import string

import random

stacks=int(input("enter number of stacks :"))

blocks=int(input("enter number of blocks :"))

def definition\_of\_problem(stacks,blocks):

l=stacks

b=list(string.ascii\_uppercase)

list\_blocks=b[:blocks]

random.shuffle(list\_blocks)

problem\_state=[]

while blocks :

if not list\_blocks: break

if stacks==1:

problem\_state.append(list\_blocks)

break

else:

r=random.randint(1,blocks)

s=list\_blocks[:r]

problem\_state.append(s)

blocks-=r

stacks-=1

list\_blocks=list\_blocks[r:]

while len(problem\_state)<l:

problem\_state+=[[]]

random.shuffle(problem\_state)

return problem\_state

problem\_state=definition\_of\_problem(stacks,blocks)

print('Generated Problem is:',problem\_state)

def solution\_of\_problem(problem\_state):

final =[]

for stack in problem\_state:

final +=stack

final.sort()

final =[final]

for i in range(len(problem\_state)-1):

final+=[[]]

return final

final =solution\_of\_problem(problem\_state)

print("the goal state will be :")

print(final)

import copy

import numpy as np

class Node :

def \_\_init\_\_(self,elements,parent=None) :

self.node=elements

self.parent=parent

self.depth=0

if parent:

self.depth=parent.depth+1

def goal\_test(self) :

if self.node==final :

print("Solution Found")

self.traceback()

return True

else:

return False

def successor(self) :

children=[]

for i,stack in enumerate(self.node) :

for j,stack1 in enumerate(self.node):

if i !=j and len(stack1):

temp =copy.deepcopy(stack)

child=copy.deepcopy(self)

temp1=copy.deepcopy(stack1)

temp.append(temp1[-1])

del temp1[-1]

child.node[i]=temp

child.node[j]=temp1

child.parent=copy.deepcopy(self)

children.append(child)

return children

def heuristics(self) :

not\_on\_stack\_zero=len(final[0])-len(self.node[0])

wrong\_on\_stack\_zero=0

for i in range(len(self.node[0])) :

if self.node[0][i]!=final[0][i] :

wrong\_on\_stack\_zero +=2

dis\_bw\_pairs=0

for stack\_iter in range(1,len(self.node)):

for val in range(len(self.node[stack\_iter])-1):

if self.node[stack\_iter][val]>self.node[stack\_iter][val+1]:

dis\_bw\_pairs+=1

return not\_on\_stack\_zero +4\*wrong\_on\_stack\_zero-dis\_bw\_pairs

def path\_cost(self):

return self.heuristics()+self.depth

def traceback(self):

s,path\_back=self,[]

while s:

path\_back.append(s.node)

s=s.parent

print('Number of moves req :',len(path\_back))

print("List of nodes forming the path from the root tothe goal")

for i in list(reversed(path\_back)) :

print(i)

problem\_state=Node(problem\_state)

current=copy.deepcopy(problem\_state)

try:

import Queue as Q

except ImportError:

import queue as Q

q=Q.PriorityQueue()

q.put((current.path\_cost(),current))

explored=[]

iter =0

max\_allowed\_qsize=3000

max\_qsize=0

while q.qsize():

max\_qsize=max(max\_qsize,q.qsize())

if q.qsize()>max\_allowed\_qsize :

print('failure due to overload')

break

current =q.get()[1]

if current.goal\_test():

break

iter+=1

explored.append(current.node)

for child in current.successor():

if q.qsize()>0:

a=[]

for el in np.array(q.queue)[:,1]:

a.append(el.node)

else :a=[]

if child.node not in explored and child.node not in a:

q.put((child.path\_cost(),child))

elif child.node in a :

for j in range(len(a)):

if a[j]==child.node:

c=q.queue[j]

if c[0]>child.path\_cost():

c[1].parent=current

c[1].cost=c[0]=child.path\_cost()

if not q.qsize():

print('solution is not possible,goal state is not achievable from given problem state.')

print('maximum allowed size for queue :',max\_allowed\_qsize)

print('maximum size of queue (during iteration):',max\_qsize)

print('number of iterations :',iter)