Assignment-1

Name: Karan Nasa

Roll: 102003008

Group: 2COE1

Q1 Find the sum of the series (1 + x + x 2 /2! + . . . + x n /n!). Number x and n should be entered at run time.

import math

x= int(input("Enter the value of x: "))

n= int(input("Enter the value of n: "))

Sum=1

if(n>1):

for i in range(1,n):

fact= 1

xpower = math.pow(x,i)

for j in range(1,i+2):

fact = fact\*j

fraction = xpower/fact

Sum=Sum + fraction

print("Sum is ",Sum)

Q2 WAP to create a list of 100 random numbers between 100 and 900. Count and print the: (i) All odd numbers (ii) All even numbers (iii) All prime numbers

import random

def randomlist(start, end, num):

res = []

for j in range(num):

res.append(random.randint(start, end))

return res

def main():

num=100

start=100

end=900

List=randomlist(start, end, num)

print("List is : \n")

for i in range(len(List)):

print(List[i],end=" ")

even=0

odd=0

prime=0

for i in range(len(List)):

if(List[i]%2==0):

even+=1

e.append(List[i])

else:

odd+=1

e.append(List[i])

flag=0

for j in range(2,List[i]):

if (List[i] % j== 0):

flag=1

break

if(flag==0):

prime+=1

p.append(List[i])

print("\nNo. of even nos.: ",even)

print("No. of odd nos.: ",odd)

print("No. of prime nos.: ",prime)

main()

Q3 Find the prime numbers between two given numbers.

num1=int(input("Enter start : "))

num2=int(input("Enter end : "))

prime=0

for i in range(num1,num2+1):

if(i==1):

continue

flag=0

for j in range(2,i):

if(i%j==0):

flag=1

break

if(flag==0):

prime+=1

print("No. of prime no. between ",num1," and ", num2, "is ",prime)

Q4 Find the common elements from two lists.

list1=[1,2,3,4,5,6]

list2=[5,6,7,8,9,10]

print("\nList 1 is: ")

for i in range(0,len(list1)):

print(list1[i],end=" ")

print("\nList 2 is: ")

for i in range(0,len(list2)):

print(list2[i],end=" ")

count=0

for i in range(0,len(list1)):

for j in range(0,len(list2)):

if(list1[i]==list2[j]):

count+=1

print("No. of same elements is ",count)

Q5 Print the leap years between any two years. The limit of the years should be entered at execution time.

print ("Print leap year between two given years")

startYear = int(input("Enter start year: "))

endYear = int(input("Enter last year: "))

print ("List of leap years:")

for year in range(startYear, endYear):

if (0 == year % 4) and (0 != year % 100) or (0 == year % 400):

print (year)

Q6 Write a Python Program to input basic salary of an employee and calculate its Gross salary according to following: Basic Salary <= 10000 : HRA = 20%, DA = 80% Basic Salary <= 20000 : HRA = 25%, DA = 90% Basic Salary > 20000 : HRA = 30%, DA = 95%.

def computeSalary( basic):

if(basic<=10000):

hra =(0.2 \* basic)

da = 0.8 \* basic

elif(basic<=20000):

hra = 0.25 \* basic

da = 0.9 \* basic

elif(basic>20000):

hra = 0.3 \* basic

da = 0.95 \* basic

gross = basic + hra + da

return gross

def main():

sal=int(input("Enter salary: "))

print("Gross salary is ",computeSalary(sal))

main()

Q7 Write a Python program to check the validity of password input by users. Validation: • At least 1 letter between [a-z] and 1 letter between [A-Z]. • At least 1 number between [0-9]. • At least 1 character from [$#@]. • Minimum length 6 characters. • Maximum length 16 characters.

import re

p= input("Input your password: ")

x = True

while x:

if (len(p)<6 or len(p)>16):

break

elif not re.search("[a-z]",p):

break

elif not re.search("[0-9]",p):

break

elif not re.search("[A-Z]",p):

break

elif not re.search("[$#@]",p):

break

elif re.search("\s",p):

break

else:

print("Valid Password")

x=False

break

if x:

print("Not a Valid Password")

Q8 Create a List L having data as= [10, 20, 30, 40, 50, 60, 70, 80]. (i) WAP to add 200 and 300 to L. (ii) WAP to remove 10 and 30 from L. (iii) WAP to sort L in ascending order. (iv) WAP to sort L in descending order.

L = [10,20,30,40,50,60,70,80]

def main():

print("List is: ")

print(L)

L.append(200)

L.append(300)

print("List is: ")

print(L)

L.remove(10)

L.remove(30)

print("List is: ")

print(L)

L.sort()

print("List is: ")

print(L)

L.reverse()

print("List is: ")

print(L)

main()

Q9) D is a dictionary defined as D= {1:”One”, 2:”Two”, 3:”Three”, 4: “Four”, 5:”Five”}. (i) WAP to add new entry in D; key=6 and value is “Six” (ii) WAP to remove key=2. (iii) WAP to check if 6 key is present in D. (iv) WAP to count the number of elements present in D. (v) WAP to add all the values present in D

def checkKey(dict, key):

if key in dict:

print("Present, ", end=" ")

print("value =", dict[key])

else:

print("Not present")

def main():

D = {1:'One', 2:'Two', 3:'Three', 4: 'Four', 5:'Five'}

D[6]='Six'

print(D)

D.pop(2)#popped using the key

print(D)

checkKey(D,6)

print(D)

count = 0

print("no of elements in dictionary:",len(D))

total=0

for key in D:

total=total+key

print(total)

main()

Q10 a) Write a function which takes principal amount, interest rate and time. This function returns compound interest. Call this function to print the output.

def compound\_interest(principle, rate, time):

Amount = principle \* (pow((1 + rate / 100), time))

print("The simple interest is:",simple\_interest)

CI = Amount - principle

print("Compound interest is ", CI)

amt=int(input("Enter amount: "))

rate=float(input("Enter rate: "))

time=int(input("Enter time: "))

compound\_interest(amt, rate, time)

b) Save this function (as a module) in a python file and call it in another python file.

from question10\_a import function

from Q10 import compound\_interest

amt=int(input("Enter amount: "))

rate=float(input("Enter rate: "))

time=int(input("Enter time: "))

compound\_interest(amt, rate, time)

Assignment-2

Name: Karan Nasa

Roll: 102003008

Group: 2COE1

Q1) Given two jugs- a 4 litre and 3 litre capacity. Neither has any measurable markers on it. There is a pump which can be used to fill the jugs with water. Simulate the procedure in Python to get exactly 2 litre of water into 4-litre jug

capacity=[4,3]

jug1=capacity[0]

jug2=capacity[1]

states=[]

def get\_all\_states(state):

    s1=state[0]

    s2=state[1]

    if(s1==2):

        states.append(state)

        return True

    if(s1==0 and s2==0):

        if(get\_all\_states((0,s1+3))):

            states.append(state)

            return True

    if(s1==0 and s2>0):

        if(get\_all\_states((s1+s2,0))):

            states.append(state)

            return True

    if(s1==3 and s2==0):

        states.append(state)

        return True

    if(s1==3 and s2==0):

        if(get\_all\_states((s1,s2+3))):

            states.append(state)

            return True

    if(s1==3 and s2==3):

        if(get\_all\_states((0,s2))):

            states.append(state)

            return True

    if(s1>3):

        if(get\_all\_states((0,s2))):

            states.append(state)

            return True

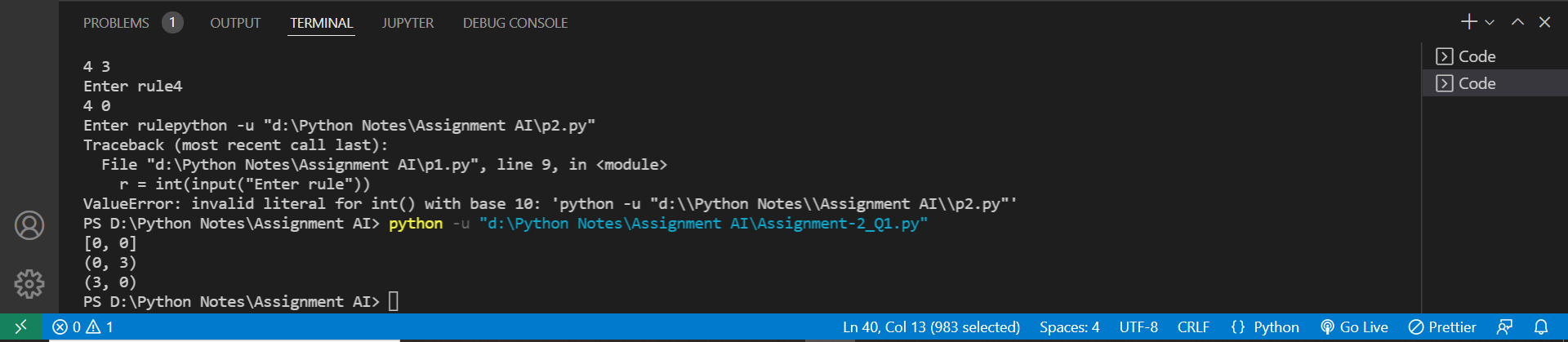
initial\_state=[0,0]

get\_all\_states(initial\_state)

states.reverse()

for i in states:

    print(i)



Q2) Given three jugs: 12, 8 and 5 liter capacities. Largest jug is completely filled. Using these 3 jugs, split the water to obtain exactly 6 liter in largest jugs.

from tokenize import Triple

capacity=(12,8,5)

x=capacity[0]

y=capacity[1]

z=capacity[2]

memory={}

ans=[]

def get\_all\_states(state):

    a=state[0]

    b=state[1]

    c=state[2]

    if(a==6 and b==6):

        ans.append(state)

        return True

    if((a,b,c) in memory):

        return False

    memory[(a,b,c)]=1

    if(a>0):

        if(a+b<=y):

            if(get\_all\_states((0,a+b,c))):

                ans.append(state)

                return True

        else:

            if(get\_all\_states((a-(y-b),y,c))):

                ans.append(state)

                return True

        if(a+c<=z):

            if(get\_all\_states((0,b,a+c))):

                ans.append(state)

                return True

        else:

            if(get\_all\_states((a-(z-c),b,z))):

                ans.append(state)

                return True

    if(b>0):

        if(a+b<=x):

            if(get\_all\_states((a+b,0,c))):

                ans.append(state)

                return True

        else:

            if(get\_all\_states((x,b-(x-a),c))):

                ans.append(state)

                return True

        if(b+c<=z):

            if(get\_all\_states((a,0,b+c))):

                ans.append(state)

                return True

        else:

            if(get\_all\_states((a,b-(z-c),z))):

                ans.append(state)

                return True

    if(c>0):

        if(a+c<=x):

            if(get\_all\_states((a+c,b,0))):

                ans.append(state)

                return True

        else:

            if(get\_all\_states((x,b,c-(x-a)))):

                ans.append(state)

                return True

        if(b+c<=y):

            if(get\_all\_states((a,b+c,0))):

                ans.append(state)

                return True

        else:

            if(get\_all\_states((a,y,c-(y-b)))):

                ans.append(state)

                return Triple

    return False

initial\_state=(12,0,0)

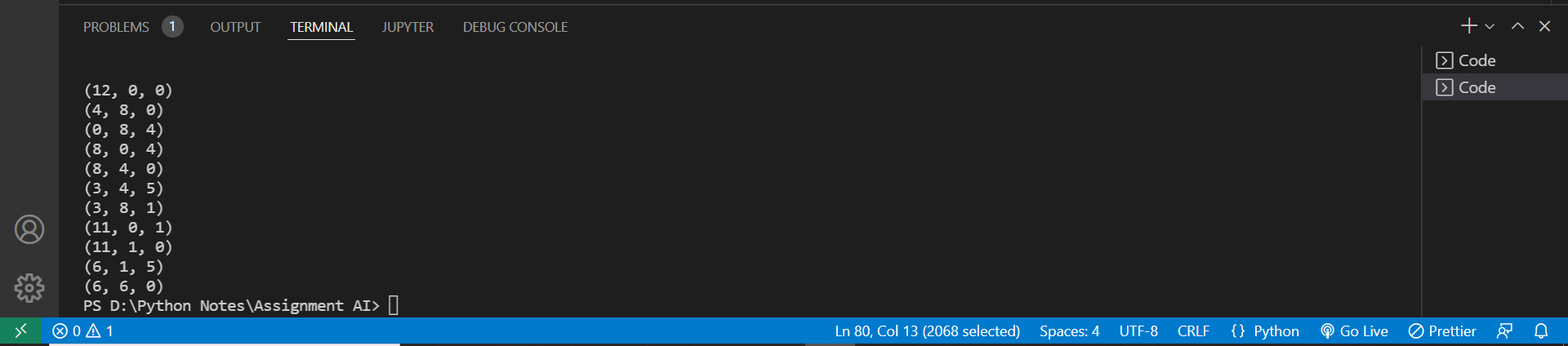
print("Starting work\n")

get\_all\_states(initial\_state)

ans.reverse()

for i in ans:

    print(i)



Q3) Write a code in python for the 8 puzzle problem by taking the following initial and final states

import copy

frontier = []

explored = []

def find\_pos(current):

    for i in range(3):

        for j in range(3):

            if current[i][j] == 0:

                r,c = i, j

                return r,c

def up(current, r, c):

    if r != 0:

        child = copy.deepcopy(current)

        child[r][c],child[r - 1][c] = (child[r - 1][c],0)

        if child not in frontier and child not in explored:

            frontier.append(child)

            return child

    return None

def down(current, r, c):

    if r != 2:

        child = copy.deepcopy(current)

        child[r][c], child[r+1][c] = (child[r+1][c],0)

        if child not in frontier and child not in explored:

            frontier.append(child)

            return child

    return None

def left(current, r, c):

    if c != 0:

        child = copy.deepcopy(current)

        child[r][c], child[r][c-1] = (child[r][c-1],0)

        if child not in frontier and child not in explored:

            frontier.append(child)

            return child

    return None

def right(current, r, c):

    if c != 2:

        child = copy.deepcopy(current)

        child[r][c], child[r][c+1] = (child[r][c+1],0)

        if child not in frontier and child not in explored:

            frontier.append(child)

            return child

    return None

def is\_goal(current, goal\_state):

    if current == goal\_state:

        return True

    else:

        return False

def search():

    initial\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

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

    frontier.append(initial\_state)

    flag = 1

    while flag:

        current = frontier.pop(0)

        if is\_goal(current,goal\_state):

            flag = 0

            print(len(explored))

        else:

            r,c = find\_pos(current)

            neighbor = [0] \* 4

            neighbor[0] = up(current,r,c)

            neighbor[1] = down(current,r,c)

            neighbor[2] = left(current,r,c)

            neighbor[3] = right(current,r,c)

            explored.append(current)

            for i in neighbor:

                if i != None and i != 0:

                    if is\_goal(i,goal\_state):

                        flag = 0

                        print("All states visited from start state to reach goal state:")

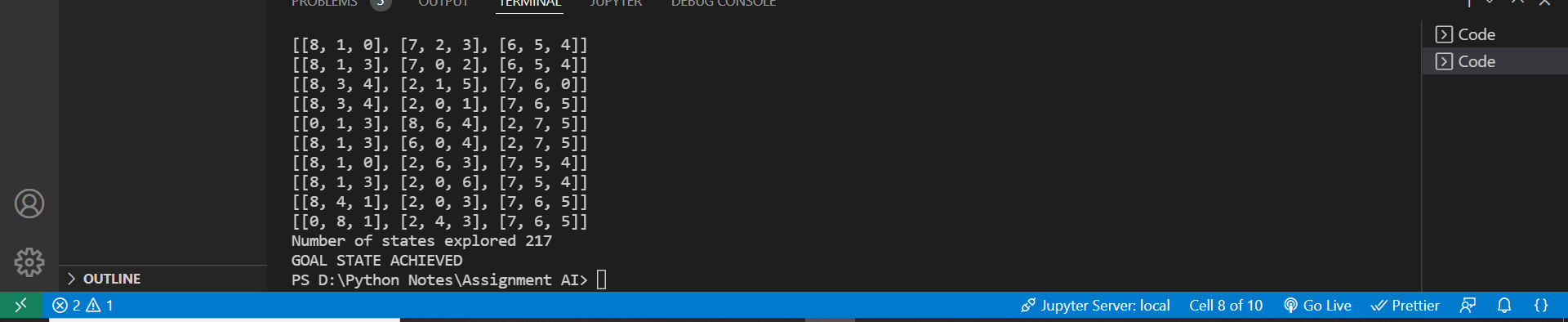
                        for state in explored:

                            print(state)

                        print("Number of states explored", len(explored))

                        print("GOAL STATE ACHIEVED")

search()



Q4) Write a Python program to implement Travelling Salesman Problem (TSP). Take the starting node from the user at run time.

V = 4

answer = []

def tsp(graph, v, currPos, n, count, cost):

    if (count == n and graph[currPos][0]):

        answer.append(cost + graph[currPos][0])

        return

    for i in range(n):

        if (v[i] == False and graph[currPos][i]):

            v[i] = True

            tsp(graph, v, i, n, count + 1,

                cost + graph[currPos][i])

            v[i] = False

if \_\_name\_\_ == '\_\_main\_\_':

    n = 4

    graph= [[ 0, 10, 15, 20 ],

            [ 10, 0, 35, 25 ],

            [ 15, 35, 0, 30 ],

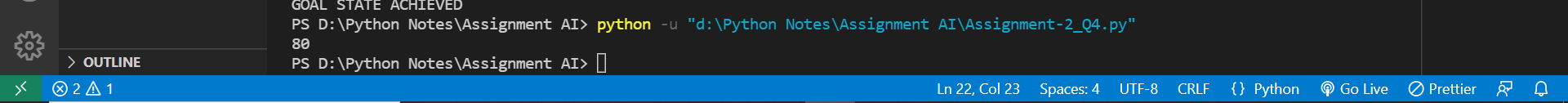
            [ 20, 25, 30, 0 ]]

    v = [False for i in range(n)]

    v[0] = True

    tsp(graph, v, 0, n, 1, 0)

    print(min(answer))



Assignment-3

Name: Karan Nasa

Roll: 102003008

Group: 2COE1

Q1) 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

statevector = []

import math

def search(mat, x):

    for i in range(len(mat)):

        row = mat[i]

        for j in range(len(row)):

            e = mat[i][j]

            if(e==x):

                return (i, j)

    return (-1, -1)

class Board:

    def \_\_init\_\_(self, board=[[0, 0, 0], [0, 0, 0], [0, 0, 0]]):

        self.board = board

        self.NOT\_VALID = False

        self.checkEmpty()

    def change(self, i, j, v):

        self.board[i, j] = v

    def checkEmpty(self):

        for i in range(len(self.board)):

            row = self.board[i]

            for j in range(len(row)):

                if(self.board[i][j] == -1):

                    self.eX = i

                    self.eY = j

                    return

    def verifyValid(self):

        self.checkEmpty()

        if(self.NOT\_VALID): return False

        if(self.eX < 0 or self.eX > 2 or self.eY < 0 or self.eY > 2): return False

        return True

    @staticmethod

    def hx(c, t):

        h = 0

        for i in range(len(c.board)):

            row = c.board[i]

            for j in range(len(row)):

                e1 = c.board[i][j]

                (ti, ty) = search(t.board, e1)

                h = h + math.sqrt(math.pow(ti - i, 2) + math.pow(ty - j, 2))

        return h

    @staticmethod

    def hxm(c, t): # Manhattan Distance

        h = 0

        for i in range(len(c.board)):

            row = c.board[i]

            for j in range(len(row)):

                e1 = c.board[i][j]

                (ti, ty) = search(t.board, e1)

                h = h + abs(ti - i) + abs(ty - j)

        return h

    @staticmethod

    def hxmz(c, t): # Minkowski Distance

        h = 0

        P = 2 # Minkowski parameter

        for i in range(len(c.board)):

            row = c.board[i]

            for j in range(len(row)):

                e1 = c.board[i][j]

                (ti, ty) = search(t.board, e1)

                h = h + math.pow(math.pow(ti - i, P) + math.pow(ty - j, P), 1/P)

        return h

    def swap(self, ix, iy, tx, ty):

        try:

            self.board[ix][iy], self.board[tx][ty] = self.board[tx][ty], self.board[ix][iy]

        except Exception as e:

            self.NOT\_VALID = True

    def copy(self):

        b = Board()

        b.board = [x[:] for x in self.board]

        b.eX = self.eX

        b.eY = self.eY

        return b

    def \_\_eq\_\_(self, t):

        return self.board == t.board

    def \_\_repr\_\_(self):

        s = "A simple board with state\n"

        for row in self.board:

            s += str(' '.join([str(i) for i in row]))

            s+="\n"

        s+="Empty spot at pos %d, %d"%(self.eX, self.eY)

        return s

class TNode:

    def \_\_init\_\_(self, init, childs=[]):

        self.parent = None

        self.h = None

        self.state = init

        self.childs = childs

    def walkAndH(self, fin):

        # Assume hole in center

        board = self.state

        self.childs = []

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX + 1, c.eY) ## Swap right

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX - 1, c.eY) ## Swap left

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX, c.eY + 1) ## Swap Up

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX, c.eY - 1) ## Swap Down

        if(c.verifyValid()):

            self.childAppend(c, fin)

    def childAppend(self, obj, fin):

        global statevector

        if(obj is not self.parent and obj not in statevector):

            statevector.append(obj) # Append into global state list ie CLOSED LIST

            t = TNode(obj)

            t.parent = self

            t.h = Board.hx(obj, fin)

            self.childs.append(t)

    def \_\_repr\_\_(self):

        s = "TNode with data: %s && Childs: %d"%(self.state, len(self.childs))

        return s

    def copy(self):

        t = TNode(self.state)

        return t

    def \_\_eq\_\_(self, other):

        return self.h == other.h

    def \_\_lt\_\_(self, other):

        return self.h < other.h

initialData = [[2, -1, 3], [1, 8, 4], [7, 6, 5]]

initBoard = Board(initialData)

finalData = [[1, 2, 3], [8, -1, 4], [7, 6, 5]]

finBoard = Board(finalData)

parent = TNode(initBoard)

q = [ parent ]

FOUND = False

fNode = None

while(q):

    e = q.pop(0)

    if(e.state == finBoard):

        FOUND = True

        fNode = e

        break

    e.walkAndH(finBoard)

    c = e.childs

    q.extend(c)

    q = sorted(q) # Sort and choose best elements

def rPrint(x):

    if(x is None): return

    rPrint(x.parent)

    print(x.state)

if(FOUND):

    print("found at")

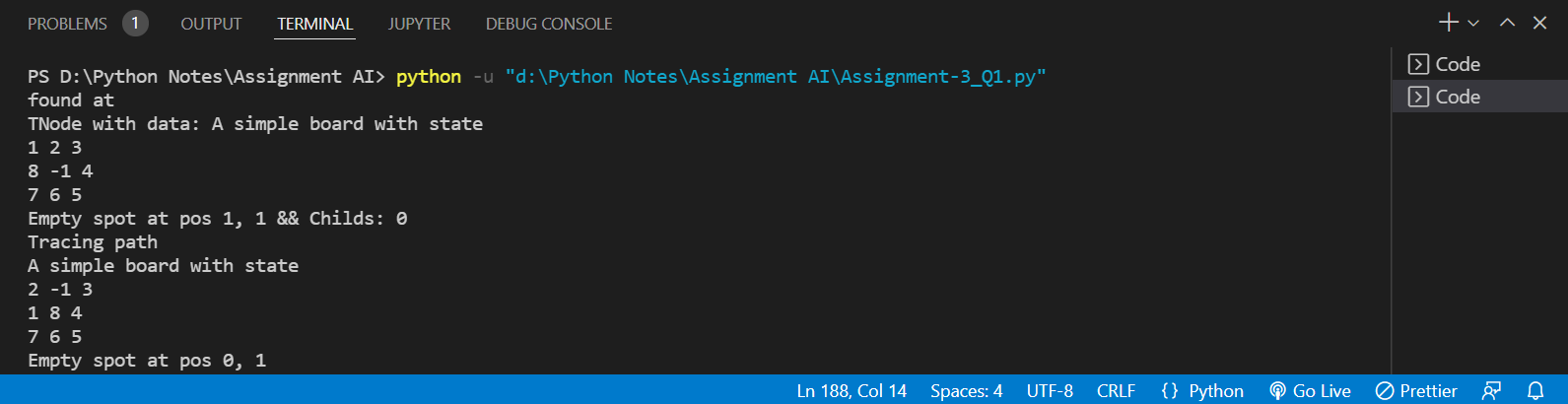
    print(fNode)

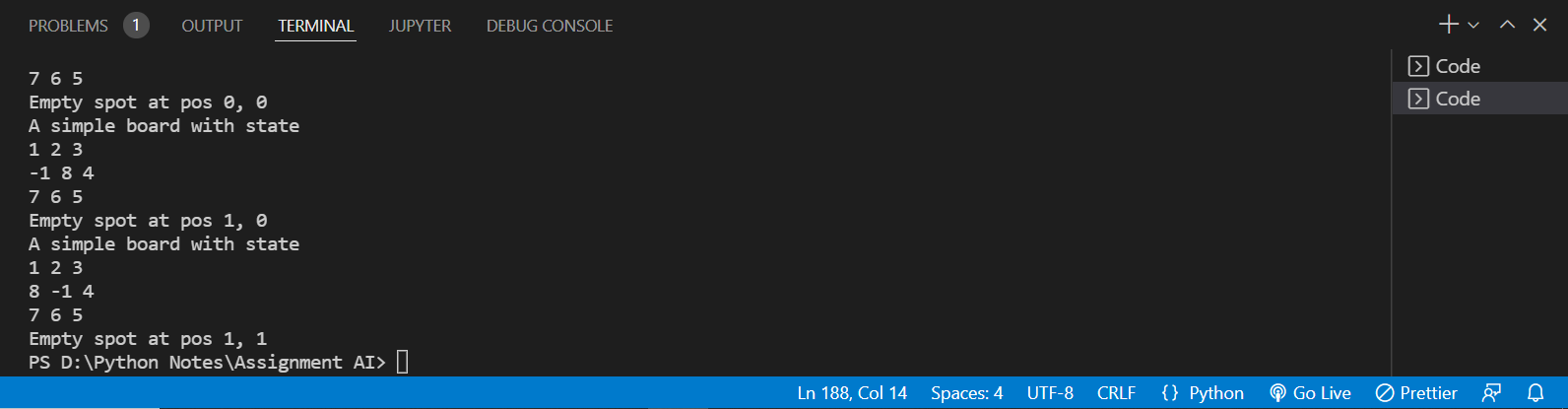
    print("Tracing path")

    x = fNode

    rPrint(x)

OUTPUT





Q2) 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.

## Schema, ie classes templates

statevector = []

import math

def search(mat, x):

    for i in range(len(mat)):

        row = mat[i]

        for j in range(len(row)):

            e = mat[i][j]

            if(e==x):

                return (i, j)

    return (-1, -1)

class Board:

    def \_\_init\_\_(self, board=[[0, 0, 0], [0, 0, 0], [0, 0, 0]]):

        self.board = board

        self.NOT\_VALID = False

        self.checkEmpty()

    def change(self, i, j, v):

        self.board[i, j] = v

    def checkEmpty(self):

        for i in range(len(self.board)):

            row = self.board[i]

            for j in range(len(row)):

                if(self.board[i][j] == -1):

                    self.eX = i

                    self.eY = j

                    return

    def verifyValid(self):

        self.checkEmpty()

        if(self.NOT\_VALID): return False

        if(self.eX < 0 or self.eX > 2 or self.eY < 0 or self.eY > 2): return False

        return True

    @staticmethod

    def hx(c, t): # Manhattan Distance

        h = 0

        for i in range(len(c.board)):

            row = c.board[i]

            for j in range(len(row)):

                e1 = c.board[i][j]

                (ti, ty) = search(t.board, e1)

                h = h + abs(ti - i) + abs(ty - j)

        return h

    def swap(self, ix, iy, tx, ty):

        try:

            self.board[ix][iy], self.board[tx][ty] = self.board[tx][ty], self.board[ix][iy]

        except Exception as e:

            self.NOT\_VALID = True

    def copy(self):

        b = Board()

        b.board = [x[:] for x in self.board]

        b.eX = self.eX

        b.eY = self.eY

        return b

    def \_\_eq\_\_(self, t):

        return self.board == t.board

    def \_\_repr\_\_(self):

        s = "A simple board with state\n"

        for row in self.board:

            s += str(' '.join([str(i) for i in row]))

            s+="\n"

        s+="Empty spot at pos %d, %d"%(self.eX, self.eY)

        return s

class TNode:

    def \_\_init\_\_(self, init, childs=[]):

        self.parent = None

        self.h = None

        self.state = init

        self.childs = childs

    def walkAndH(self, fin):

        # Assume hole in center

        board = self.state

        self.childs = []

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX + 1, c.eY) ## Swap right

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX - 1, c.eY) ## Swap left

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX, c.eY + 1) ## Swap Up

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX, c.eY - 1) ## Swap Down

        if(c.verifyValid()):

            self.childAppend(c, fin)

    def childAppend(self, obj, fin):

        global statevector

        if(obj is not self.parent and obj not in statevector):

            statevector.append(obj) # Append into global state list ie CLOSED LIST

            t = TNode(obj)

            t.parent = self

            t.h = Board.hx(obj, fin)

            self.childs.append(t)

    def \_\_repr\_\_(self):

        s = "TNode with data: %s && Childs: %d"%(self.state, len(self.childs))

        return s

    def copy(self):

        t = TNode(self.state)

        return t

    def \_\_eq\_\_(self, other):

        return self.h == other.h

    def \_\_lt\_\_(self, other):

        return self.h < other.h

initialData = [[2, -1, 3], [1, 8, 4], [7, 6, 5]]

initBoard = Board(initialData)

finalData = [[1, 2, 3], [8, -1, 4], [7, 6, 5]]

finBoard = Board(finalData)

parent = TNode(initBoard)

q = [ parent ]

FOUND = False

fNode = None

while(q):

    e = q.pop(0)

    if(e.state == finBoard):

        FOUND = True

        fNode = e

        break

    e.walkAndH(finBoard)

    c = e.childs

    q.extend(c)

    q = sorted(q) # Sort and choose best elements

def rPrint(x):

    if(x is None): return

    rPrint(x.parent)

    print(x.state)

if(FOUND):

    print("found at")

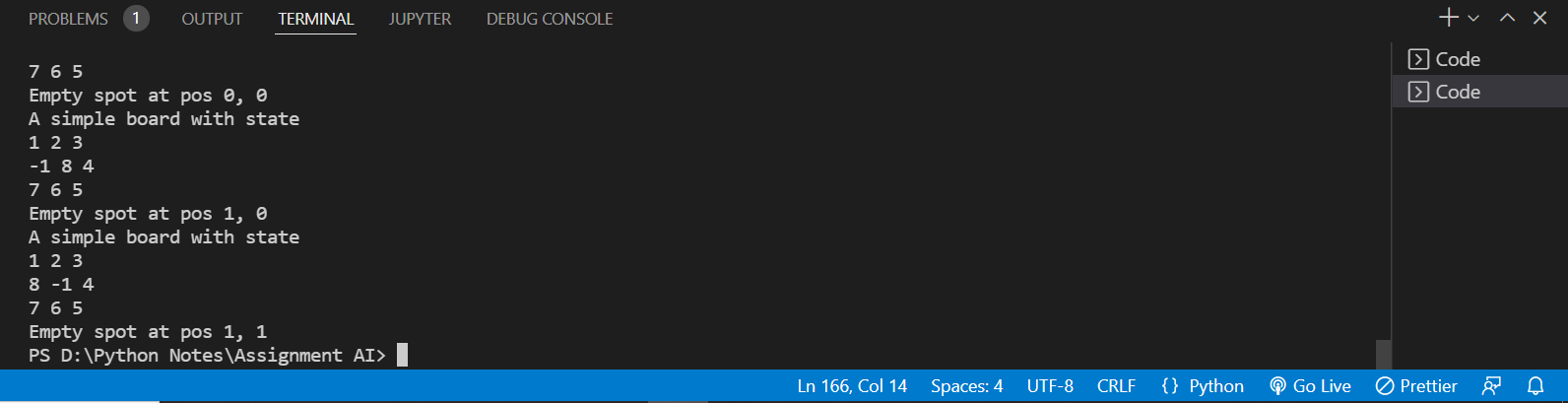
    print(fNode)

    print("Tracing path")

    x = fNode

    rPrint(x)

OUTPUT



Q3) 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.

## Schema, ie classes templates

statevector = []

import math

def search(mat, x):

    for i in range(len(mat)):

        row = mat[i]

        for j in range(len(row)):

            e = mat[i][j]

            if(e==x):

                return (i, j)

    return (-1, -1)

class Board:

    def \_\_init\_\_(self, board=[[0, 0, 0], [0, 0, 0], [0, 0, 0]]):

        self.board = board

        self.NOT\_VALID = False

        self.checkEmpty()

    def change(self, i, j, v):

        self.board[i, j] = v

    def checkEmpty(self):

        for i in range(len(self.board)):

            row = self.board[i]

            for j in range(len(row)):

                if(self.board[i][j] == -1):

                    self.eX = i

                    self.eY = j

                    return

    def verifyValid(self):

        self.checkEmpty()

        if(self.NOT\_VALID): return False

        if(self.eX < 0 or self.eX > 2 or self.eY < 0 or self.eY > 2): return False

        return True

    @staticmethod

    def hx(c, t): # Manhattan Distance

        h = 0

        for i in range(len(c.board)):

            row = c.board[i]

            for j in range(len(row)):

                e1 = c.board[i][j]

                (ti, ty) = search(t.board, e1)

                h = h + abs(ti - i) + abs(ty - j)

        return h

    def swap(self, ix, iy, tx, ty):

        try:

            self.board[ix][iy], self.board[tx][ty] = self.board[tx][ty], self.board[ix][iy]

        except Exception as e:

            self.NOT\_VALID = True

    def copy(self):

        b = Board()

        b.board = [x[:] for x in self.board]

        b.eX = self.eX

        b.eY = self.eY

        return b

    def \_\_eq\_\_(self, t):

        return self.board == t.board

    def \_\_repr\_\_(self):

        s = "A simple board with state\n"

        for row in self.board:

            s += str(' '.join([str(i) for i in row]))

            s+="\n"

        s+="Empty spot at pos %d, %d"%(self.eX, self.eY)

        return s

class TNode:

    def \_\_init\_\_(self, init, childs=[]):

        self.parent = None

        self.h = None

        self.state = init

        self.childs = childs

    def walkAndH(self, fin):

        # Assume hole in center

        board = self.state

        self.childs = []

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX + 1, c.eY) ## Swap right

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX - 1, c.eY) ## Swap left

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX, c.eY + 1) ## Swap Up

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX, c.eY - 1) ## Swap Down

        if(c.verifyValid()):

            self.childAppend(c, fin)

    def childAppend(self, obj, fin):

        global statevector

        if(obj is not self.parent and obj not in statevector):

            statevector.append(obj) # Append into global state list ie CLOSED LIST

            t = TNode(obj)

            t.parent = self

            t.h = Board.hx(obj, fin)

            self.childs.append(t)

    def \_\_repr\_\_(self):

        s = "TNode with data: %s && Childs: %d"%(self.state, len(self.childs))

        return s

    def copy(self):

        t = TNode(self.state)

        return t

    def \_\_eq\_\_(self, other):

        return self.h == other.h

    def \_\_lt\_\_(self, other):

        return self.h < other.h

initialData = [[2, -1, 3], [1, 8, 4], [7, 6, 5]]

initBoard = Board(initialData)

finalData = [[1, 2, 3], [8, -1, 4], [7, 6, 5]]

finBoard = Board(finalData)

parent = TNode(initBoard)

q = parent

FOUND = False

fNode = None

while(q):  # HILL CLIMB, ALWAYS GET BEST SOLN

    if(q.state == finBoard):

        FOUND = True

        fNode = q

        break

    if(not q): break

    q.walkAndH(finBoard)

    c = q.childs

    if(len(c)==0): break

    q = max(c, key=lambda item: item.h)

def rPrint(x):

    if(x is None): return

    rPrint(x.parent)

    print(x.state)

if(FOUND):

    print("found at")

    print(fNode)

    print("Tracing path")

    x = fNode

    rPrint(x)

Q4) 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

statevector = []

import math

def search(mat, x):

    for i in range(len(mat)):

        row = mat[i]

        for j in range(len(row)):

            e = mat[i][j]

            if(e==x):

                return (i, j)

    return (-1, -1)

class Board:

    def \_\_init\_\_(self, board=[[0, 0, 0], [0, 0, 0], [0, 0, 0]]):

        self.board = board

        self.NOT\_VALID = False

        self.checkEmpty()

    def change(self, i, j, v):

        self.board[i, j] = v

    def checkEmpty(self):

        for i in range(len(self.board)):

            row = self.board[i]

            for j in range(len(row)):

                if(self.board[i][j] == -1):

                    self.eX = i

                    self.eY = j

                    return

    def verifyValid(self):

        self.checkEmpty()

        if(self.NOT\_VALID): return False

        if(self.eX < 0 or self.eX > 2 or self.eY < 0 or self.eY > 2): return False

        return True

    @staticmethod

    def hx(c, t):

        h = 0

        for i in range(len(c.board)):

            row = c.board[i]

            for j in range(len(row)):

                e1 = c.board[i][j]

                (ti, ty) = search(t.board, e1)

                h = h + math.sqrt(math.pow(ti - i, 2) + math.pow(ty - j, 2))

        return h

    def swap(self, ix, iy, tx, ty):

        try:

            self.board[ix][iy], self.board[tx][ty] = self.board[tx][ty], self.board[ix][iy]

        except Exception as e:

            self.NOT\_VALID = True

    def copy(self):

        b = Board()

        b.board = [x[:] for x in self.board]

        b.eX = self.eX

        b.eY = self.eY

        return b

    def \_\_eq\_\_(self, t):

        return self.board == t.board

    def \_\_repr\_\_(self):

        s = "A simple board with state\n"

        for row in self.board:

            s += str(' '.join([str(i) for i in row]))

            s+="\n"

        s+="Empty spot at pos %d, %d"%(self.eX, self.eY)

        return s

class TNode:

    def \_\_init\_\_(self, init, childs=[]):

        self.parent = None

        self.h = None

        self.state = init

        self.childs = childs

    def walkAndH(self, fin):

        # Assume hole in center

        board = self.state

        self.childs = []

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX + 1, c.eY) ## Swap right

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX - 1, c.eY) ## Swap left

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX, c.eY + 1) ## Swap Up

        if(c.verifyValid()):

            self.childAppend(c, fin)

        c = board.copy()

        c.swap(c.eX, c.eY, c.eX, c.eY - 1) ## Swap Down

        if(c.verifyValid()):

            self.childAppend(c, fin)

    def childAppend(self, obj, fin):

        global statevector

        if(obj is not self.parent and obj not in statevector):

            statevector.append(obj) # Append into global state list ie CLOSED LIST

            t = TNode(obj)

            t.parent = self

            t.h = Board.hx(obj, fin)

            self.childs.append(t)

    def \_\_repr\_\_(self):

        s = "TNode with data: %s && Childs: %d"%(self.state, len(self.childs))

        return s

    def copy(self):

        t = TNode(self.state)

        return t

    def \_\_eq\_\_(self, other):

        return self.h == other.h

    def \_\_lt\_\_(self, other):

        return self.h < other.h

initialData = [[2, 8, 3], [1, 5, 4], [7, 6, -1]]

initBoard = Board(initialData)

finalData = [[1, 2, 3], [8, -1, 4], [7, 6, 5]]

finBoard = Board(finalData)

parent = TNode(initBoard)

q = [ parent ]

FOUND = False

fNode = None

while(q):

    e = q.pop(0)

    if(e.state == finBoard):

        FOUND = True

        fNode = e

        break

    e.walkAndH(finBoard)

    c = e.childs

    q.extend(c)

    q = sorted(q) # Sort and choose best elements

def rPrint(x):

    if(x is None): return

    rPrint(x.parent)

    print(x.state)

if(FOUND):

    print("found at")

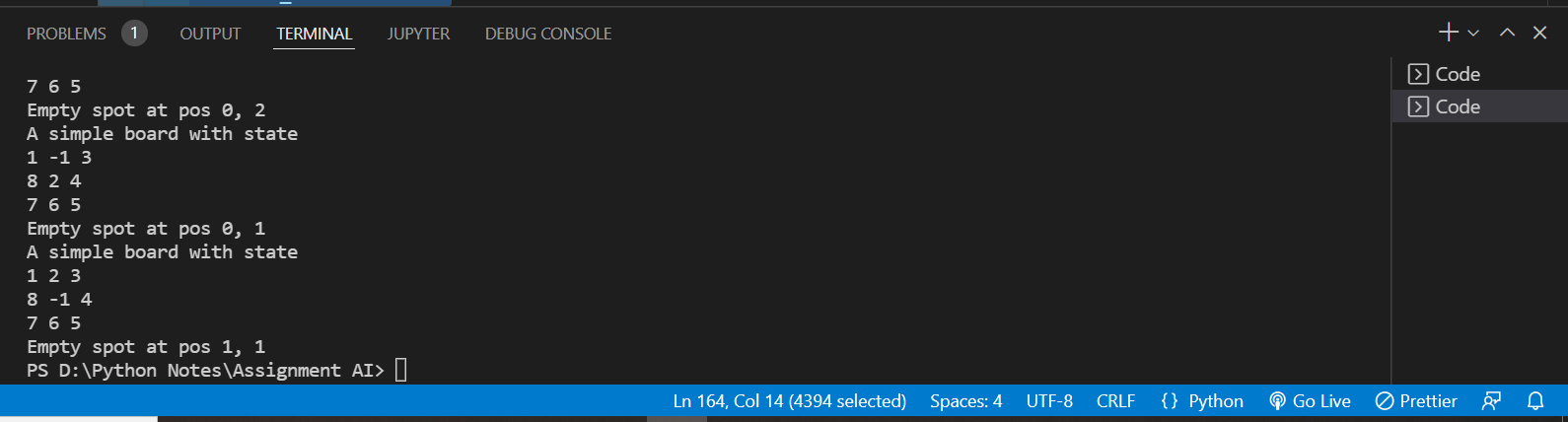
    print(fNode)

    print("Tracing path")

    x = fNode

    rPrint(x)

OUTPUT



Q5) Solve this given problem using Uniform Cost search. A is the initial state and G is the goal state

def uniform\_cost\_search(goal, start):

    global graph, cost

    answer = []

    queue = []

    for i in range(len(goal)):

        answer.append(10\*\*8)

    queue.append([0, start])

    visited = {}

    # count

    count = 0

    while (len(queue) > 0):

        queue = sorted(queue)

        p = queue[-1]

        del queue[-1]

        # get the original value

        p[0] \*= -1

        if (p[1] in goal):

            # get the position

            index = goal.index(p[1])

            if (answer[index] == 10\*\*8):

                count += 1

            if (answer[index] > p[0]):

                answer[index] = p[0]

            # pop the element

            del queue[-1]

            queue = sorted(queue)

            if (count == len(goal)):

                return answer

        if (p[1] not in visited):

            for i in range(len(graph[p[1]])):

                queue.append(

                    [(p[0] + cost[(p[1], graph[p[1]][i])]) \* -1, graph[p[1]][i]])

        # mark as visited

        visited[p[1]] = 1

    return answer

# main function

if \_\_name\_\_ == '\_\_main\_\_':

    # create the graph

    graph, cost = [[] for i in range(8)], {}

    # add edge

    graph[0].append(1)

    graph[0].append(3)

    graph[0].append(2)

    graph[3].append(4)

    graph[2].append(4)

    graph[1].append(4)

    # add the cost

    cost[(0, 1)] = 1

    cost[(0, 3)] = 15

    cost[(0, 2)] = 5

    cost[(3, 4)] = 5

    cost[(2, 4)] = 5

    cost[(1, 4)] = 10

    # goal state

    goal = []

    goal.append(4)

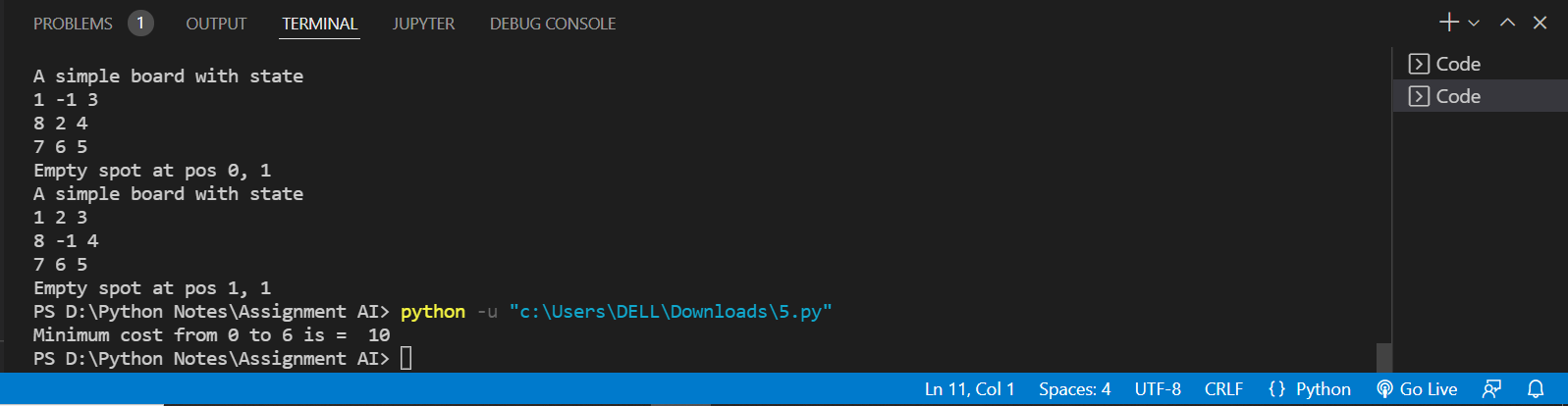
    # get the answer

    answer = uniform\_cost\_search(goal, 0)

    # print the answer

    print("Minimum cost from 0 to 6 is = ", answer[0])

OUTPUT



Assignment-4

Name: Karan Nasa

Roll: 102003008

Group: 2COE1

Q1) Solve the following blocks world problem using Simple Hill Climbing Algorithm. Let the heuristic be +1 if the block is resting on the correct block and -1 if it is resting on the incorrect block

from copy import deepcopy as dc

class pair:

    def \_\_init\_\_(self,below,block):

        self.below = below

        self.block = block

def heuristic(state,goal):

    val = 0                 # heuristic value to return at end

    for ps in state:        # loop through all pairs in state

        for pg in goal:     # loop through all pair in goal state

            if(ps.block == pg.block):

                if(ps.below == pg.below):   # if block on correct block +1 val

                    val +=1

                else:

                    val -= 1                # else -1 val

                break

    return val

def get\_top\_blocks(state):          # return name of block which can be moved

    top = ['A','B','C','D']

    for pair in state:

        if pair.below in top:

            top.remove(pair.below)

    return top

def update(state,entry):    # add new pos of block specified by entry and remove previous position

    for p in state:

        if(p.block == entry.block):

            state.remove(p)

            state.append(entry)

            return

def move(init,goal):        # produce steps and return one with more heuristic value the init state

    state = dc(init)

    heu = heuristic(init,goal)

    top = get\_top\_blocks(state)

    # n^2 moves n = len(top)

    for i in range(len(top)):

        for j in range(len(top)):

            if( i == j ):   # place ith block on ground

                update(state,pair(None,top[i]))

                if(heuristic(state,goal)>heu):

                    return state

                state = dc(init)

            else:           # place the ith block on top of jth block

                update(state,pair(top[j],top[i]))

                if(heuristic(state,goal)>heu):

                    return state

                state = dc(init)

    return None     # if no better state found return none to show failure of simple hill climb

def print\_state(state):     # print state array as list of tuple rather than user defined class pair

    for p in state:

        print((p.below,p.block),end=', ')

    print()

def solve(init,goal):       # try solving blocks world problem using simple hill climb

    steps = 0

    state = dc(init)

    while(state!=None):

        print\_state(state)

        if(heuristic(state,goal)==len(state)):

            print('Hill Climb Successfull')

            print('Steps taken:',steps)

            return

        else:

            state = move(state,goal)

        steps +=1

    print('Hill Climb Unsuccessfull!')

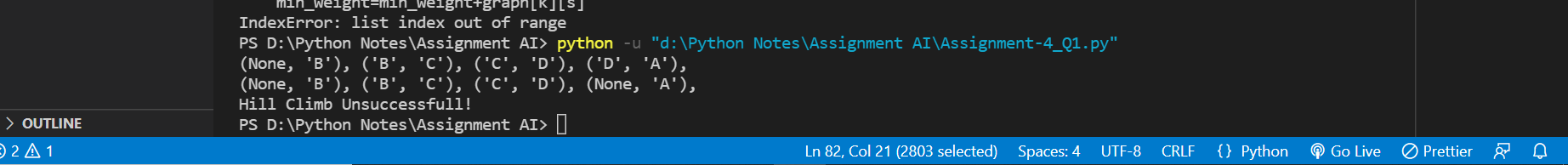
    return

if(\_\_name\_\_=='\_\_main\_\_'):

    init = [pair(None,'B'),pair('B','C'),pair('C','D'),pair('D','A')]

    goal = [pair(None,'A'),pair('A','B'),pair('B','C'),pair('C','D')]

    solve(init,goal)



Q2) Solve the following blocks world problem using Simple Hill Climbing Algorithm. Take the following heuristic function: +n for block which is resting on the current support structure and n is equal to number of blocks below it. -n for block which is resting on the incurrent support structure and n is equal to number of blocks below it.

from copy import deepcopy as dc

class pair:

    def \_\_init\_\_(self,below,block):

        self.below = below

        self.block = block

def state\_level\_map(state):     # return the map of level vs list of block at that level (used to cal heuristic value)

    level\_map = {-1: [None] ,0:[],1:[],2:[],3:[]}

    for i in range(4):

        if(len(level\_map[i-1])==0):

            break

        for p in state:

            if p.below in level\_map[i-1]:

                level\_map[i].append(p.block)

    return level\_map

def get\_below(state,block):     # in given state return the block below the specified block

    if(block == None):

        return None

    for p in state:

        if(p.block == block):

            return p.below

def heuristic(state,goal):      # calculate the heuristic value as +n : if block at correct pos with all below at correct and -n otherwise

                                # n = no of block below

    state\_level  = state\_level\_map(state)

    goal\_level = state\_level\_map(goal)

    val = 0

    for i in range(1,len(state)):   # iterate through 1 to n-1 (0 level doesn't affect val)

        if(len(state\_level[i])>0 and len(goal\_level[i])>0):

            for block in state\_level[i]:

                if block in goal\_level[i]:

                    bi = block

                    bg = bi

                    add = True

                    for k in range(i,0,-1):

                        bi = get\_below(state,bi)

                        bg = get\_below(goal,bg)

                        if(bi!=bg):

                           val -= i

                           add = False

                           break

                    if(add):

                        val += i

                else:

                    val -= i

        elif(len(state\_level[i])!= 0):

            val -= i

        else:

            return val

    return val

def get\_top\_blocks(state):      # get block on top which can only move

    top = ['A','B','C','D']

    for pair in state:

        if pair.below in top:

            top.remove(pair.below)

    return top

def update(state,entry):    # add entry state to state array and remove prev state

    for p in state:

        if(p.block == entry.block):

            state.remove(p)

            state.append(entry)

            return

def move(init,goal):    # generate new moves and return one with better heuristic value

    state = dc(init)

    heu = heuristic(init,goal)

    top = get\_top\_blocks(state)

    # n^2 move n = len(top)

    for i in range(len(top)):

        for j in range(len(top)):   # place the ith block on top of jth block

            if( i == j ):   # place ith block on ground

                update(state,pair(None,top[i]))

                if(heuristic(state,goal)>heu):

                    return state

                state = dc(init)

            else:

                update(state,pair(top[j],top[i]))

                if(heuristic(state,goal)>heu):

                    return state

                state = dc(init)

    return None     # return none to specify no better state found

def print\_state(state): # print state as tuple

    for p in state:

        print((p.below,p.block),end=', ')

    print()

def solve(init,goal):       # solve block world problem using the simple hill climb and given heuristic

    steps = 0

    state = dc(init)

    goal\_heu = heuristic(goal,goal)

    while(state!=None):

        print\_state(state)

        if(heuristic(state,goal)==goal\_heu):

            print('Hill Climb Successfull')

            print('Steps taken:',steps)

            return

        else:

            state = move(state,goal)

        steps +=1

    print('Hill Climb Unsuccessfull!')

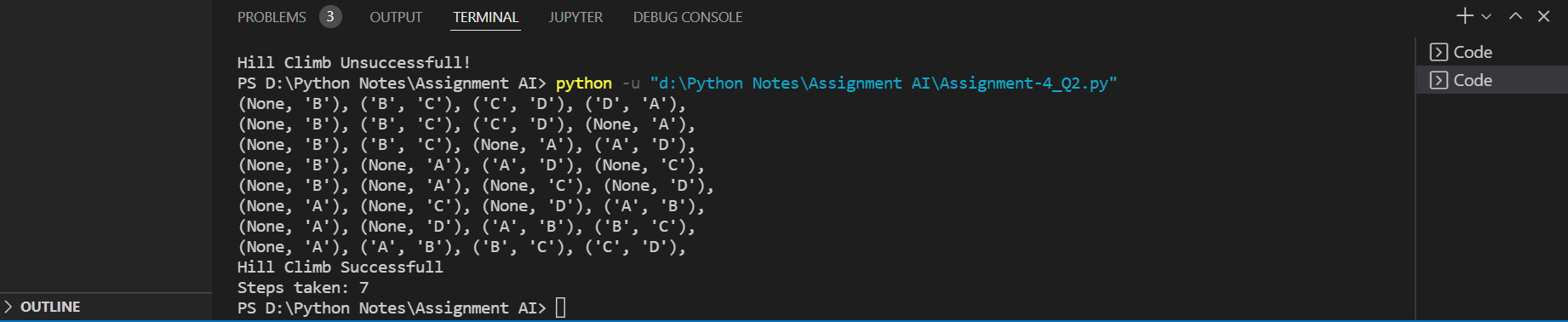
    return

if(\_\_name\_\_=='\_\_main\_\_'):

    init = [pair(None,'B'),pair('B','C'),pair('C','D'),pair('D','A')]

    goal = [pair(None,'A'),pair('A','B'),pair('B','C'),pair('C','D')]

    solve(init,goal)



Q3) Solve the following blocks world problem using Steepest Hill Climbing Algorithm. Take the following heuristic function: +n for block resting on the current support structure and n is equal to number of blocks below it. -n for block resting on the incurrent support structure and n is equal to number of blocks below it

from copy import deepcopy as dc

class pair:

    def \_\_init\_\_(self,below,block):

        self.below = below

        self.block = block

def state\_level\_map(state):         # return the map of level vs list of block at that level (used to cal heuristic value)

    level\_map = {-1: [None] ,0:[],1:[],2:[],3:[]}

    for i in range(len(state)):

        if(len(level\_map[i-1])==0):

            break

        for p in state:

            if p.below in level\_map[i-1]:

                level\_map[i].append(p.block)

    return level\_map

def get\_below(state,block):     # in given state return the block below the specified block

    if(block == None):

        return None

    for p in state:

        if(p.block == block):

            return p.below

def heuristic(state,goal):      # calculate the heuristic value as +n : if block at correct pos with all below at correct and -n otherwise

                                # n = no of block below

    state\_level  = state\_level\_map(state)

    goal\_level = state\_level\_map(goal)

    val = 0

    for i in range(1,len(state)):

        if(len(state\_level[i])>0 and len(goal\_level[i])>0):

            for block in state\_level[i]:

                if block in goal\_level[i]:

                    bi = block

                    bg = bi

                    add = True

                    for k in range(i,0,-1):

                        bi = get\_below(state,bi)

                        bg = get\_below(goal,bg)

                        if(bi!=bg):

                           val -= i

                           add = False

                           break

                    if(add):

                        val += i

                else:

                    val -= i

        elif(len(state\_level[i])!= 0):

            val -= i

        else:

            return val

    return val

def get\_top\_blocks(state):              # get block on top which can only move

    top = ['A','B','C']

    for pair in state:

        if pair.below in top:

            top.remove(pair.below)

    return top

def update(state,entry):         # add entry state to state array and remove prev state

    for p in state:

        if(p.block == entry.block):

            state.remove(p)

            state.append(entry)

            return

def move(init,goal):    # generate all possible moves and return one with best heuristic value

    state = dc(init)

    max\_heu = heuristic(init,goal)

    ret\_state = init

    top = get\_top\_blocks(state)

    # n^2 move n = len(top)

    for i in range(len(top)):

        for j in range(len(top)):   # place the ith block on top of jth block

            if( i == j ):   # place ith block on ground

                update(state,pair(None,top[i]))

                h\_val = heuristic(state,goal)

                if(h\_val>=max\_heu):

                    max\_heu = h\_val

                    ret\_state = dc(state)

                state = dc(init)

            else:

                update(state,pair(top[j],top[i]))

                h\_val = heuristic(state,goal)

                if(h\_val>=max\_heu):

                    max\_heu = h\_val

                    ret\_state = dc(state)

                state = dc(init)

    if (ret\_state == init):

        return None             # return None to show failue of Steepest Hill Climb

    return ret\_state

def print\_state(state):      # print state as tuple

    for p in state:

        print((p.below,p.block),end=', ')

    print()

def solve(init,goal):       # solve block world problem using the steepest hill climb and given heuristic

    steps = 0

    state = dc(init)

    goal\_heu = heuristic(goal,goal)

    while(state!=None):

        print\_state(state)

        if(heuristic(state,goal)==goal\_heu):

            print('Steepest Hill Climb Successfull')

            print('Steps taken:',steps)

            return

        else:

            state = move(state,goal)

        steps +=1

    print('Steepest Hill Climb Unsuccessfull!')

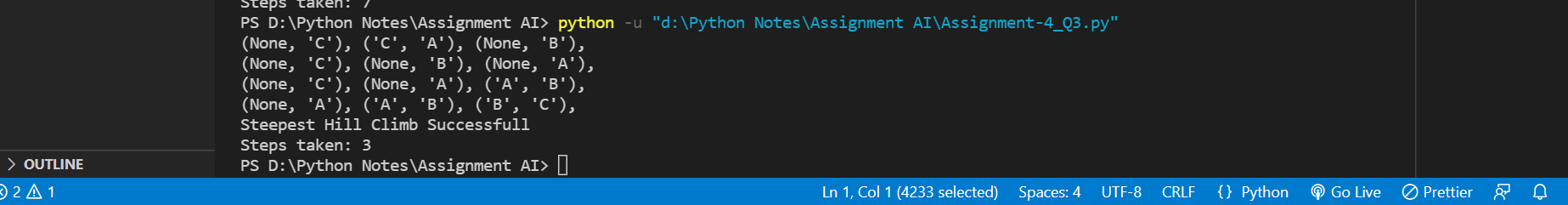
    return

if(\_\_name\_\_=='\_\_main\_\_'):

    init = [pair(None,'C'),pair('C','A'),pair(None,'B')]

    goal = [pair(None,'A'),pair('A','B'),pair('B','C')]

    solve(init,goal)



Q4) Solve the following problem using beam search algorithm, by taking (i) Beam width =2 (ii) Beam width =3 Heuristic values are given in the diagram; A is the starting node and G is the goal node.

def explore(graph,val,pq,parent,beta): # add child node to the queue

    if(graph[val]==None):

        return

    for heu,name in graph[val]:

        node = (heu,val,name)

        pq.append(node)

        parent.append((val,name))      # set parent of each child node

    pq.sort()                           # sort priority queue in incr order of heuristic value

    while(len(pq)>beta):

        pq.pop(beta)        # restrict the number of elements in queue by beta

def get\_steps(parent,goal):    # traverse from goal to init node to get path

    node = goal

    steps = []

    while(node!=None):

        steps.insert(0,node)    # since going backward each node inserted at 0 idx

        for p in parent:

            if(p[1]==node):

                node = p[0]     # set node to the parent of node

                break

    return steps

def beam\_search(graph,init,goal,beta):

    pq = [(None,None,init)]     # priority queue of max length beta

    parent=[(None,init)]        # maintain parent of each node help to get path if found

    while(pq):                  # loop till priority queue is not empty

        val = pq.pop(0)[2]      # get node with min heuristic value

        if(val == goal):        # if node is goal node break and return steps

            steps = get\_steps(parent,goal)

            return steps

        else:

            explore(graph,val,pq,parent,beta)   # explore child node and add them to pq

    return None

if(\_\_name\_\_=='\_\_main\_\_'):

    graph = {

        'A' : [(1,'B'),(3,'C')],

        'B' : [(2,'D'),(2,'E')],

        'C' : [(3,'F'),(0,'G')],

        'D' : None,

        'E' : None,

        'F' : None,

        'G' : None,

    }

    init = 'A'

    goal = 'G'

    beta = 2

    steps = beam\_search(graph,init,goal,beta)

    if(steps == None):

        print('\nBeam Search unsuccessful for beta:',beta)

    else:

        print('Steps:')

        print(steps)

    beta = 3

    steps = beam\_search(graph,init,goal,beta)

    if(steps == None):

        print('\nBeam Search unsuccessful for beta:',beta)

    else:

        print('\nSteps for beta',beta,':')

        print(steps)

