CS301

2023-2024 Spring

Project Report

Group 62

::Group Members::

Berke Ayyıldızlı 31018

Beyza Balota 31232

1. Problem Description

1.1 Overview: Graph Colouring is one of the most anticipated problems in the computer science field. It consists of assigning colours to the vertices of a graph, but with a nuance that no two vertices share the same colour. The main aim of this algorithm is to find a way to paint the nodes while having the colour amount less, with respect to the rules.

1.2 Decision Problem: The decision problem of graph colouring is actually not a thought question, but it is a yes or no question which is "Can the vertices of a graph G be coloured with at most k different colours so that no two adjacent vertices have the same colour?". This problem is helpful in understanding the problem's computational complexity.

1.3 Optimization Problem: In this version of the problem, the main goal is to determine the smallest number k, which is the chromatic number of the graph, such that the graph G can be coloured with k colours, and no two adjacent vertices have the same colour. This problem's complexity is mostly higher than the decision problem since it aims to find the smallest number of k rather than verifying if a specific k is sufficient.

1.4 Example Illustration:

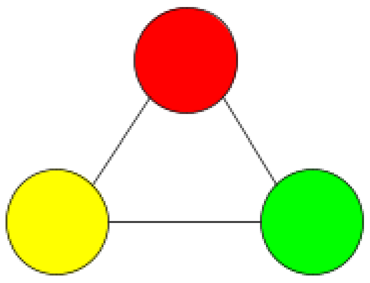


Figure 1: A graph with 3 nodes.

From here, we can see, on a small number of vertices, the chromatic number tends to be larger despite the fact that the vertex amount is lower. This phenomenon is caused by the increasing number of adjacent vertices.

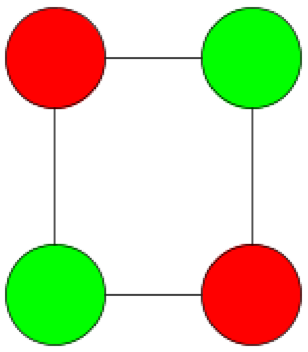


Figure 2: A graph with 4 nodes.

As it can be seen here, although the vertex amount is increased, the number of colours, aka the chromatic number, decreased, this graph is used to show that the number of vertices does not actually mean the number of colours when solving the graphs.

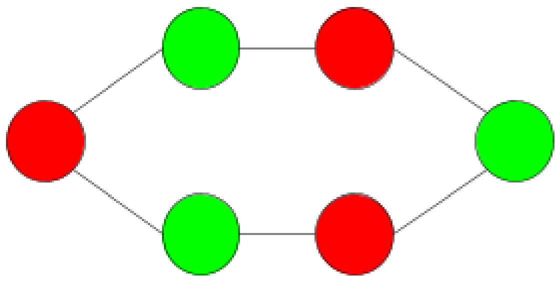


Figure 3: A graph with 6 nodes.

While also proving the phenomenon, the figure 3 helps us to further understand how the algorithm works when painting the graphs. It starts from the leftmost node, and colours it. After this operation, it checks the adjacent vertices, and uses recursive calls to learn if it needs a new colour or not. While using this approach, it successfully moves or backtracks on the graph, colouring it on the process.

1.5 Real World Applications: The graph colouring algorithms are used in many areas across the industry, some of them are as follows:

1.5.1 *Scheduling:* Graph colouring is used on scheduling tasks, where events or classes are represented as vertices, and the edges are drawn between two vertices if the events or classes have overlapping situations. The goal is to assign colours, that no event or class share the same colour (time slot).

1.5.2 *Map Colouring:* Graph colouring is mainly used in map colouring where adjacent cells must have different colours. The minimum number of colours needed to paint all the map is called the chromatic number.

1.5.3 *Wireless Communication:* Graph colouring is also used in wireless communication to ensure that the frequencies assigned to the devices are different from each other. Again, the chromatic number here gives the minimum number of frequencies to avoid confusion between the devices.

1.6 Hardness of the Problem:

The complexity of the graph colouring problem, particularly the 3COLOR decision problem, is evidenced by its classification as NP-hard. Cook-Levin theorem supports this classification, which demonstrates that the SAT problem (Boolean satisfiability problem) is NP-complete.[1] Being NP-complete means a problem is verifiable in polynomial time and is as computationally intensive as the most challenging problems in the NP class.

[1] Steven Rutherford, "On the Computational Hardness of Graph Coloring," *University of Washington, Department of Mathematics*, June 3, 2011.

To demonstrate the NP-hardness of 3COLOR, we use a reduction from SAT to 3SAT and then to 3COLOR. This method includes transforming a general SAT problem into an instance of 3SAT. This transformation indicates that solving the 3COLOR problem is at least as challenging as solving some of the most complex issues in computer science, highlighting the difficulties in finding efficient solutions to graph colouring in practical scenarios.

From this explanation, we can conclude that the graph colouring problem is classified as NP-hard because it can be reduced from the 3SAT problem, which is NP-complete. This shows that solving the 3COLOR problem is at least as difficult as the most challenging problems in NP, which makes it NP-Hard.

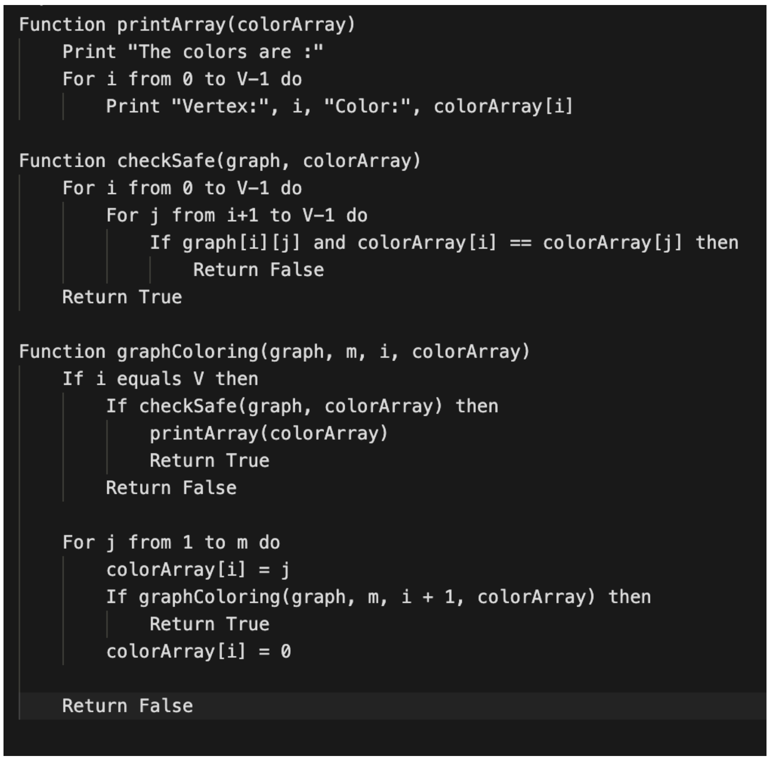
2. Algorithm Description (Brute Force)

2.1 Overview: The simplest approach to solve this problem would be to generate all possible combinations of colours. After the creation of one combination, the algorithm must check if the adjacent vertices have the same colour. If the wanted result is present, the loop is broken and the combination is added to the result. If we assume there are n number of colours, the total number of combinations are nv. From here we can see that the algorithm will try every colour for every node, the time complexity would be exponential.

Steps:

1. A recursive function that takes the current index, vertices, and the colour array.
2. Checks if the current colour is usable, meaning no adjacent vertices are painted with the same colour.
3. Assign another colour to the vertex. The range of the colour differs from 1 to m.
4. For every assigned colour, use the recursive function again.
5. If the call returns true, break the loop and return true.

2.2 Pseudocode:



```
Function printArray(colorArray)
    Print "The colors are :"
    For i from 0 to V-1 do
        Print "Vertex:", i, "Color:", colorArray[i]

Function checkSafe(graph, colorArray)
    For i from 0 to V-1 do
        For j from i+1 to V-1 do
            If graph[i][j] and colorArray[i] == colorArray[j] then
                Return False
    Return True

Function graphColoring(graph, m, i, colorArray)
    If i equals V then
        If checkSafe(graph, colorArray) then
            printArray(colorArray)
            Return True
        Return False

    For j from 1 to m do
        colorArray[i] = j
        If graphColoring(graph, m, i + 1, colorArray) then
            Return True
        colorArray[i] = 0

    Return False
```

Figure 4: The Pseudocode for the algorithm

3. Algorithm Analysis (Brute Force)

3.1 Correctness Analysis:

Claim: Given a graph g, for each vertex i in that graph, if it is not impossible to colour vertices from 0 to i - 1 without any adjacent two vertices using the same colour, up to m colours; then it is possible to implement this colouring to vertex i.

Proof (by contradiction):

Base Case: When i is equal to 0, the only colour that is needed to be painted is the first vertex. It can be coloured with any colour, ranging from 1 to m. Since there are no previously coloured vertices, the base case trivially holds as there are no adjacent vertices to violate the constraints.

Inductive Step: We are assuming that the hypothesis of vertices up to i - 1 have been coloured correctly using up to m colours, such that no two adjacent vertices share the same colour.

1. Now, considering vertex i, we are attempting to colour it with each colour starting from 1 in each turn.
2. For each colouring, we check it using the “checkSafe” function.
3. If this new painting to i holds the validity, (no adjacent vertex j < i shares the same colour) we continue with the validation.
4. If a colour can be found for vertex i, that does not create problems with other previously painted vertices, we can clearly say the hypothesis holds for i. Meaning we have extended the correct usage from i - 1 to i.
5. If no other option of colour can be found, the recursive function comes back to i -1 to try alternative colouring options.
6. The recursive calls are continued until all vertices have been checked out.
7. If a valid colouring is present for all V vertices, the program prints it and returns true.
8. If no viable colouring options are available after checking, the program returns false.

Conclusion: By using proof of induction, we establish that it is possible to colour any i - 1 vertices, or determine that no valid extension exists. Thus, the program correctly identifies a configuration, if not, reports its unavailability. This solution demonstrates the algorithm’s reliability in solving a graph colouring problem.

3.2 Time Complexity

To understand the time complexity of the algorithm, we must first start with considering the recursive nature of the functions, including the validation processes. Since our program explores all possible ways to paint V number of vertices with m number of colours, this exploration generates m^V amount of combinations. So, we can see it works in an exponential manner. The time complexity is O(m^v)

3.3 Space Complexity:

While using the brute force approach to solve the graph colouring problem, we are not required to use any more space rather than the space to store the vertices on a recursive stack. The space complexity is equal to the number of vertices. It is O(V).

4. Sample Generation (Random Instance Generator)

Input parameters to the algorithm are as follows:

- **Sample Size:** This is the number of samples that the user wants to create for the testing of the brute force algorithm.

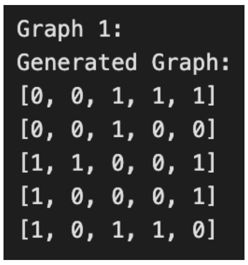
- **Number of Vertices:** This represents the number of vertices that the user wants

- **Edge Density:** This floating number represents the probability that an edge will exist between any two vertices.

Procedure:

1. The main procedure starts by iterating a graph generation loop that runs for the number of "sample_size" times. This loop generates each individual graph sample.
2. For each iteration, "generate_random_graph_adj_matrix" function is called. This function creates an empty adjacency matrix, with dimensions of "num_vertices x num_vertices", initialising all values to zero. This matrix represents the graph where rows and columns correspond to vertices and values indicate the presence of an edge between vertices.
3. Within the "generate_random_graph_adj_matrix", another nested loop iterates over each possible pair of vertices (i, j). For each pair, a random number is generated between 0 and 1. If edge_density is bigger than the number, an edge is created between those two vertices by setting the corresponding positions in the adjacency matrix to 1.

4. When a graph is generated, they are appended into a list “graphs” to get them stored. Later on, “print_graph” function is called to visually display the newly created graph by printing its adjacency matrix.
5. After all the graphs are generated, the generate_sample_graphs function returns a list of these graphs, each represented as an adjacency matrix, for further use or analysis.



```
Graph 1:
Generated Graph:
[0, 0, 1, 1, 1]
[0, 0, 1, 0, 0]
[1, 1, 0, 0, 1]
[1, 0, 0, 0, 1]
[1, 0, 1, 1, 0]
```

Figure 5: The matrix that the code generates

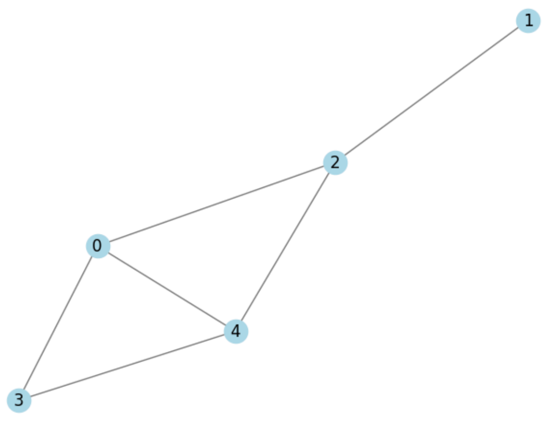
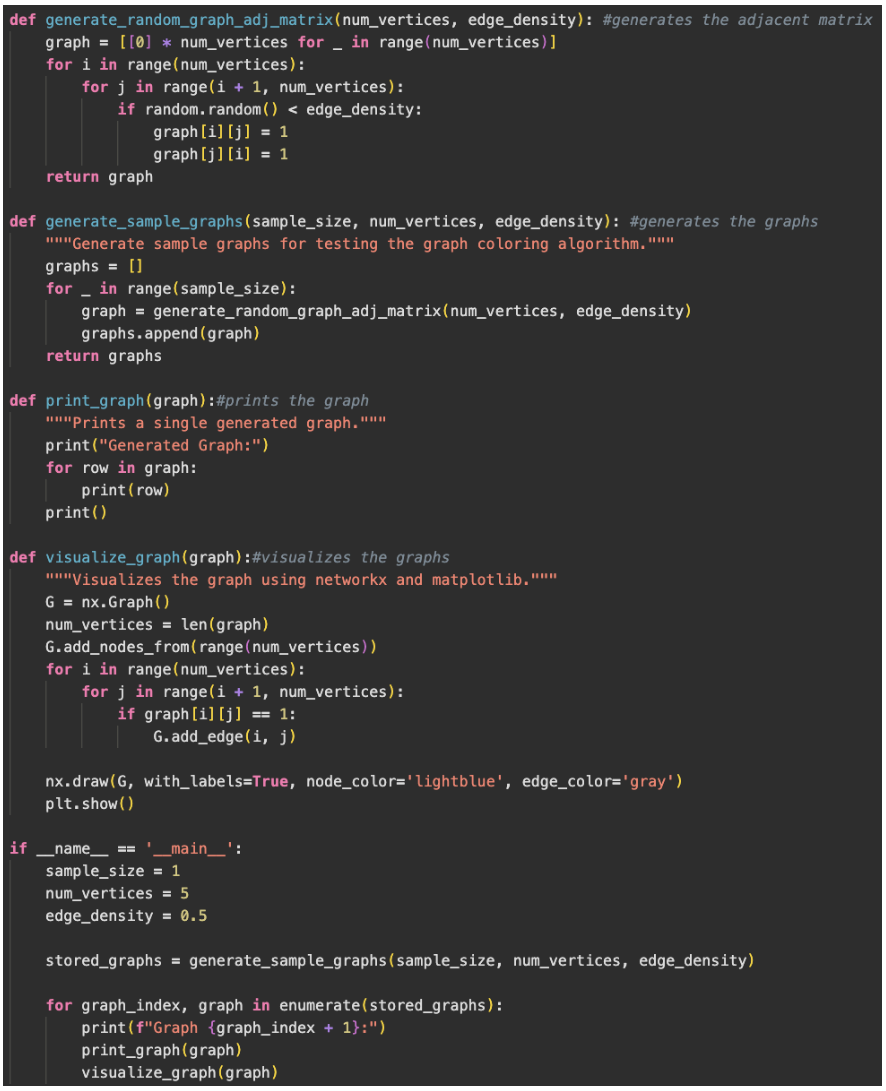


Figure 6: Matrix’s graph equivalent



```python
def generate_random_graph_adj_matrix(num_vertices, edge_density): #generates the adjacent matrix
    graph = [[0] * num_vertices for _ in range(num_vertices)]
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if random.random() < edge_density:
                graph[i][j] = 1
                graph[j][i] = 1
    return graph

def generate_sample_graphs(sample_size, num_vertices, edge_density): #generates the graphs
    """Generate sample graphs for testing the graph coloring algorithm."""
    graphs = []
    for _ in range(sample_size):
        graph = generate_random_graph_adj_matrix(num_vertices, edge_density)
        graphs.append(graph)
    return graphs

def print_graph(graph):#prints the graph
    """Prints a single generated graph."""
    print("Generated Graph:")
    for row in graph:
        print(row)
    print()

def visualize_graph(graph):#visualizes the graphs
    """Visualizes the graph using networkx and matplotlib."""
    G = nx.Graph()
    num_vertices = len(graph)
    G.add_nodes_from(range(num_vertices))
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] == 1:
                G.add_edge(i, j)

    nx.draw(G, with_labels=True, node_color='lightblue', edge_color='gray')
    plt.show()

if __name__ == '__main__':
    sample_size = 1
    num_vertices = 5
    edge_density = 0.5

    stored_graphs = generate_sample_graphs(sample_size, num_vertices, edge_density)

    for graph_index, graph in enumerate(stored_graphs):
        print(f"Graph {graph_index + 1}:")
        print_graph(graph)
        visualize_graph(graph)
```

Figure 7: Code for sample generation

5. Algorithm Implementations

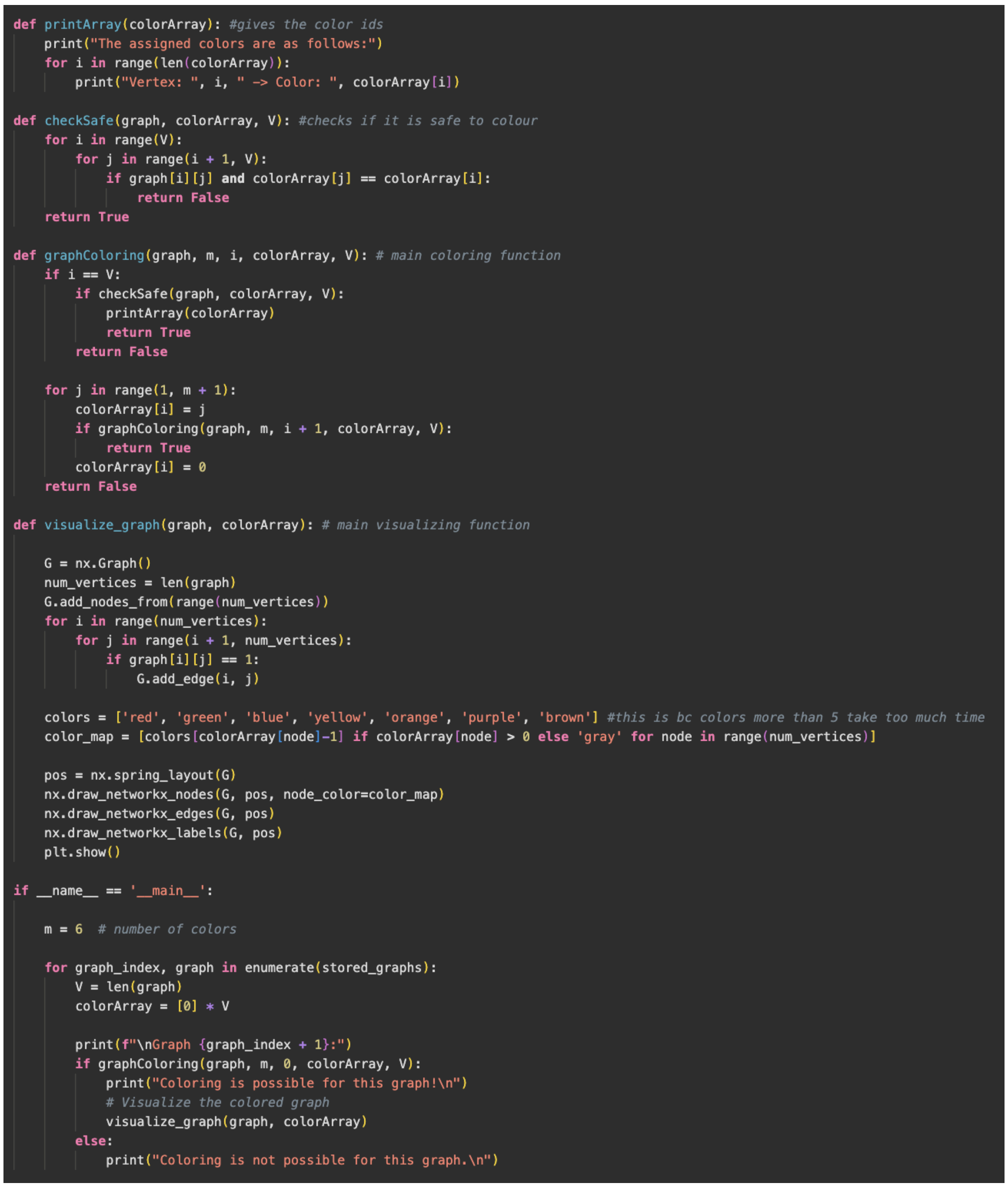
5.1 Algorithm

Input parameters to the algorithm are as follows:

- Graph: A graph from the list that we have created on the sample generation part, which consists of [V][V]
- M: An integer number for the maximum amount of colours can be used.

Procedure:

1. The procedure starts by initialising a colour array filled with zeros “colorArray”, no vertex has a colour assigned to it.
2. The graphColoring function tries assigning colours to all vertices. Ensuring no two vertices share the same colour. It iteratively assigns a colour from 1 to m to vertex i and resources to colour vertex i+1. If a valid colouring is found, the “checkSafe” function checks the safety of this colouring. If any adjacent vertice pair that have the same colour exists, the colouring is considered unsafe.
3. Later on, the “printArray” function outputs the colours of the vertices if a valid colouring is found. If colouring is found unsafe, it simply prints that colouring is not possible.
4. For the visual representation of the graphs, the “visualize_graph” function uses Matplotlib for plotting. It constructs the graphs and assigns colours from a predefined list (a list that stores colours names i.e. “red”, “blue”, “green”, etc) to the vertices.



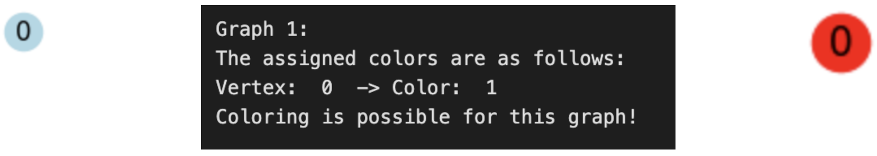
```python
def printArray(colorArray): #gives the color ids
    print("The assigned colors are as follows:")
    for i in range(len(colorArray)):
        print("Vertex: ", i, " -> Color: ", colorArray[i])

def checkSafe(graph, colorArray, V): #checks if it is safe to colour
    for i in range(V):
        for j in range(i + 1, V):
            if graph[i][j] and colorArray[j] == colorArray[i]:
                return False
    return True

def graphColoring(graph, m, i, colorArray, V): # main coloring function
    if i == V:
        if checkSafe(graph, colorArray, V):
            printArray(colorArray)
            return True
        return False

    for j in range(1, m + 1):
        colorArray[i] = j
        if graphColoring(graph, m, i + 1, colorArray, V):
            return True
        colorArray[i] = 0
    return False

def visualize_graph(graph, colorArray): # main visualizing function

    G = nx.Graph()
    num_vertices = len(graph)
    G.add_nodes_from(range(num_vertices))
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] == 1:
                G.add_edge(i, j)

    colors = ['red', 'green', 'blue', 'yellow', 'orange', 'purple', 'brown'] #this is bc colors more than 5 take too much time
    color_map = [colors[colorArray[node]-1] if colorArray[node] > 0 else 'gray' for node in range(num_vertices)]

    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, node_color=color_map)
    nx.draw_networkx_edges(G, pos)
    nx.draw_networkx_labels(G, pos)
    plt.show()

if __name__ == '__main__':

    m = 6  # number of colors

    for graph_index, graph in enumerate(stored_graphs):
        V = len(graph)
        colorArray = [0] * V

        print(f"\nGraph {graph_index + 1}:")
        if graphColoring(graph, m, 0, colorArray, V):
            print("Coloring is possible for this graph!\n")
            # Visualize the colored graph
            visualize_graph(graph, colorArray)
        else:
            print("Coloring is not possible for this graph.\n")
```

Figure 8 : Code for the brute force algorithm

5.2 Initial Test Of the Algorithm:

15 numbers of samples have been tested with the brute force algorithm, the program works without errors or failures. However, due to the exponential nature of the algorithm, after the vertex amount is more than 15, and the maximum colour amount is more than 5, it takes too much time to find the correct colours. Because of this issue, we have tried V = 15 and m = 5, and got a result in approximately 5 minutes. The created samples and several solutions are as follows:

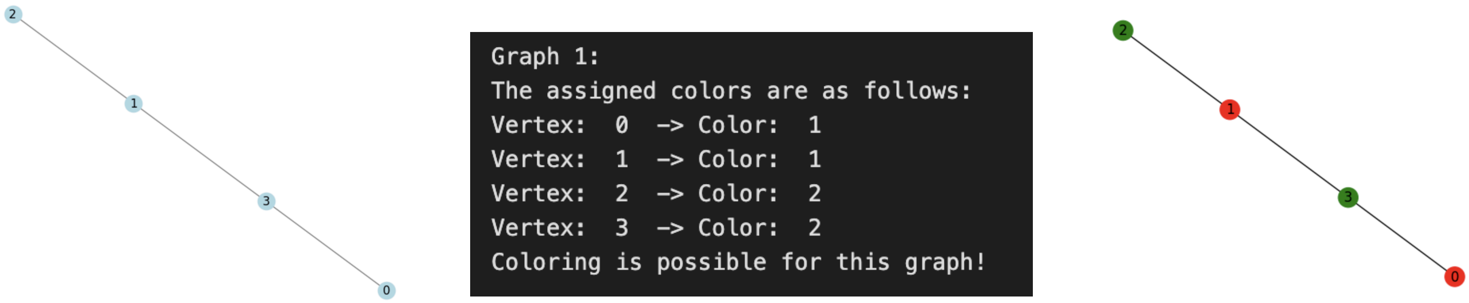
1. V = 1, m = 1



```
Graph 1:
The assigned colors are as follows:
Vertex:  0  -> Color:  1
Coloring is possible for this graph!
```

Figure 9: Initial graph, output, and the coloured graph for V = 1 and m = 1

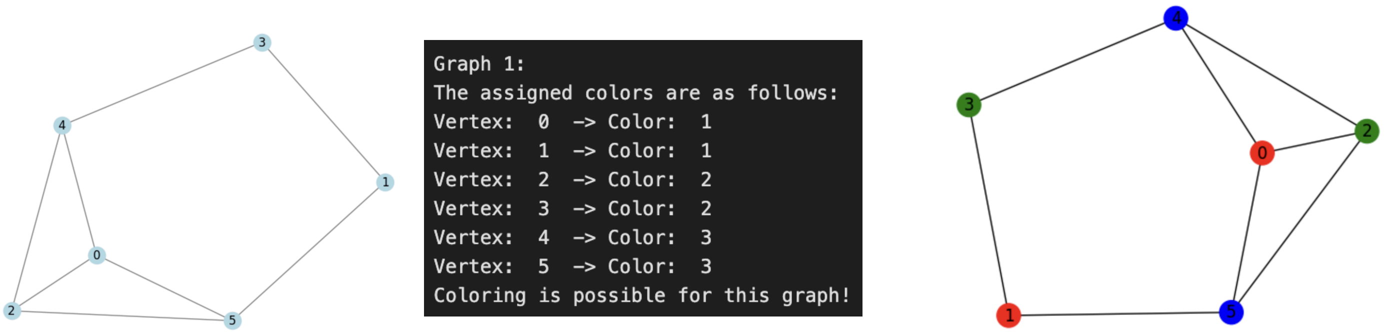
2. V = 2, m = 2
3. V = 4, m = 2



```
Graph 1:
The assigned colors are as follows:
Vertex:  0  -> Color:  1
Vertex:  1  -> Color:  1
Vertex:  2  -> Color:  2
Vertex:  3  -> Color:  2
Coloring is possible for this graph!
```

Figure 10: Initial graph, output, and the coloured graph for V =4 and m = 2

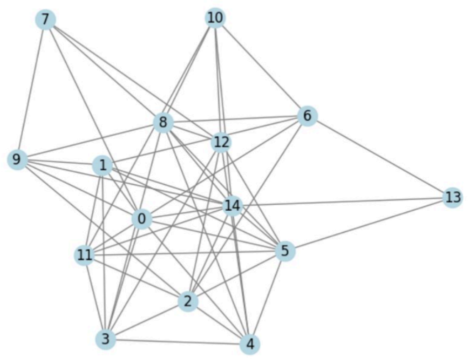
4. V = 4, m = 3
5. V = 5, m = 4
6. V = 5, m = 5
7. V = 5, m = 2
8. V = 6, m = 3
9. V = 6, m = 4

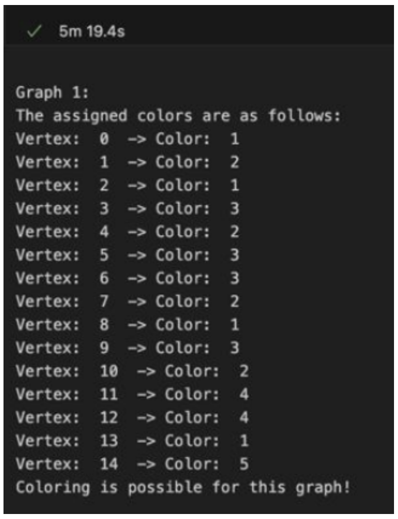


```
Graph 1:
The assigned colors are as follows:
Vertex:  0  -> Color:  1
Vertex:  1  -> Color:  1
Vertex:  2  -> Color:  2
Vertex:  3  -> Color:  2
Vertex:  4  -> Color:  3
Vertex:  5  -> Color:  3
Coloring is possible for this graph!
```

Figure 11: Initial graph, output, and the coloured graph for V =6 and m = 4

10. V = 20, m = 11
11. V = 11, m = 6
12. V = 12, m = 6
13. V = 8, m = 6
14. V = 14, m = 5
15. V = 15, m = 5





```
✓ 5m 19.4s

Graph 1:
The assigned colors are as follows:
Vertex:  0  -> Color:  1
Vertex:  1  -> Color:  2
Vertex:  2  -> Color:  1
Vertex:  3  -> Color:  3
Vertex:  4  -> Color:  2
Vertex:  5  -> Color:  3
Vertex:  6  -> Color:  3
Vertex:  7  -> Color:  2
Vertex:  8  -> Color:  1
Vertex:  9  -> Color:  3
Vertex:  10  -> Color:  2
Vertex:  11  -> Color:  4
Vertex:  12  -> Color:  4
Vertex:  13  -> Color:  1
Vertex:  14  -> Color:  5
Coloring is possible for this graph!
```

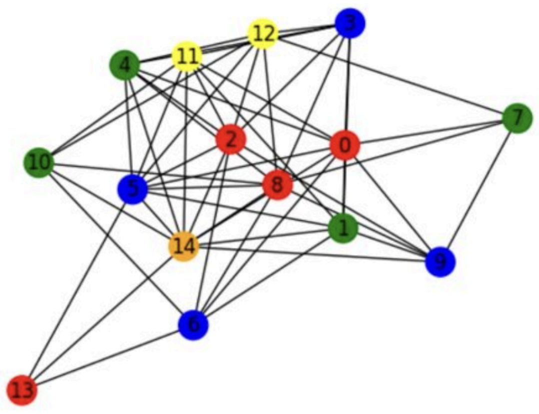


Figure 12: Initial graph, output, the coloured graph and the running time for V =15 and m = 5

References

Rutherford, Steven. 2011. "On The Computational Hardness of Graph Coloring." *University of Washington, Department of Mathematics*, June.