Assignment ReportTower of Hanoi (BFS implementation)

short line

Name : Capt MD Rafsun Sheikh

Student ID : 201614123

Section : B

Course Code : CSE 404

5th April, 2021

# Introduction

1. Title : Tower of Hanoi using BFS
2. Language used : python
3. IDE used : Pycharm

We need to implement Code for Tower of Hanoi using Breadth First Search Algorithm where we can select:

1. Peg number
2. Total number of disks
3. Initial disk position in pegs
4. And Final disk positions in disk

And we will get the following output:

1. Initial state
2. Final state
3. Step number and steps with Parent node and Child node
4. Tracking back path including the node numbers and states of that node

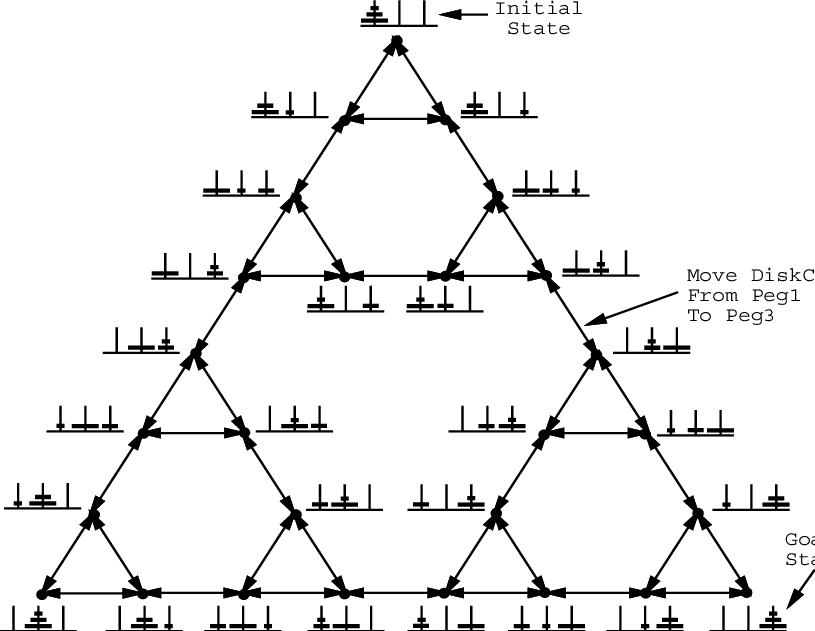
# 

# 

# Methodology

To solve the problem I used the following approach :

1. Asked for the user input to select :
   1. Peg Number
   2. For number of pegs in a for loop asked for Disks in all the pegs for the initial state
   3. For number of pegs in a for loop asked for Disks in all the pegs for the final state



1. Created a Node class and some initial node variables. Setting some default values and values from user input passed a Node class object to BFS function.
2. By taking the object of the node BFS function checks for whether it reaches the final state or not. If not reached it keeps on traversing the states for the final state and tree like structure is created.
3. All the nodes are capable of pointing to its parent state and child state. This is required so that at the end we can call the parent states from the final state.
4. All the nodes are assigned with a node number so that we can see the node number at any time.

# Code

from copy import deepcopy

class Node:

def \_\_init\_\_(self):

self.state = [[], [], []]

self.nodeNumber = 0

self.status = 'idle'

self.neighbours = []

self.parent = None

self.children = [] # For BFS

self.point = 10

def move(st1, st2):

s1 = st1[:]

s2 = st2[:]

if len(s1) > 0:

topDisc = s1[len(s1) - 1]

lastofS2 = len(s2) - 1

if len(s2) == 0 or s2[lastofS2] > topDisc:

s2.append(topDisc)

s1.pop()

return s1, s2

else:

return None

else:

return None

def moveDisc(n):

global noOfPegs

stacks = []

for x in range(0, noOfPegs):

for y in range(0, noOfPegs):

stacks = move(n.state[x], n.state[y])

if stacks != None:

# print 'states after move', states

nextnode = Node()

nextnode = deepcopy(n)

nextnode.state[x] = deepcopy(stacks[0])

nextnode.state[y] = deepcopy(stacks[1])

# print 'states', states

# print '\n'

# print 'next node',nextnode.state

if nextnode.state in states:

# print 'nextnode in states'

a = 1 # dumb value

else:

nodenumber = nextnode.nodeNumber

# print nextnode.state, 'next not in states'

states.append(nextnode.state)

return nextnode

# print 'DEAD END'

return None

def printPath(node):

print('Tracing back the Path')

while True:

print('Node number: ', node.nodeNumber, ' State: ', node.state)

if node.parent != None:

node = node.parent

else:

break

def BFS(node):

global parentList, nodenumber, childList, targetFound, step

print('\n STEP : ', step)

step += 1

for node in parentList:

if targetFound == False:

print('Parent Node:', node.nodeNumber, ' State :', node.state)

exhausted = False

parent = deepcopy(node)

i = 1

while exhausted == False:

i += 1

childnode = moveDisc(node)

if childnode != None:

nodenumber += 1

childnode.nodeNumber = nodenumber

childnode.parent = node

parent.children.append(childnode)

childList.append(childnode)

print(' Child Node:', childnode.nodeNumber, 'State:', childnode.state)

# print 'states', states

if childnode.state == finalState:

print('Final target reached')

printPath(childnode)

targetFound = True

else:

exhausted = True

parentList = deepcopy(childList)

childList = []

if targetFound == False:

BFS(parentList)

def readState():

global noOfPegs

state = []

for x in range(0, noOfPegs):

print('Discs in Peg', x + 1, ' : ', )

a = [int(x) for x in input().split()]

state.append(a)

return state

noOfPegs = 3

shouldContinue = True

while shouldContinue:

print('\n\nTower of Hanoi')

print('1. Breadth First Search')

print('2. Exit')

algoNumber = input("Please select the option --> ")

if algoNumber == '2':

print('\nExiting')

quit()

print('\nInstructions for input:')

print('-->An example input for discs in a peg >>> 3 2 1')

print('-->This means your peg have 3 discs with disc of size 3 at bottom and disc of size 1 at top')

print('-->If the peg is empty, just click ENTER; Do not input anything in that case')

noOfPegs = int(input("\nEnter number of pegs--> "))

print('\nEnter details for initial State')

initialState = readState()

print('\nEnter details for final State')

finalState = readState()

print('\nInitial state : ', initialState)

print('Final states : ', finalState)

# initialState=[[1],[3],[2]]

# finalState=[[3,1],[2],[]]

# initialState=[[3],[1],[2]]

# finalState=[[3,2,1],[],[]]

states = []

states = [initialState]

nodenumber = 1

time = 1

targetFound = False

node = Node()

node.state = initialState

node.nodeNumber = nodenumber

parentList = [node]

childList = []

targetFound = False

largestInTarget = False

step: int = 1

parentList = [node]

childList = []

if algoNumber == '1':

print('\nYou selected Breadth First Search')

BFS(node)

elif algoNumber == '2':

print('\nExiting')

quit()

else:

print('Please select a valid option')

continue

# Explanation of the code:

1. At first a class Node is created which will be required to create objects of the class. What class Node takes is
2. Present state that algorithm is working with like [ [3,2,1], [], [] ]
3. Number of the present node like 1 or 2 or 3 or ..
4. Parent of the present node
5. Children of the present node
6. Status of the present node
7. After taking the user input a node object is passed to function bfs where :
8. Step number is printed first
9. Then checking for the target node existing node is copied as parent node and passed the node object to function moveDisc
10. Inside the moveDisc the node is divided into possible child nodes and the child nodes are returned to childnode in function bfs
11. Then inside bfs function its checked if childnode has something in it, previous existing node is set as parent node of this childnode.
12. This childnode is appended as the child node of the previous parent node. And the node number is saved also.
13. Inside the parent Lists and child List all these nodes are saved in sequence.
14. At the end bfs function is called again until the target node is not generated.

# 

# Conclusion

For solving the problem I have taken help from the internet including some github repositories of some contributors. This problem is a bit different from usual bfs implementation. But I learned a lot while solving the problem. As there are no external libraries called here so the code can be run easily through command line using python. This code can be implemented more efficiently if tried.