AI

ASSIGNMENT 3

Kishan Pandey

CS12

102016114

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

import copy

import math

import numpy as np

def find\_pos(s,elem):

    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)

    res\_mat = res\_mat.reshape(len(s),len(s))

    for x1 in range(len(s)):

        for y1 in range(len(s[0])):

            elem = s[x1][y1]

            pos = find\_pos(g,elem)

            x2 = pos[0]

            y2 = pos[1]

            res\_mat[x1][y1] = math.sqrt((x2-x1)\*\*2 + (y2-y1)\*\*2)

    summ = 0

    for i in range(len(res\_mat)):

        summ += sum(res\_mat[i])

    return summ

def manhattan(s,g):

    res\_mat = np.zeros(len(s)\*len(s[0]),dtype = float)

    res\_mat = res\_mat.reshape(len(s),len(s))

    for x1 in range(len(s)):

        for y1 in range(len(s[0])):

            elem = s[x1][y1]

            pos = find\_pos(g,elem)

            x2 = pos[0]

            y2 = pos[1]

            res\_mat[x1][y1] = abs(x2-x1) + abs(y2-y1)

    summ = 0

    for i in range(len(res\_mat)):

        summ += sum(res\_mat[i])

    return summ

def minkowiski(s,g,p):

    res\_mat = np.zeros(len(s)\*len(s[0]),dtype = float)

    res\_mat = res\_mat.reshape(len(s),len(s))

    for x1 in range(len(s)):

        for y1 in range(len(s[0])):

            elem = s[x1][y1]

            pos = find\_pos(g,elem)

            x2 = pos[0]

            y2 = pos[1]

            res\_mat[x1][y1] = ((abs(x2-x1)\*\*p) + (abs(y2-y1)\*\*p))\*\*(1./p)

    summ = 0

    for i in range(len(res\_mat)):

        summ += sum(res\_mat[i])

    return summ

def main():

    p\_val = 3

    s0 = [[2,0,3],[1,8,4],[7,6,5]]

    g = [[1,2,3],[8,0,4],[7,6,5]]

    euc = eucledian(s0,g)

    man = manhattan(s0,g)

    mink = minkowiski(s0,g,p\_val)

    print(euc)

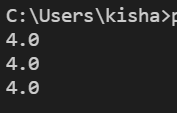
    print(man)

    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

import copy

q = []

visited = []

def compare(s,g):

    if s==g:

        return(1)

    else:

        return(0)

def find\_pos(s):

    for i in range(3):

        for j in range(3):

            if s[i][j] == 0:

                return([i,j])

def up(s,pos):

    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):

    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):

    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):

    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):

    global q

    q = q + [(val,s)]

def heuristic(s,g):

    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):

    global q

    global visited

    q.sort()

    visited = visited + [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 visited

    while(1):

        pos = find\_pos(curr\_state)

        new = up(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new,heuristic(new,g))

        new = down(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new,heuristic(new,g))

        new = right(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new,heuristic(new,g))

        new = left(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new,heuristic(new,g))

        if len(q) > 0:

            curr\_state = dequeue(g)

        else:

            print ("not found")

            return

def main():

    s = [[2,0,3],[1,8,4],[7,6,5]]

    g = [[1,2,3],[8,0,4],[7,6,5]]

    global q

    global visited

    q = q

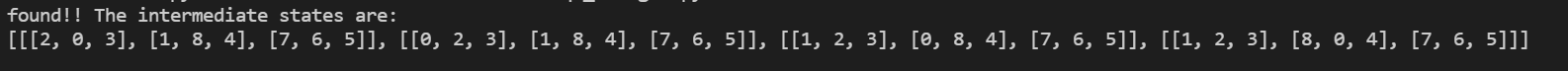
    visited = visited + [s]

    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:**

#Amardeep Singh

import sys

import copy

curr\_min = sys.maxsize

q = []

visited = []

def compare(s,g):

    if s==g:

        return(1)

    else:

        return(0)

def find\_pos(s):

    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):

    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):

    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):

    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):

    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):

    global q

    q = q + [s]

def heuristic(s,g):

    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):

    h = []

    global q

    global visited

    global curr\_min

    for i in range(len(q)):

        h = h + [heuristic(q[i],g)]

    if min(h) < curr\_min:

        curr\_min = min(h)

        index = h.index(min(h))

        visited = visited + [q[index]]

    else:

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

        print (visited)

        exit()

    elem = q[index]

    q = []

    return (elem)

def search(s,g):

    curr\_state = copy.deepcopy(s)

    if s == g:

        return

    global visited

    while(1):

        pos = find\_pos(curr\_state)

        new = up(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new)

        new = down(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new)

        new = right(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new)

        new = left(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new)

        if len(q) > 0:

            curr\_state = dequeue(g)

        else:

            print ("not found")

            return

def main():

    s = [[2,0,3],[1,8,4],[7,6,5]]

    g = [[1,2,7],[8,0,4],[7,6,5]]

    global q

    global visited

    q = q + [s]

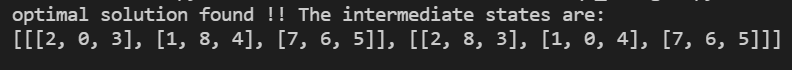
    visited = visited + [s]

    search(s,g)

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

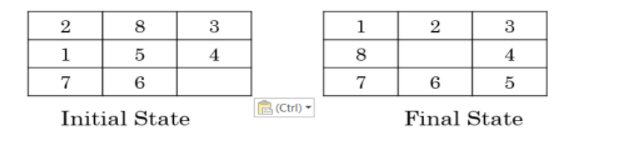
    main()

**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 sys

import copy

from heuristic import \*

q = []

visited = []

def compare(s,g):

    if s==g:

        return(1)

    else:

        return(0)

def find\_pos(s):

    for i in range(3):

        for j in range(3):

            if s[i][j] == 0:

                return([i,j])

def up(s,pos):

    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):

    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):

    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):

    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):

    global q

    q = q + [(val,s)]

def heuristic(s,g):

    d = manhattan(s,g)

    return d

def dequeue(g):

    global q

    global visited

    q.sort()

    visited = visited + [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 visited

    while(1):

        pos = find\_pos(curr\_state)

        new = up(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new,heuristic(new,g))

        new = down(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new,heuristic(new,g))

        new = right(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new,heuristic(new,g))

        new = left(curr\_state,pos)

        if new != curr\_state:

            if new == g:

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

                print (visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new,heuristic(new,g))

        if len(q) > 0:

            curr\_state = dequeue(g)

        else:

            print ("not found")

            return

def main():

    s = [[2,8,3],[1,5,4],[7,6,0]]

    g = [[1,2,3],[8,0,4],[7,6,5]]

    global q

    global visited

    q = q

    visited = visited + [s]

    search(s,g)

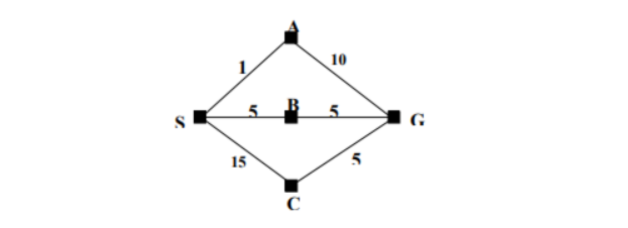
if \_\_name\_\_ == "\_\_main\_\_":

    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 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)

        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]]

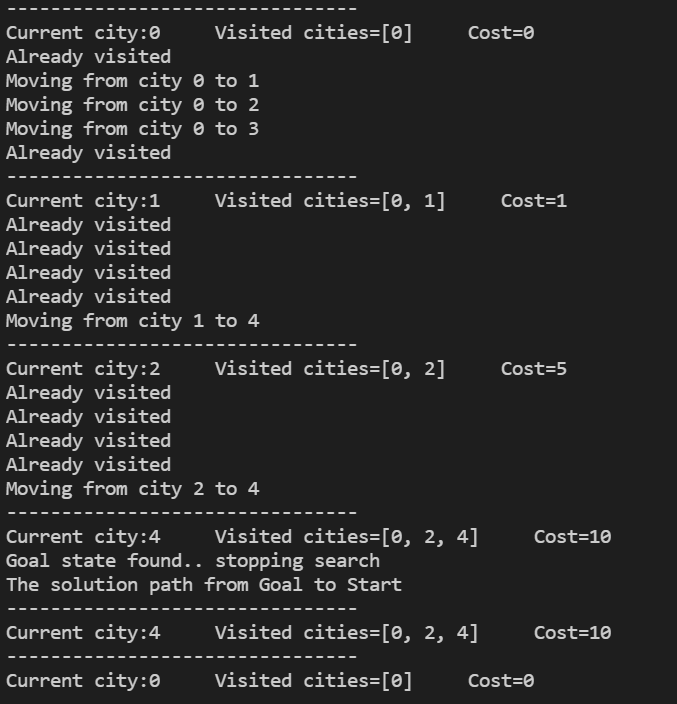
start=0

goal=4

problem=MyShortestPath(map, start, goal)

UCS(problem)

**OUTPUT:**

****

**END**