enqueue(s,val): #adds specific state in queue with its heuristic value

    global q

    q = q + [(val,s)]

def heuristic(s,g):

    d = manhattan(s,g) #calculates the Manhattan value between current state and the goal state

    return d

def dequeue(g):

     # this function actually works on a priority basis where we take the state with minimum cost as the next state to be explored

    global q

    global ps\_102016114

    q.sort()

    ps\_102016114 = ps\_102016114 + [q[0][1]]

    elem = q[0][1]

    del q[0]

    return (elem)

def search(s,g):

    curr\_state = copy.deepcopy(s)

    if s == g:

        return

    global ps\_102016114

    while(1):

        pos = panshul\_saxena(curr\_state)

        new = up(curr\_state,pos)

        if new != curr\_state: # checks if new state not equal to current state

            if new == g: #if new state is goal state

                print ("found!! The intermediate states are:")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114: #new state not in ps\_102016114 list then add it

                    enqueue(new,heuristic(new,g))  # new state added in queue along with heuristic value

 #performing the same operations for down, left and right as well

        new = down(curr\_state,pos)

        if new != curr\_state:

            if new == g:

                print ("found!! The intermediate states are:")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114:

                    enqueue(new,heuristic(new,g))

        new = right(curr\_state,pos)

        if new != curr\_state:

            if new == g:

                print ("found!! The intermediate states are:")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114:

                    enqueue(new,heuristic(new,g))

        new = left(curr\_state,pos)

        if new != curr\_state:

            if new == g:

                print ("found!! The intermediate states are:")

                print (ps\_102016114 + [g])

                return

            else:

                if new not in ps\_102016114:

                    enqueue(new,heuristic(new,g))

        if len(q) > 0: # if q not empty then choose the current state as the state with minimum heuristic value present in q

            curr\_state = dequeue(g)

        else: # if q is empty and we do not reach the goal state => no solution exists

            print ("not found")

            return

def main():

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

    g = [[1,2,3],[8,0,4],[7,6,5]] #end state

    global q

    global ps\_102016114

    q = q

    ps\_102016114 = ps\_102016114 + [s] # global ps\_102016114 list updated with start state of the problem

    search(s,g)

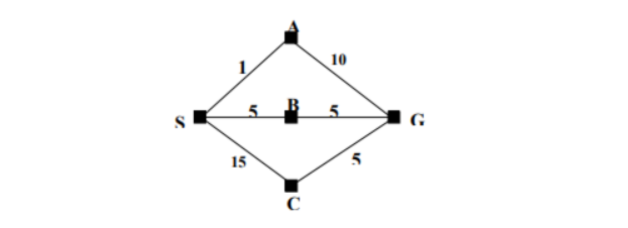
if \_\_name\_\_ == "\_\_main\_\_":

    main()

**OUTPUT:**

Taking too long

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

****

**CODE:**

import copy

class PanshulSaxena:

    def \_\_init\_\_(self, map, startCity, goalCity):

        PanshulSaxena.map=map

        self.currentCity=startCity

        self.goalCity=goalCity

        self.visitedList=[]

        self.cost=0

        self.visitedList=[]

        self.visitedList.append(self.currentCity)

        self.prevState=None

    def ps\_102016114(self):    # The current state is displayed

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

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

    def \_\_gt\_\_(self, other):       # returns true if the cost to reach the current node > other node

        return self.cost>other.cost

    def \_\_lt\_\_(self, other):       # returns true if the cost to reach the current node > other node

        return self.cost<other.cost

    def \_\_eq\_\_(self, other):             # returns true if the current list = other list

        return self.visitedList==other.visitedList

    def isGoalReached(self):    # returns true if the goal city is in visited list

        if self.goalCity in self.visitedList:

            return True

        else:

            return False

    def move(self, city):

        # if the city is not visited but still the city lies in map for the current city

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

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

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

            self.currentCity=city

            self.visitedList.append(self.currentCity)

            return True

        else:   #if city is visited

            print("Already visited")

            return False

    def possibleNextStates(self):   # Used to define the next possible state

        stateList=[]

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

            state=copy.deepcopy(self)   # deepcopy is used so that any changes made in the copy object is not reflected in main object

            if state.move(i):

                self.prevState=copy.deepcopy(self)

                stateList.append(state)

        return stateList

def constructPath(goalState):  # constucts path from start node to end node

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

    while goalState is not None:

        goalState.ps\_102016114()

        goalState=goalState.prevState

open=[]

closed=[]

def UCS(state):

    open.append(state)

    # perfoming the queue operation wherein we make use of priority queue in open and add states in closed respectively

    while(open):

        thisState=open.pop(0)

        thisState.ps\_102016114()

        if thisState not in closed:   # if current state not in closed then we add it to closed state

            closed.append(thisState)

            if thisState.isGoalReached():    # checks for each state if goal node is present or not

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

                constructPath(thisState)

                break

            else:

                nextStates=thisState.possibleNextStates()   #for every state we define the possible next states

                for eachState in nextStates:

                    if eachState not in open and eachState not in closed:  #each state not in both lists

                        open.append(eachState)

                        open.sort()

                    elif eachState in open:

                        index=open.index(eachState)  #determines the position of eachState

                        if open[index].cost>eachState.cost:   #checks if cost to reach state from beginning > cost

                            open.pop(index)

                            open.append(eachState)

                            open.sort()

                    elif eachState in closed:

                        index=closed.index(eachState)   #determines the position of eachState

                        if closed[index].cost>eachState.cost:  #checks if cost to reach state from beginning > cost

                            closed.pop(index)

                            closed.append(eachState)

                            propogateImprovement(eachState)

def propogateImprovement(state):   #recursively determines the best next state for the current state in the closed list

    # also updates the current open and closed lists for future process

    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]]  # respective distances are witten here

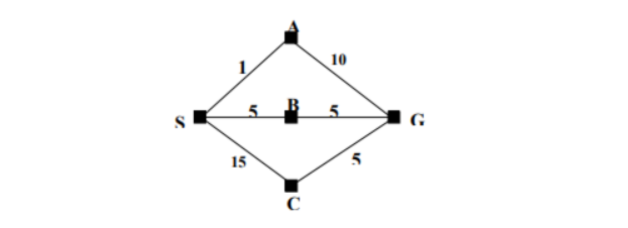
# The start and end nodes are defined and function is called ->

start=0

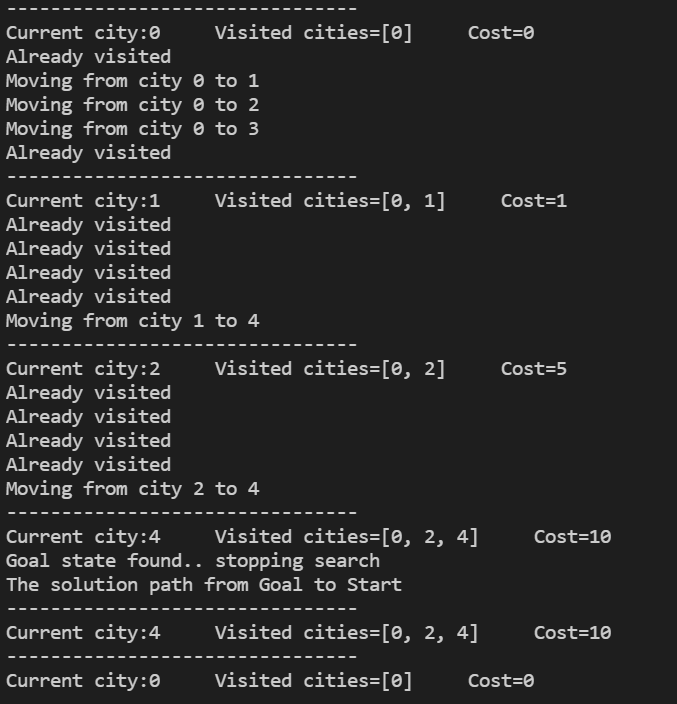
goal=4

problem=PanshulSaxena(map, start, goal)

UCS(problem)

****

**OUTPUT:**

****

**END**