Day 2

Breadth first search:

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = [] # List for visited nodes.

queue = [] #Initialize a queue

def bfs(visited, graph, node): #function for BFS

visited.append(node)

queue.append(node)

while queue: # Creating loop to visit each node

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

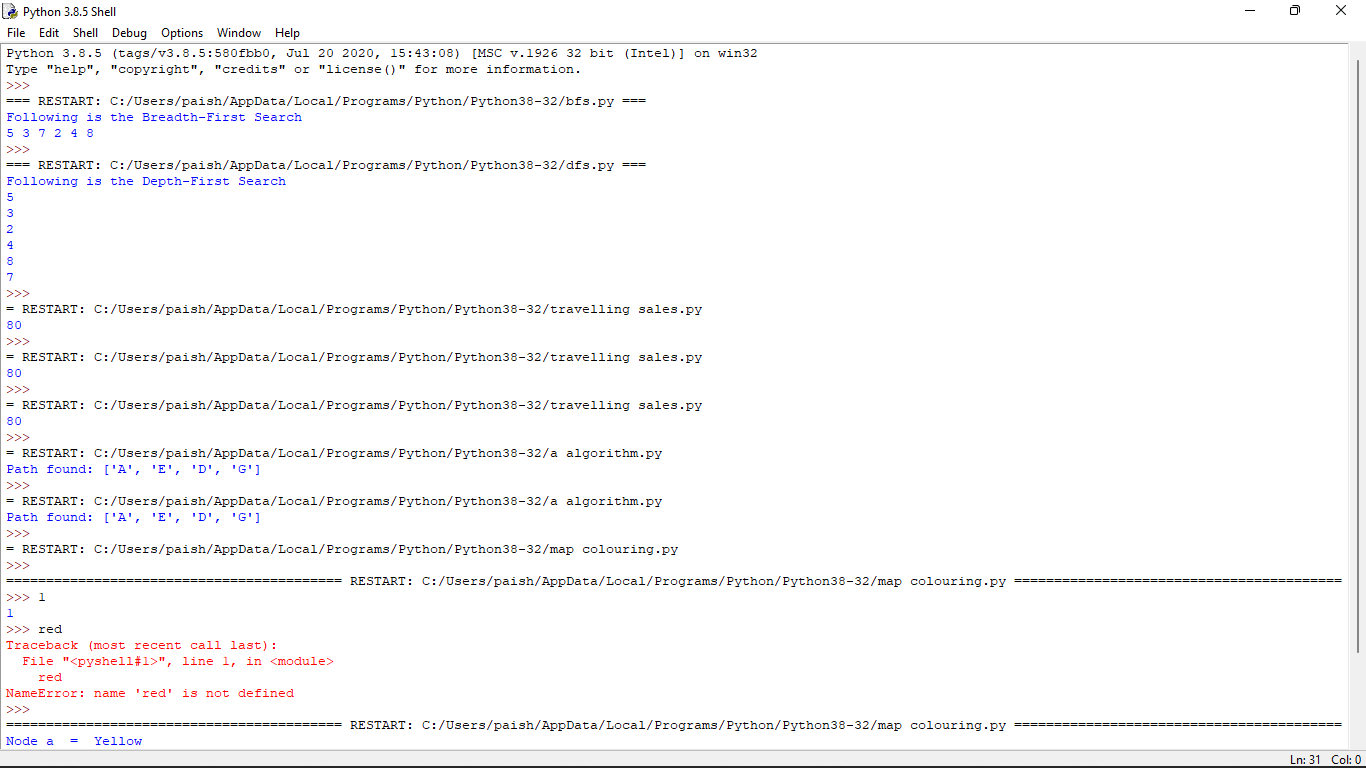
visited.append(neighbour)

queue.append(neighbour)

# Driver Code

print("Following is the Breadth-First Search")

bfs(visited, graph, '5') # function calling



Depth first search

# Using a Python dictionary to act as an adjacency list

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs

if node not in visited:

print (node)

visited.add(node)

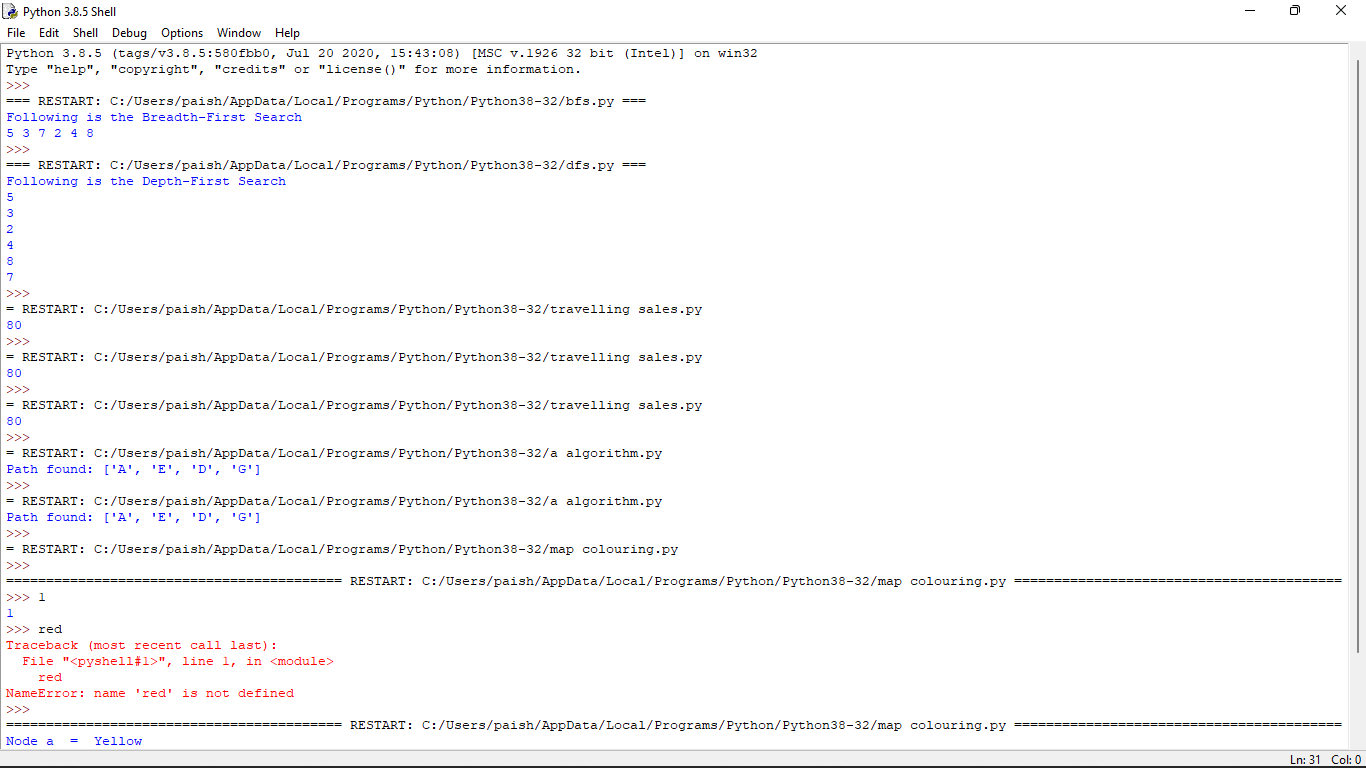
for neighbour in graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

print("Following is the Depth-First Search")

dfs(visited, graph, '5')



Travelling salesman problem:

from sys import maxsize

from itertools import permutations

V = 4

def travellingSalesmanProblem(graph, s):

vertex = []

for i in range(V):

if i != s:

vertex.append(i)

min\_path = maxsize

next\_permutation=permutations(vertex)

for i in next\_permutation:

current\_pathweight = 0

k = s

for j in i:

current\_pathweight += graph[k][j]

k = j

current\_pathweight += graph[k][s]

min\_path = min(min\_path, current\_pathweight)

return min\_path

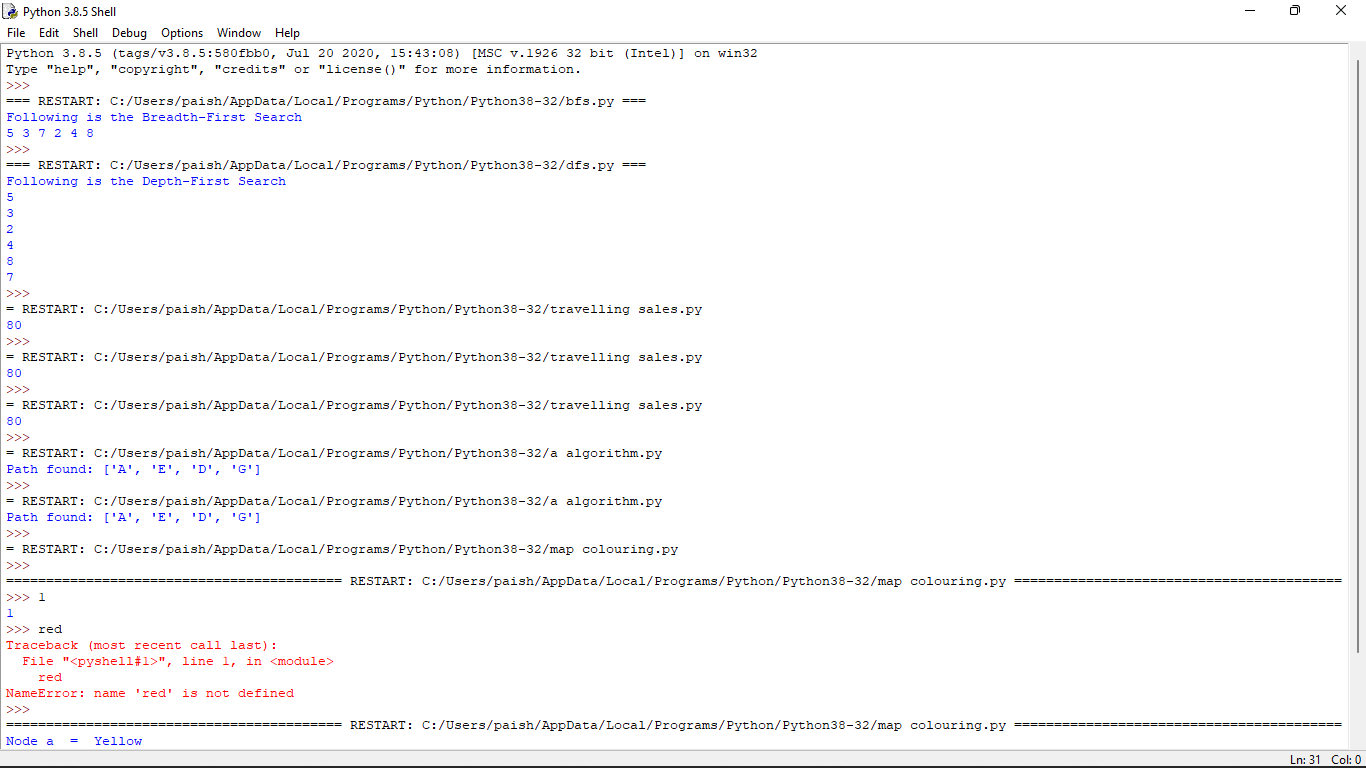
if \_\_name\_\_ == "\_\_main\_\_":

graph = [[0, 10, 15, 20], [10, 0, 35, 25],

[15, 35, 0, 30], [20, 25, 30, 0]]

s = 0

print(travellingSalesmanProblem(graph, s))



A\*algorithm program:

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {}

parents = {}

g[start\_node] = 0

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] == None:

pass

else:

for (m, weight) in get\_neighbors(n):

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 99,

'D': 1,

'E': 7,

'G': 0,

}

return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

'A': [('B', 2), ('E', 3)],

'B': [('C', 1),('G', 9)],

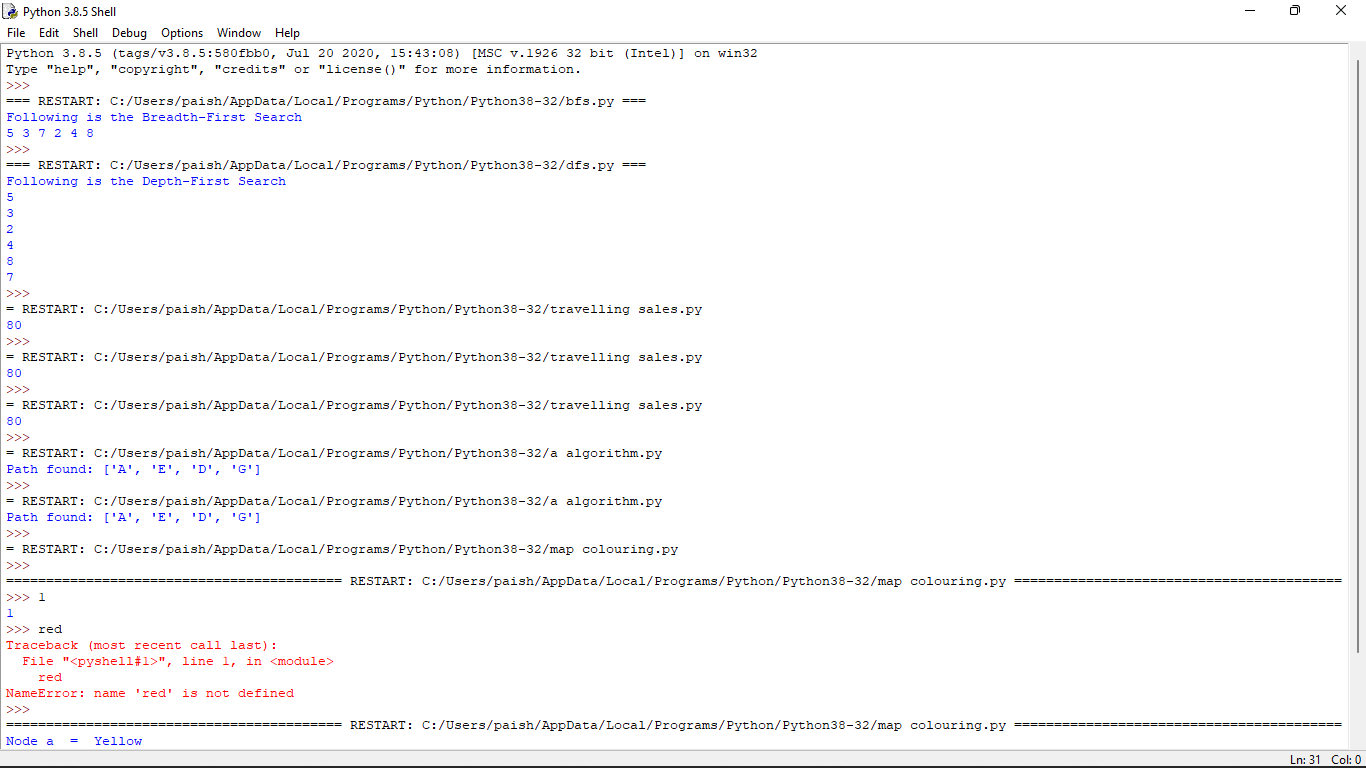
'C': None,

'E': [('D', 6)],

'D': [('G', 1)],

}

aStarAlgo('A', 'G')



Map colouring

G = [[ 0, 1, 1, 0, 1, 0],

[ 1, 0, 1, 1, 0, 1],

[ 1, 1, 0, 1, 1, 0],

[ 0, 1, 1, 0, 0, 1],

[ 1, 0, 1, 0, 0, 1],

[ 0, 1, 0, 1, 1, 0]]

node = "abcdef"

t\_={}

for i in range(len(G)):

t\_[node[i]] = i

degree =[]

for i in range(len(G)):

degree.append(sum(G[i]))

colorDict = {}

for i in range(len(G)):

colorDict[node[i]]=["Blue","Red","Yellow","Green"]

sortedNode=[]

indeks = []

for i in range(len(degree)):

\_max = 0

j = 0

for j in range(len(degree)):

if j not in indeks:

if degree[j] > \_max:

\_max = degree[j]

idx = j

indeks.append(idx)

sortedNode.append(node[idx])

theSolution={}

for n in sortedNode:

setTheColor = colorDict[n]

theSolution[n] = setTheColor[0]

adjacentNode = G[t\_[n]]

for j in range(len(adjacentNode)):

if adjacentNode[j]==1 and (setTheColor[0] in colorDict[node[j]]):

colorDict[node[j]].remove(setTheColor[0])

for t,w in sorted(theSolution.items()):

print("Node",t," = ",w)

