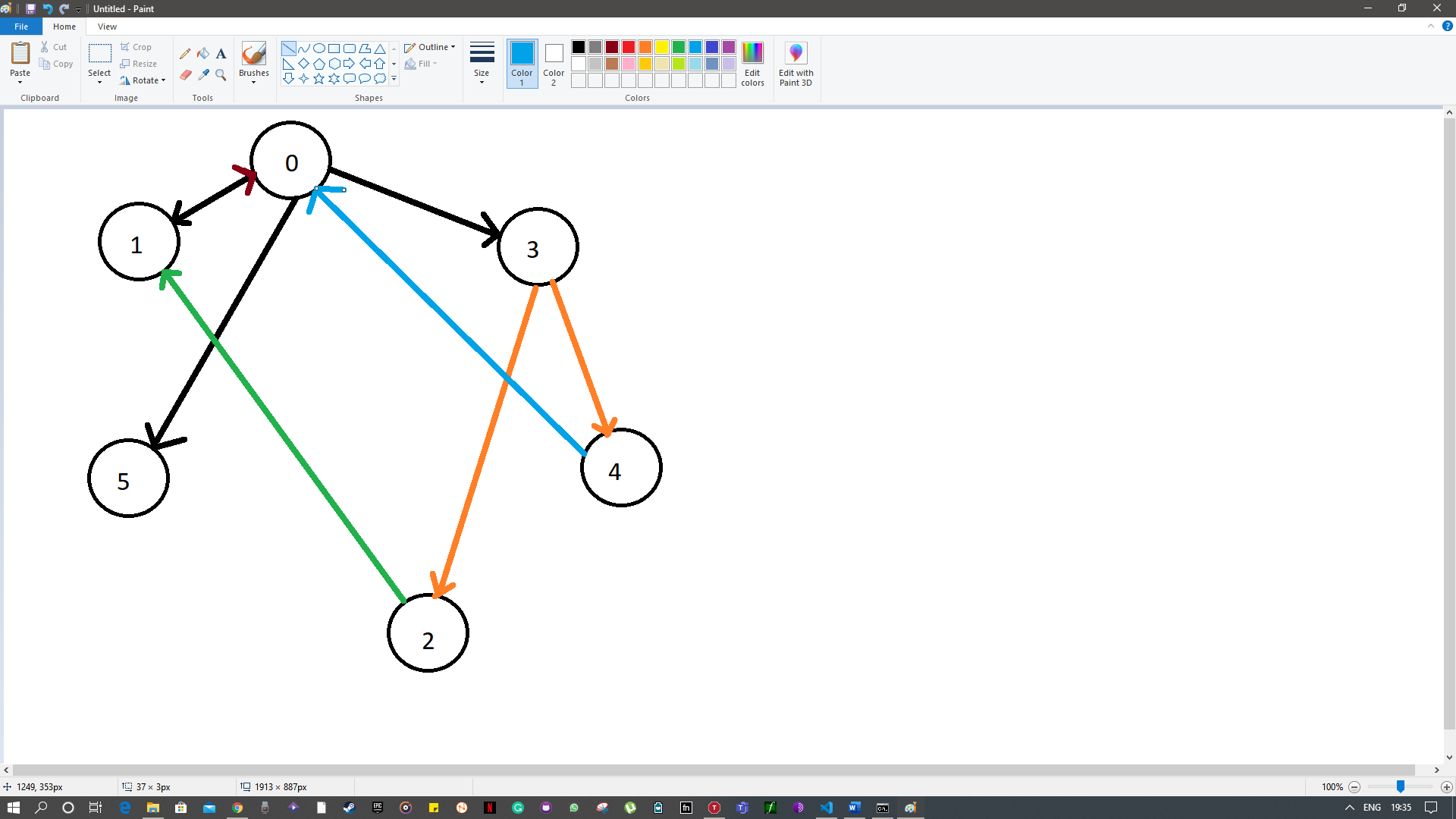
AI LAB Assignment

Abhirup Chakravarty – 17BCE7055

# Breadth First Search Vs. Depth First Search Comparison:

## Graph used for comparison:



## BFS:

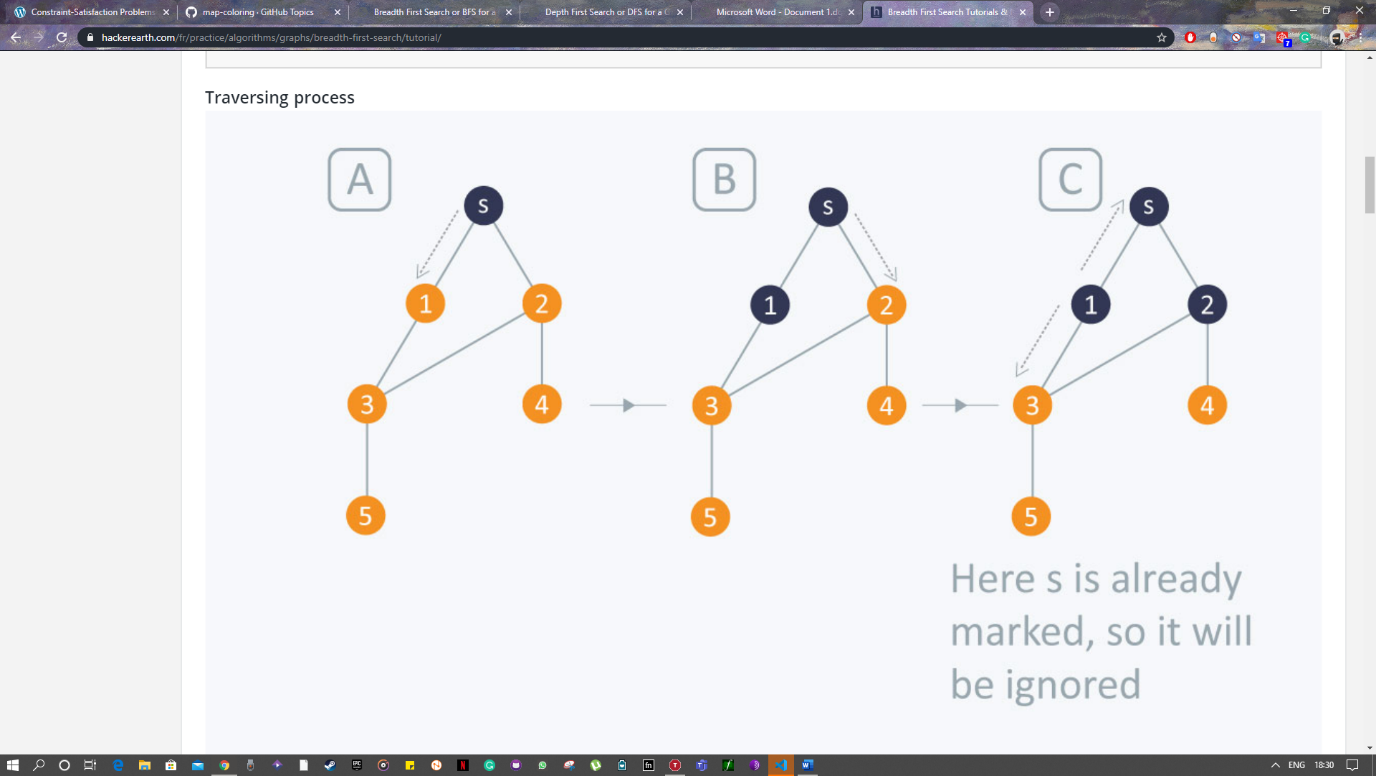
### Introduction:

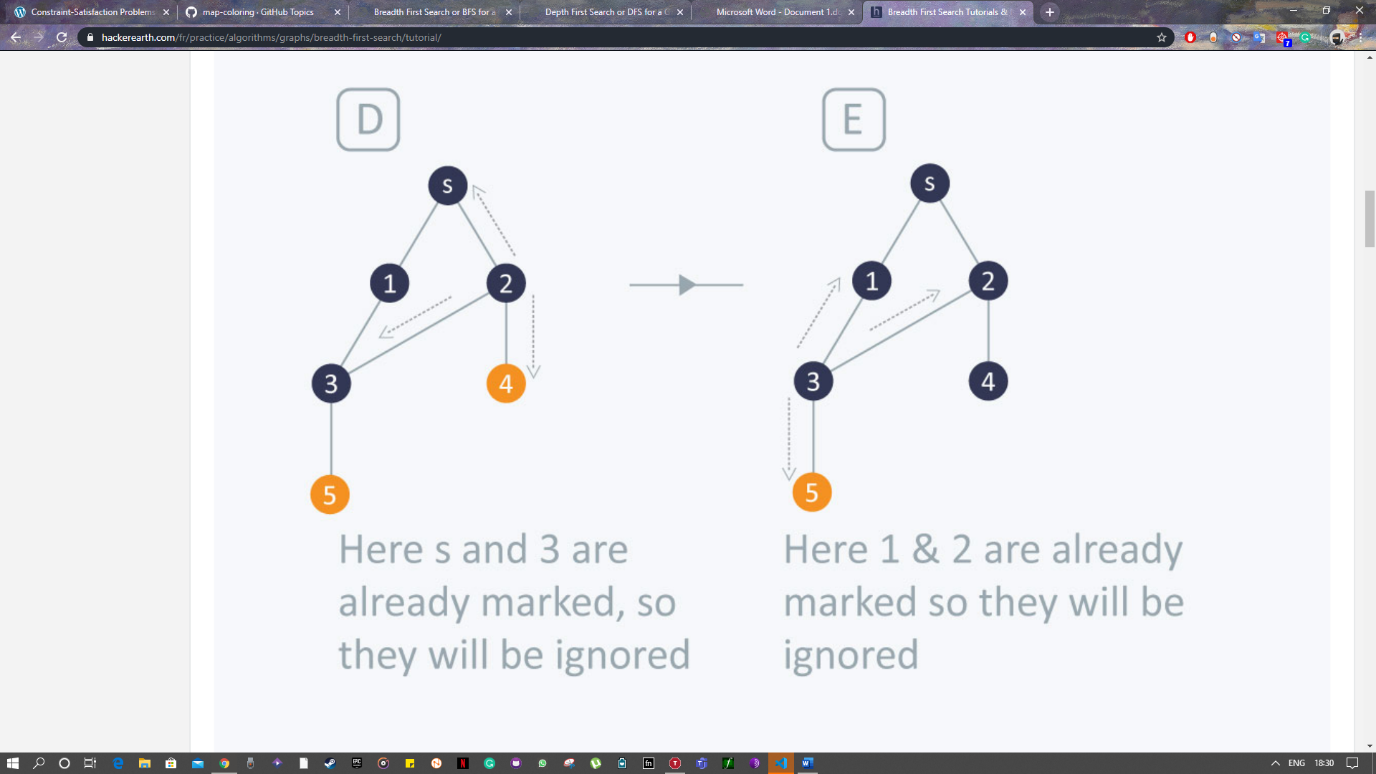
BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

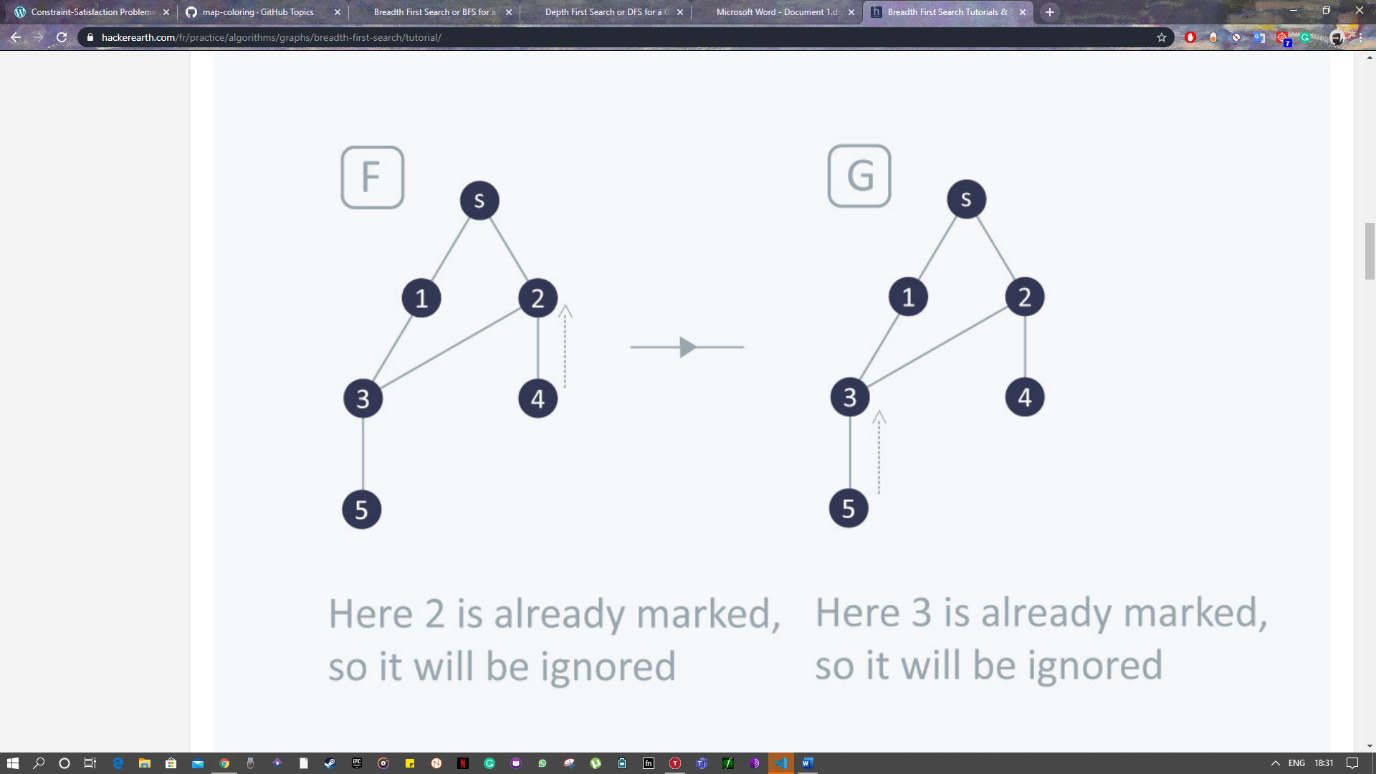
As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

An illustrative example is as follows:







The time complexity of BFS is **O(V + E)**, where V is the number of nodes and E is the number of edges.

### Pseudo Code:

BFS (G, s) #Where G is the graph and s is the source node

let Q be queue.

Q.enqueue( s ) #Inserting s in queue until all its neighbour vertices are marked.

mark s as visited.

while ( Q is not empty)

#Removing that vertex from queue,whose neighbour will be visited now

v = Q.dequeue( )

#processing all the neighbours of v

for all neighbours w of v in Graph G

if w is not visited

Q.enqueue( w ) #Stores w in Q to further visit its neighbour

mark w as visited.

### Code:

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def addEdge(self,u,v):

        self.graph[u].append(v)

    def \_BFS(self, s):

        \_alreadyVisited = [False] \* (len(self.graph))

        queue = []

        queue.append(s)

        \_alreadyVisited[s] = True

        while queue:

            s = queue.pop(0)

            print (s, end = " ")

            for i in self.graph[s]:

                if \_alreadyVisited[i] == False:

                    queue.append(i)

                    \_alreadyVisited[i] = True

g = Graph()

g.addEdge(0, 5)

g.addEdge(0, 3)

g.addEdge(0, 1)

g.addEdge(1, 0)

g.addEdge(2, 1)

g.addEdge(3, 2)

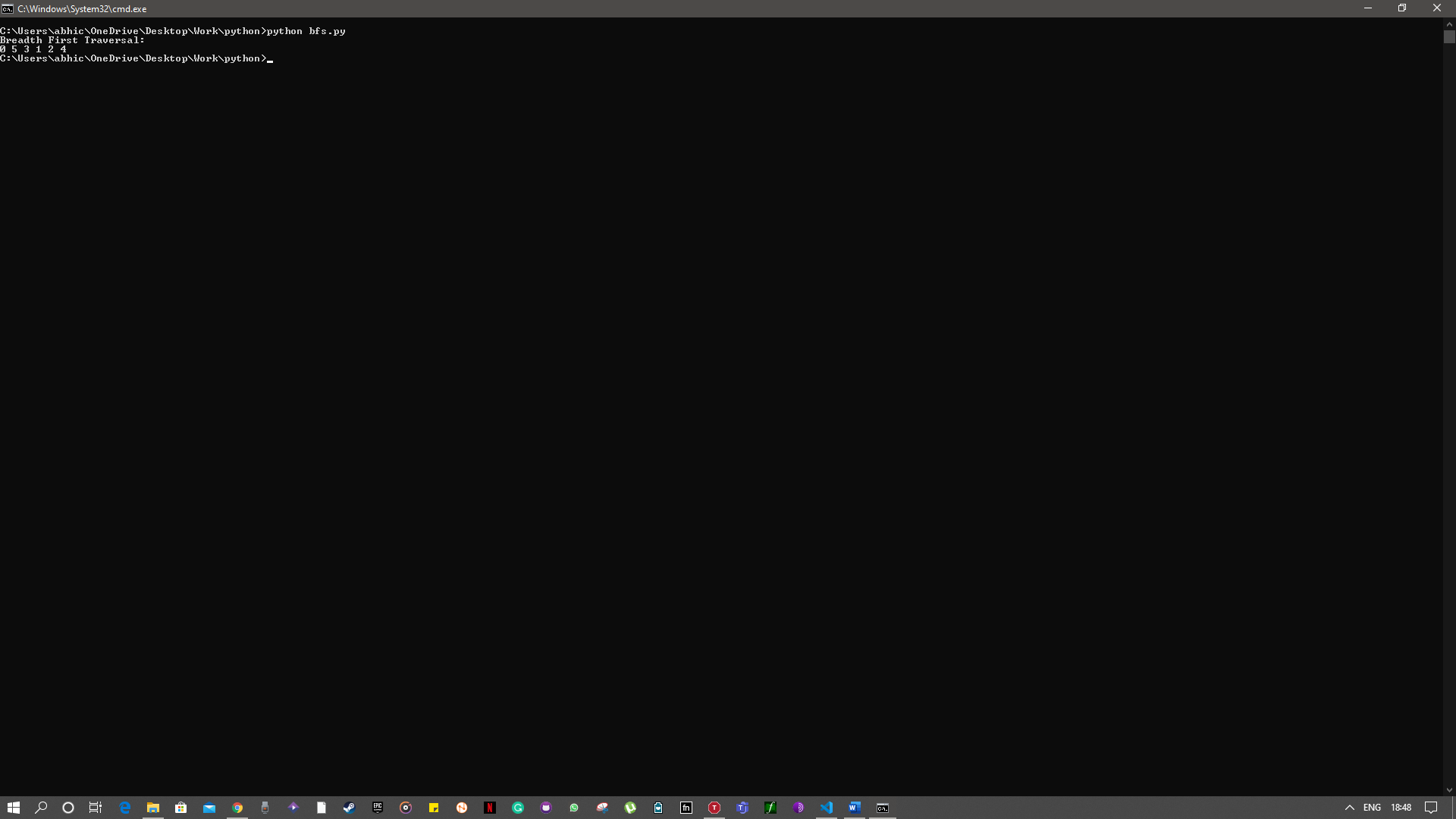
g.addEdge(3, 4)

g.addEdge(4, 0)

print ("Breadth First Traversal:")

g.\_BFS(0)

### Output:



## DFS:

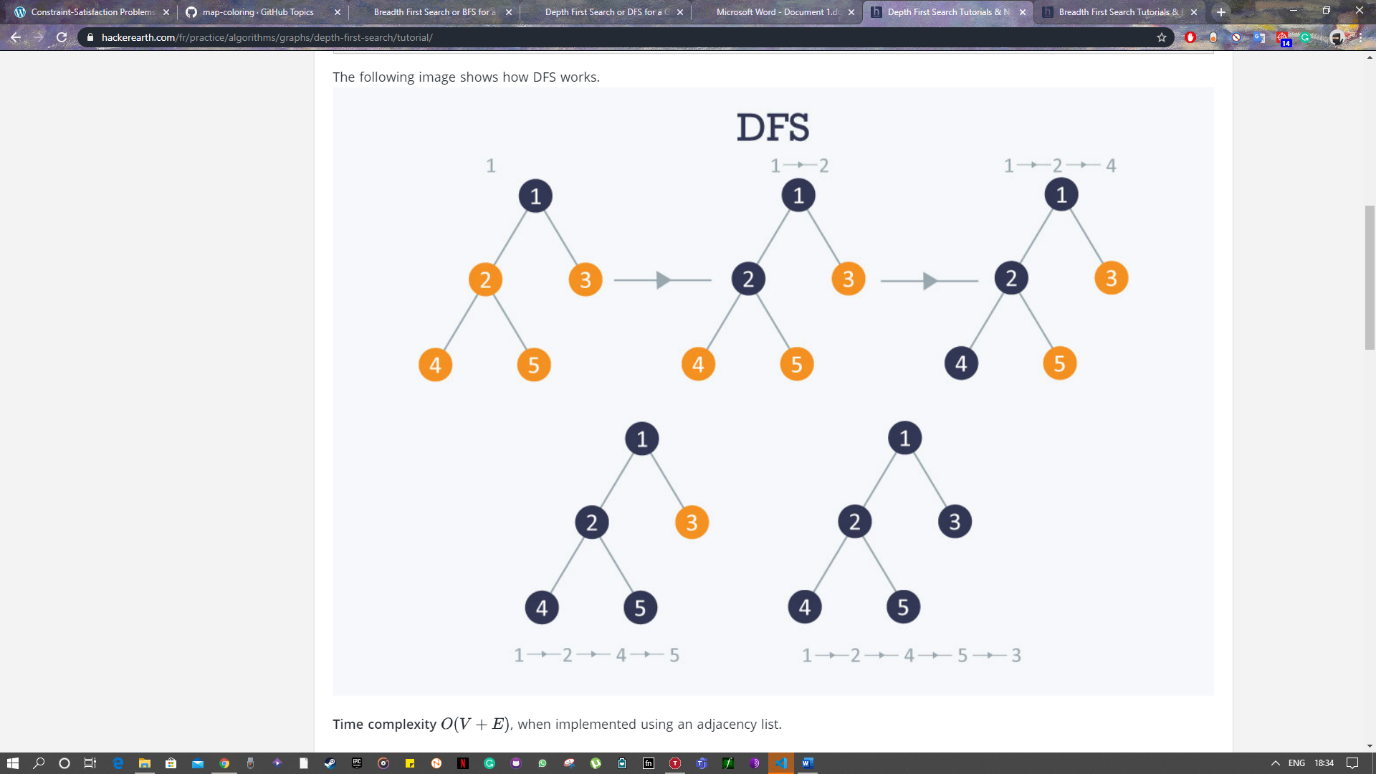
### Introduction:

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:  
Pick a starting node and push all its adjacent nodes into a stack.  
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.  
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

An illustrative example is as follows:



The time complexity of DFS is **O(V + E)**, where V is the number of nodes and E is the number of edges.

### Pseudo Code:

DFS-iterative (G, s): #Where G is graph and s is source vertex

let S be stack

S.push( s ) #Inserting s in stack

mark s as visited.

while ( S is not empty):

#Pop a vertex from stack to visit next

v = S.top( )

S.pop( )

#Push all the neighbours of v in stack that are not visited

for all neighbours w of v in Graph G:

if w is not visited :

S.push( w )

mark w as visited

DFS-recursive(G, s):

mark s as visited

for all neighbours w of s in Graph G:

if w is not visited:

DFS-recursive(G, w)

### Code:

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def addEdge(self,u,v):

        self.graph[u].append(v)

    def \_Utility(self, v, \_alreadyVisited):

        \_alreadyVisited[v]= True

        print (v)

        for i in self.graph[v]:

            if \_alreadyVisited[i] == False:

                self.\_Utility(i, \_alreadyVisited)

    def \_DFS(self):

        V = len(self.graph)

        \_alreadyVisited =[False]\*(V)

        for i in range(V):

            if \_alreadyVisited[i] == False:

                self.\_Utility(i, \_alreadyVisited)

g = Graph()

g.addEdge(0, 5)

g.addEdge(0, 3)

g.addEdge(0, 1)

g.addEdge(1, 0)

g.addEdge(2, 1)

g.addEdge(3, 2)

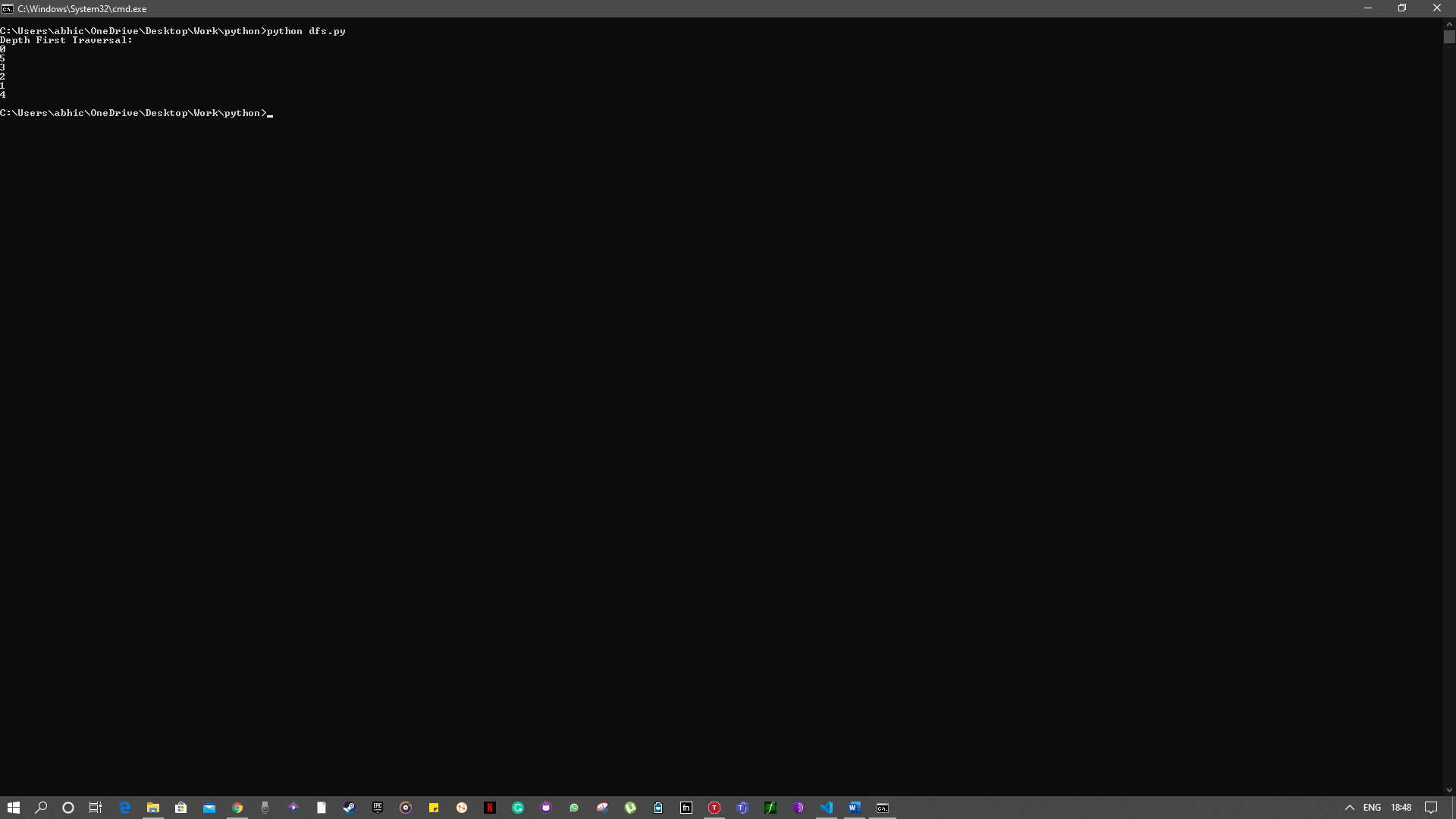
g.addEdge(3, 4)

g.addEdge(4, 0)

print ("Depth First Traversal:")

g.\_DFS()

### Output:



# Map Colouring as a Constraint Satisfaction Problem:

## 2.1. Constraint Satisfaction Problem:

## Constraint satisfaction problems (CSPs) are mathematical questions defined as a set of objects whose state must satisfy several constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research since the regularity in their formulation provides a common basis to analyse and solve problems of many seemingly unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. Constraint Programming (CP) is the field of research that specifically focuses on tackling this kind of problems.

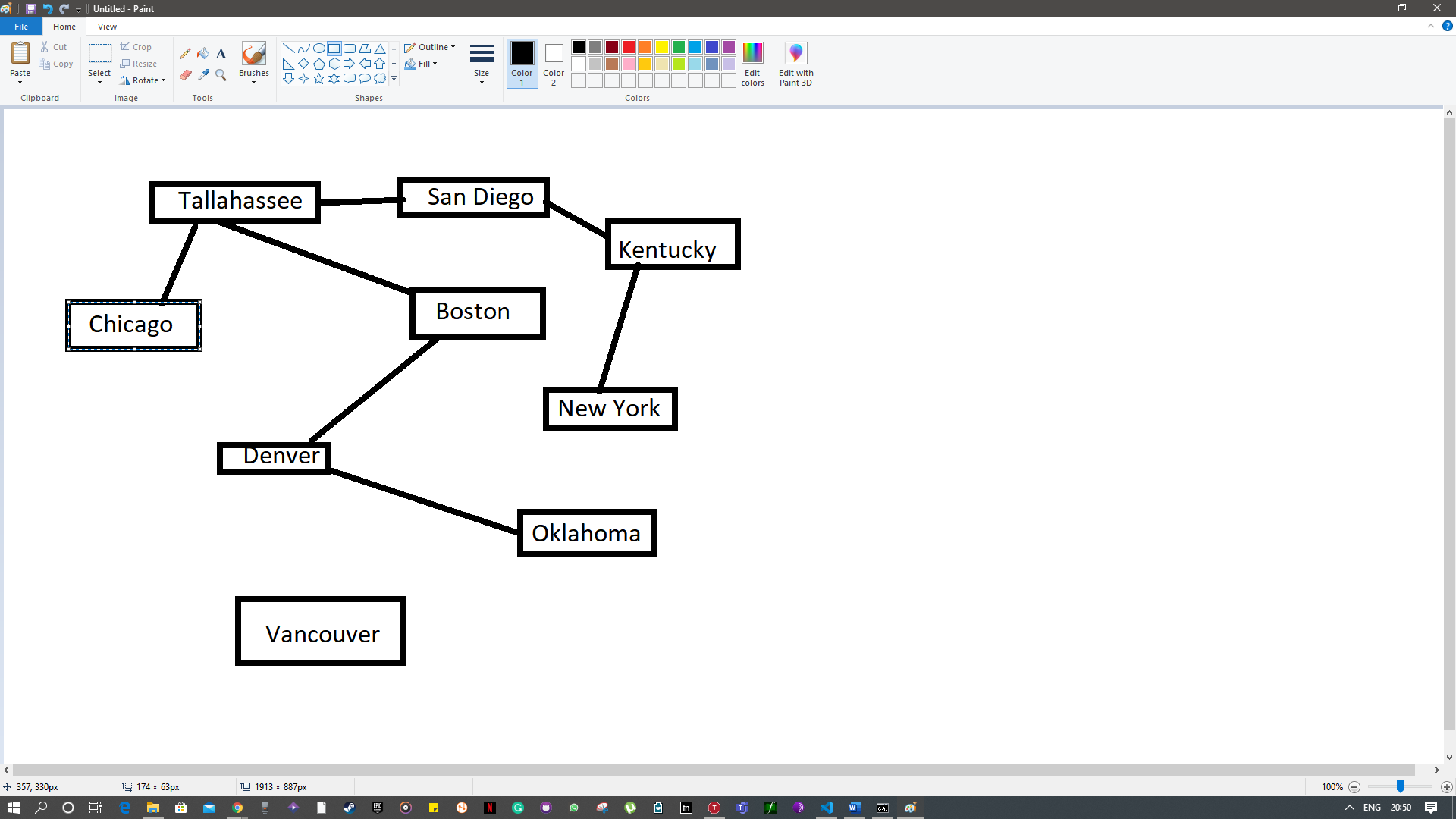
## 2.2. Map Colouring Problem:

In graph theory, map colouring, AKA graph colouring is a special case of graph labelling; it is an assignment of labels traditionally called "colours" to elements of a graph subject to certain constraints. In its simplest form, it is a way of colouring the vertices of a graph such that no two adjacent vertices are of the same colour; this is called a vertex colouring. Similarly, an edge colouring assigns a colour to each edge so that no two adjacent edges are of the same colour, and a face colouring of a planar graph assigns a colour to each face or region so that no two faces that share a boundary have the same colour.

Vertex colouring is usually used to introduce graph colouring problems since other colouring problems can be transformed into a vertex colouring instance. For example, an edge colouring of a graph is just a vertex colouring of its line graph, and a face colouring of a plane graph is just a vertex colouring of its dual. However, non-vertex colouring problems are often stated and studied as is. This is partly pedagogical, and partly because some problems are best studied in their non-vertex form, as in the case of edge colouring.

The convention of using colours originates from colouring the countries of a map, where each face is literally coloured. This was generalized to colouring the faces of a graph embedded in the plane. By planar duality it became colouring the vertices, and in this form, it generalizes to all graphs. In mathematical and computer representations, it is typical to use the first few positive or non-negative integers as the "colours". In general, one can use any finite set as the "colour set". The nature of the colouring problem depends on the number of colours but not on what they are.

## 2.3. Map Used:



## 2.4. Code:

import sys

from operator import itemgetter

import operator

def build\_g(map):

    graph = {}

    cs = map.split(";")

    cs = [c.strip('\n').replace(" ", "") for c in cs if len(

    c.strip('\n').replace(" ", "")) != 0]

    for c\_nbors in cs:

        c, nbors = c\_nbors.split(":")

        nbors = nbors.replace("[", "").replace("]", "").replace("\n", "").split(",")

        nbors = [(nbor, '') for nbor in nbors if nbor != '']

        graph[(c, False)] = nbors

    return graph

def diff(li1, li2):

    li\_dif = [i for i in li1 + li2 if i not in li1 or i not in li2]

    return li\_dif

def get\_allowed\_clrs(graph, c, clrs):

    not\_allowed\_clrs = [c[1] for c in graph[(c, False)] if c[1] is not '']

    allowed\_clrs = diff(clrs, not\_allowed\_clrs)

    return allowed\_clrs

def deg\_h(graph):

    id\_nbor\_len = [(index, len(l[1])) for index, l in enumerate(graph.items()) if l[0][1] is False]

    max\_nbor = max(id\_nbor\_len, key=itemgetter(1))[1]

    max\_id\_nbor\_len = [t[0] for t in id\_nbor\_len if t[1] == max\_nbor]

    cs = [list(graph)[index][0] for index in max\_id\_nbor\_len]

    return cs

def mrv(graph, clrs):

    cs\_wo\_clr = [(c[0]) for c, clred in graph.items() if c[1] is False]

    allowed\_clr\_eachc = {}

    for c in cs\_wo\_clr:

        allowed\_clr\_eachc[c] = get\_allowed\_clrs(graph, c, clrs)

    min\_ava\_clr\_len = min([len(allowed\_clrs) for c, allowed\_clrs in allowed\_clr\_eachc.items()])

    cs = [c for c, clrs in allowed\_clr\_eachc.items() if len(clrs) is min\_ava\_clr\_len]

    return cs

def lcv(graph, clrs):

    c\_clr = []

    clr\_no = {}

    for nbors in graph.values():

        [c\_clr.append(c) for c in nbors if c[1] is not '']

    all\_used\_clrs = list(dict.fromkeys(c\_clr))

    if not all\_used\_clrs:

        return clrs

    for cc in all\_used\_clrs:

        if cc[1] not in clr\_no:

            clr\_no[cc[1]] = 1

        else:

            clr\_no[cc[1]] += 1

    no\_max = max(clr\_no.items(), key=operator.itemgetter(1))[1]

    clrs = [key for (key, value) in clr\_no.items() if value is no\_max]

    return clrs

def clring(graph, c, clr):

    nbors = graph[(c, False)]

    del graph[(c, False)]

    graph[(c, True)] = nbors

    for nei\_list in graph.values():

        for n in nei\_list:

            if n[0] == c:

                l = list(n)

                l[1] = clr

                t = tuple(l)

                nei\_list.remove(n)

                nei\_list.append(t)

if \_\_name\_\_ == "\_\_main\_\_":

    map = """Tallahassee: [San Diego, Boston, Chicago];

            San Diego: [Tallahassee, Kentucky];

            Boston: [Denver, Tallahassee];

            Chicago: [Tallahassee];

            New York: [Kentucky];

            Kentucky: [San Diego, New York];

            Vancouver: [];

            Denver: [Boston, Oklahoma];

            Oklahoma: [Denver];"""

    clrs = ['VIOLET', 'INDIGO', 'BLUE', 'GREEN', 'YELLOW', 'ORANGE','RED']

    graph = build\_g(map)

    for \_ in range(len(graph)):

        max\_deg\_c = deg\_h(graph)

        cs\_min\_rem\_clrs = mrv(graph, clrs)

        much\_used\_clrs = lcv(graph, clrs)

        sel\_c = set(max\_deg\_c).intersection(set(cs\_min\_rem\_clrs)).pop()

        clrs\_of\_sel\_c = get\_allowed\_clrs(graph, sel\_c, clrs)

        common\_clr = set(much\_used\_clrs).intersection(set(clrs\_of\_sel\_c))

        try:

            if common\_clr:

                clr = common\_clr.pop()

            else:

                clr = clrs\_of\_sel\_c.pop()

            clring(graph, sel\_c, clr)

        except IndexError:

            sys.exit("Error. Try putting more colors.")

    alone\_cities = [graph[c].append("Is separated from the others. Can use any color.") for c, nbors in graph.items() if len(nbors) == 0]

    for city, neighbors in graph.items():

        print (city,"Neighboured by :", neighbors)

## 2.4. Output:

