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:

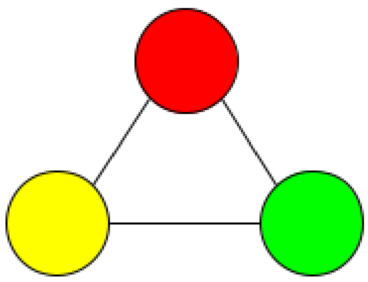


Figure 1: A graph with 3 nodes.

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.

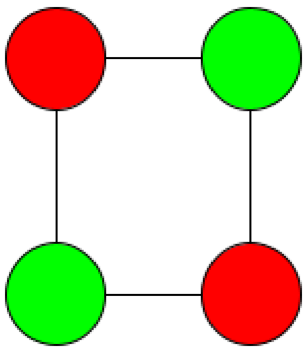


Figure 2: A graph with 4 nodes.

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.

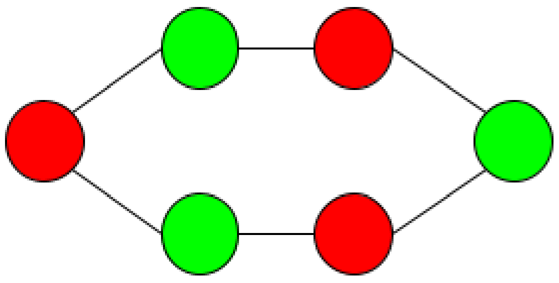


Figure 3: A graph with 6 nodes.

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)

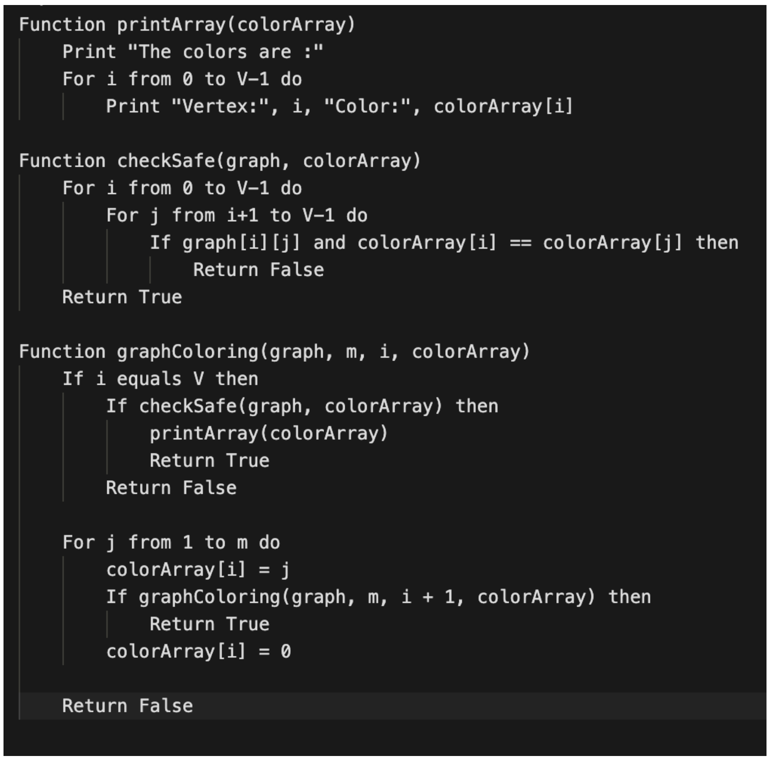
2.1 Overview: The simplest approach to solve this problem would be to generate all possible combinations of colours. After the creation of one combination, the algorithm must check if the adjacent vertices have the same colour. If the wanted result is present, the loop is broken and the combination is 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

2.3 Heuristic Algorithm

2.3.1 Overview:

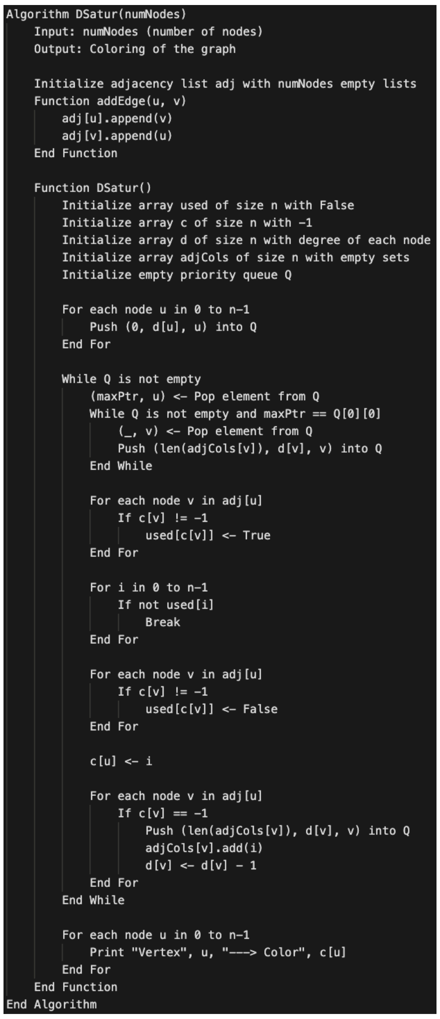
The heuristic algorithm that we have found is called DSatur, abbreviated from “Degree of Saturation”, DSatur has been developed by Swiss mathematician Daniel Brelaz in 1979. The algorithm has a specific idea on its own, as explained in its name, it generates vertex ordering by looking at the saturation numbers, and it selects the next uncolored vertex with the highest saturation degree. The saturation degree of a vertex is explained as the amount of different colours assigned to the neighbour vertices. The overall complexity of the DSatur algorithm is O(n^2), where n is the number of vertices. The analysis of the complexity will be done in its respective part. Daniel Brelaz proves his point by explaining the algorithm on bipartite graphs.[2] Considering G as a connected bipartite graph, suppose there is a vertex x that is holding a saturation level of two, meaning that it is neighbouring with two vertices of different colours. At this point, we can make 2 chains. Since G is infinite, we know that these chains will, at some point, intersect at a common vertex y, thus forming a loop. If this loop is even, the other two neighbour vertices of x will, in fact, share the same colour, which ensures that G is a bipartite graph. Consequently, his algorithm, DSatur, correctly solves these kinds of graphs, making it an exact algorithm for this problems. In addition to that, Brelaz specifies that DSatur can be also used to show if a graph is bipartite.

Steps:

1. Let v denote our uncolored vertex in G with the highest saturation degree. In the case of a tie, chose the vertex with the highest degree in the subgraph by the uncolored vertices.
2. Assigns v to colour i, where i is the smallest value from the set of colours assigned that is also not assigned to any neighbour of v.
3. If there are any remaining uncolored vertices, repeat all the steps from the start. Else, end at this step.

[2] Daniel Brelaz, "New Method to Color the Vertices of a Graph," *Communications of the ACM* 22, no. 4 (1977): 251–56.

2.3.2 Pseudocode:



```
Algorithm DSatur(numNodes)
    Input: numNodes (number of nodes)
    Output: Coloring of the graph

    Initialize adjacency list adj with numNodes empty lists
    Function addEdge(u, v)
        adj[u].append(v)
        adj[v].append(u)
    End Function

    Function DSatur()
        Initialize array used of size n with False
        Initialize array c of size n with -1
        Initialize array d of size n with degree of each node
        Initialize array adjCols of size n with empty sets
        Initialize empty priority queue Q

        For each node u in 0 to n-1
            Push (0, d[u], u) into Q
        End For

        While Q is not empty
            (maxPtr, u) <- Pop element from Q
            While Q is not empty and maxPtr == Q[0][0]
                (_, v) <- Pop element from Q
                Push (len(adjCols[v]), d[v], v) into Q
            End While

            For each node v in adj[u]
                If c[v] != -1
                    used[c[v]] <- True
            End For

            For i in 0 to n-1
                If not used[i]
                    Break
            End For

            For each node v in adj[u]
                If c[v] != -1
                    used[c[v]] <- False
            End For

            c[u] <- i

            For each node v in adj[u]
                If c[v] == -1
                    Push (len(adjCols[v]), d[v], v) into Q
                    adjCols[v].add(i)
                    d[v] <- d[v] - 1
            End For
        End While

        For each node u in 0 to n-1
            Print "Vertex", u, "---> Color", c[u]
        End For
    End Function
End Algorithm
```

Figure 5: Pseudocode for the algorithm.

The DSatur algorithm is a greedy algorithm, because at each step (vertex), it selects the optimum solution (the one with the maximum saturation degree). Although it can be understood as optimum solution, it is only locally optimal, because rather than looking at the whole graph to colour, it just selects for the vertex that it is in.

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).

3.4 Heuristic Algorithm: DSatur

3.4.1 Correctness Analysis:

Claim: The DSatur algorithm correctly colours the given graph such that not two adjacent vertices holds the same colour and it utilizes minimum number of colours in the process.

Proof: (by induction):

Base Case: For a graph named G with n (vertex count) as 1, the algorithm will select the single vertex and colour it with the first colour. Since there is not any other vertices present in the graph, it is trivially correct to say that no two adjacent vertices share the same colour.

Inductive Step: We are assuming that since the DSatur trivially works with the graphs that has only 1 vertices, it will also work on the graphs with k vertices (k>= 1), correctly colouring the graph such that no two adjacent vertices use the same colour.

1. Arrays and priority queues are initialised, uncolored vertices are inserted .
2. In the main loop, while queue is not empty, ie. there are still vertices left, select the vertex u with the maximum number of saturation.
3. The smallest available colour, that is not being used by the neighbors, is assigned to u.
4. Here, by induction hypothesis, we assume that the algorithm correctly coloured this vertex. After the kth vertex, algorithm processes the (k+1)th vertex u. It also selects k+1 according to the highest saturation number.
5. Here, again by hypothesis, we assume that the vertexes are coloured correctly until now. The next vertex will be coloured using the colour with the smallest value.
6. After the colouring of one vertex, the data structure is updated with the new saturation values. Again by hypothesis, we know that the remaining vertices will be coloured correctly.

Conclusion: As shown in the proof by induction, the DSatur algorithm correctly finds and paints the vertices in a way such that no two adjacent vertices share the same colour and in the process, it tries to minimise the amount of colour used. The idea of using a value such as saturation number is what differentiates it from a basic greedy algorithm.

3.4.2 Time Complexity: From the pseudocode, we can understand that the initialization process take O(nlogn) time because of the priority queue initialization takes O(n) and each insertion takes O(logn). To understand the complexity of the main loop, we need to think about the worst case. While checking the queues and assigning colours all take O(nlogn) time, due to the fact that finding the smallest unused colour may require for us to check each vertex, its time complexity is O(n^2). This makes the total time complexity of the DSatur algorithm as $\Theta(n$^2).

3.4.3 Space Complexity: Also from the pseudocode, we can understand the complexity of adjacent colours array may go up to n^2, due to the fact that if, in the worst case each set(n) can contain up to n elements, if all adjacent elements has different colours. Comparing this complexity with the worst cases of other arrays, we see that no matter what the situation is, priority queue that stores all the vertices, the array that holds the degrees of vertices, the array that holds the colour information for each vertex, each hold a space less than O(n^2). Thus, making the worst case space complexity of the DSatur O(V^2).

4. Sample Generation (Random Instance Generator)

Input parameters to the algorithm are as follows:

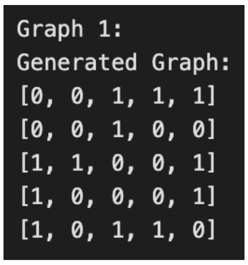
- **Sample Size:** This is the number of samples that the user wants to create for the testing of the brute force algorithm.

- **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 6: The matrix that the code generates

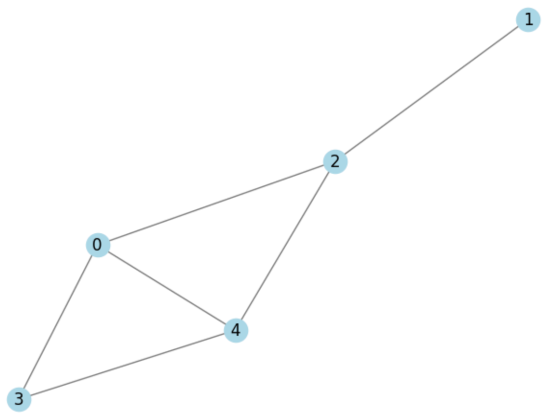
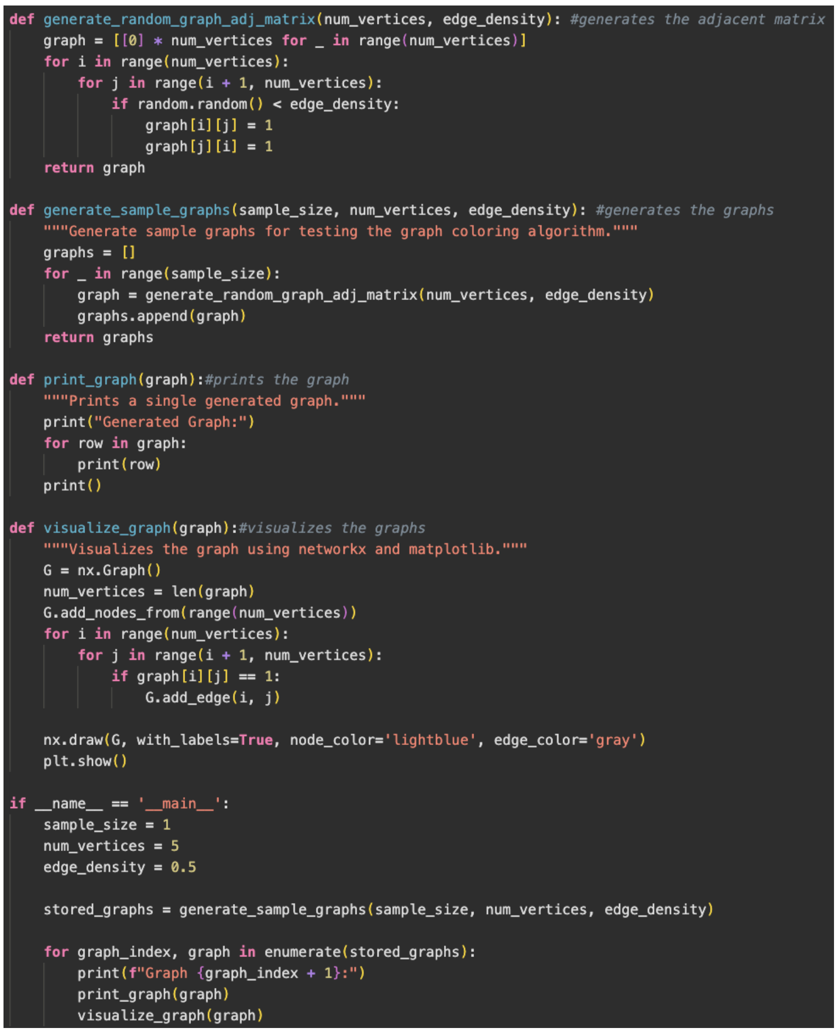


Figure 7: Matrix's graph equivalent



```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 8: Code for sample generation

5. Algorithm Implementations

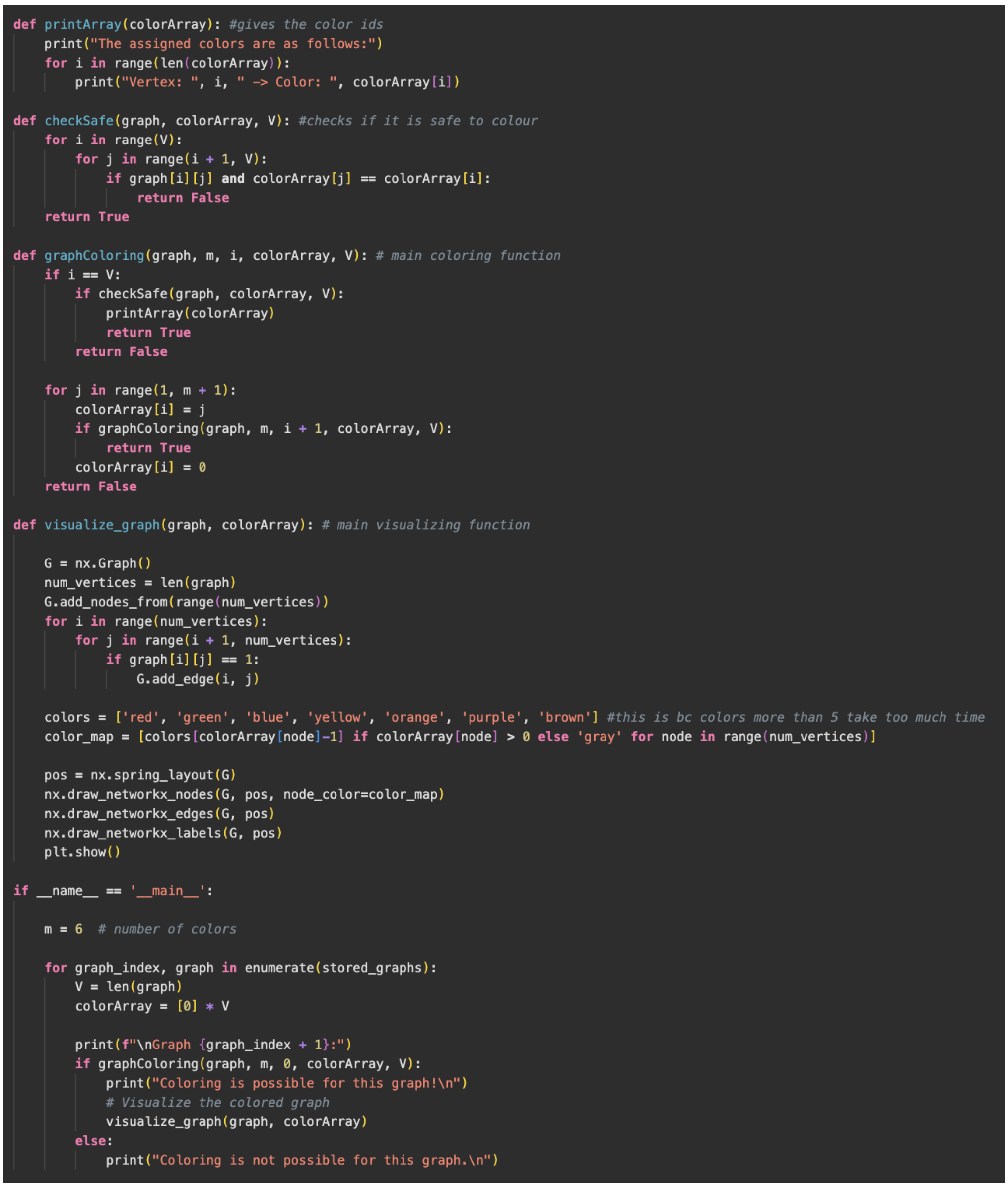
5.1 Algorithm (Brute Force)

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.



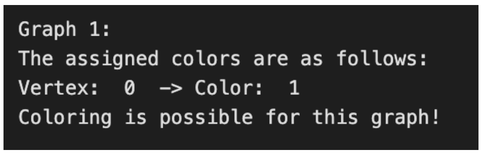
```python
def printArray(colorArray): #gives the color ids
    print("The assigned colors are as follows:")
    for i in range(len(colorArray)):
        print("Vertex: ", i, " -> Color: ", colorArray[i])

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 9 : 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:

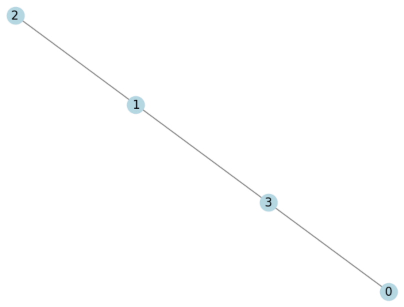
1. V = 1, m = 1

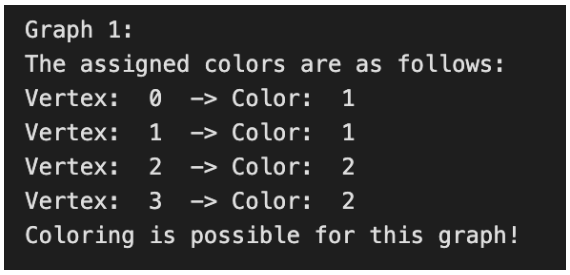


```
Graph 1:
The assigned colors are as follows:
Vertex:  0  -> Color:  1
Coloring is possible for this graph!
```

Figure 10: 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!
```

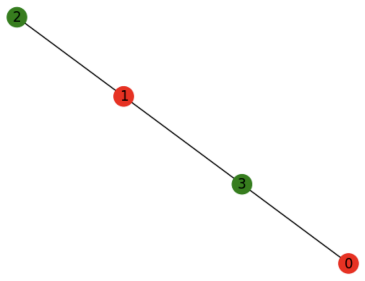
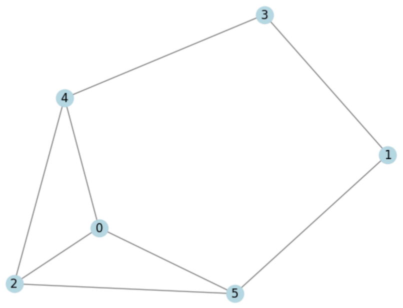
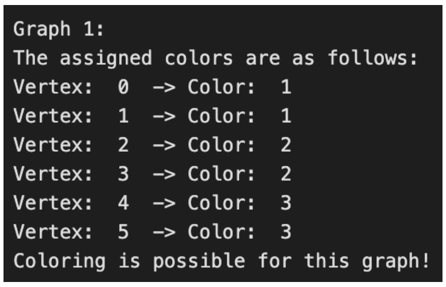


Figure 11: Initial graph, output, and the coloured graph for V =4 and m = 2

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!
```

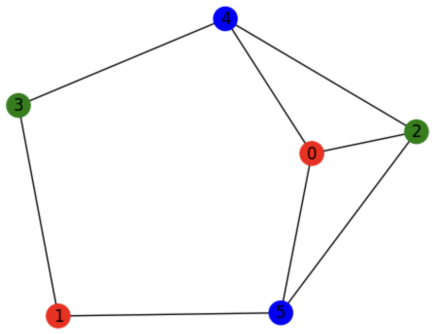
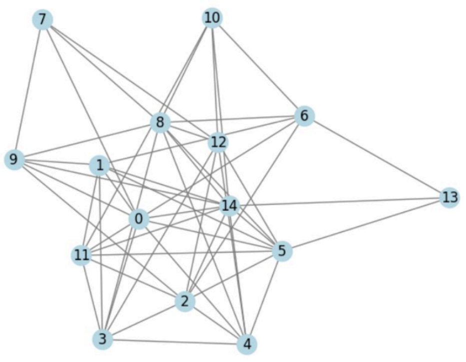
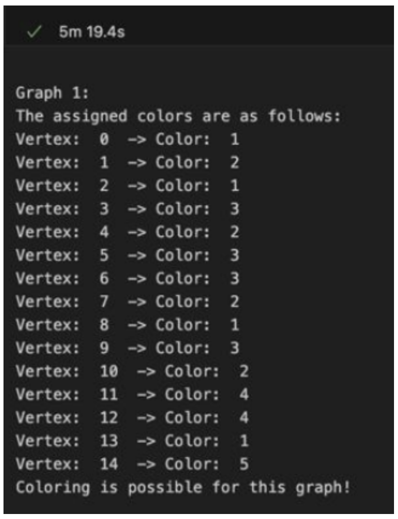


Figure 12: Initial graph, output, and the coloured graph for V =6 and m = 4

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!
```

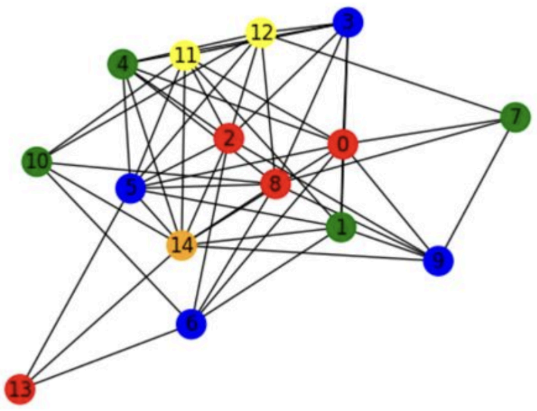


Figure 13: Initial graph, output, the coloured graph and the running time for V =15 and m = 5

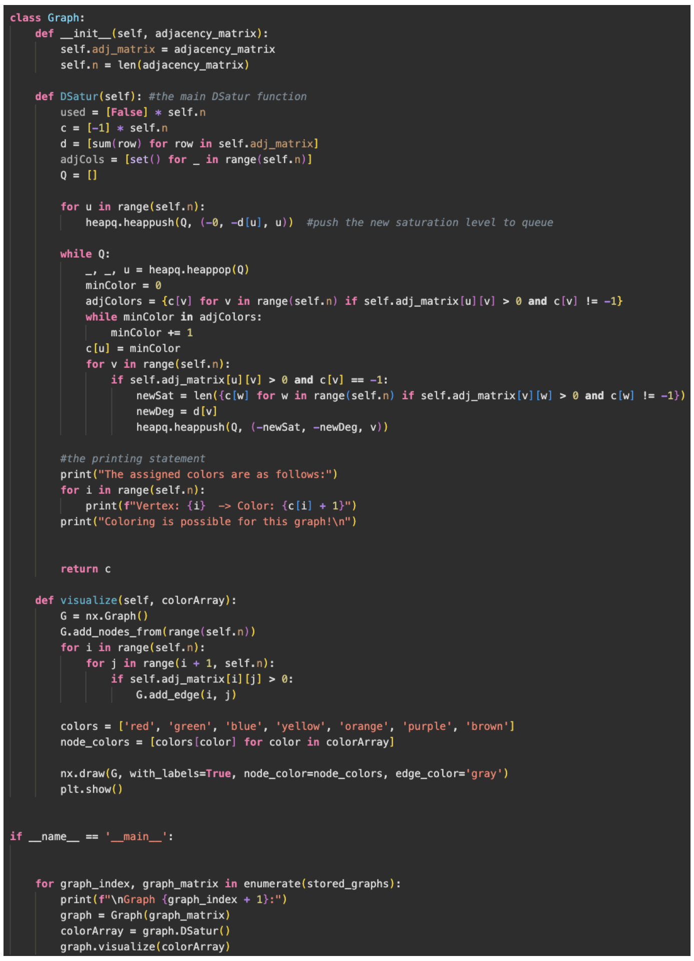
5.2 Heuristic Algorithm

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]

Procedure:

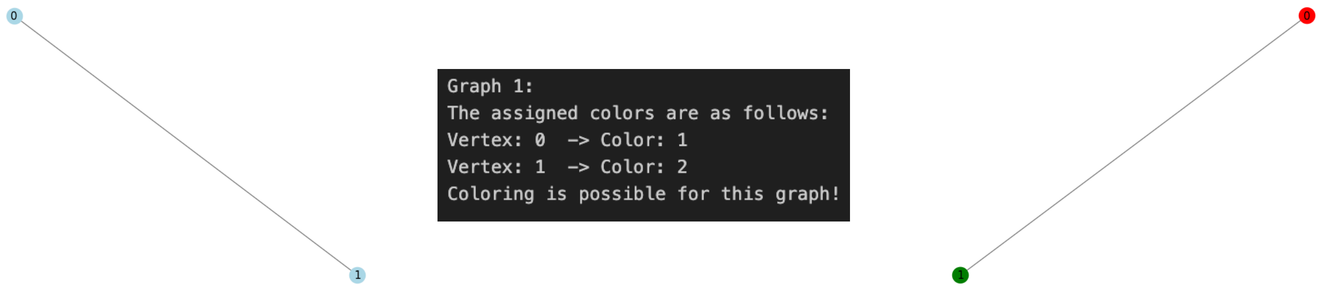
1. The algorithm starts by initializing the data strcutures and each vertex is pushed into the priority queue with with a saturation of 0.
2. The vertices are processes on by one from the queue, starting with the vertex with the highest saturation.
3. For each vertex, the algorithm determines the minimum color that is not used by the adjacent vertices by checking the adjColors.
4. After the assignment of the colour to a vertex, it is updated in the respected data structures.
5. The priority queue is updated to ensure that the changes on the saturation and the degrees are noted.
6. Once all the vertices have been coloured, the algorithm terminates. The output is printed.
7. After the DSatur terminates, our "visualize" function, utilizing the NetworkX library displays the graph.



```python
class Graph:
    def __init__(self, adjacency_matrix):
        self.adj_matrix = adjacency_matrix
        self.n = len(adjacency_matrix)

    def DSatur(self): #the main DSatur function
        used = [False] * self.n
        c = [-1] * self.n
        d = [sum(row) for row in self.adj_matrix]
        adjCols = [set() for _ in range(self.n)]
        Q = []

        for u in range(self.n):
            heapq.heappush(Q, (-0, -d[u], u))  #push the new saturation level to queue

        while Q:
            _, _, u = heapq.heappop(Q)
            minColor = 0
            adjColors = {c[v] for v in range(self.n) if self.adj_matrix[u][v] > 0 and c[v] != -1}
            while minColor in adjColors:
                minColor += 1
            c[u] = minColor
            for v in range(self.n):
                if self.adj_matrix[u][v] > 0 and c[v] == -1:
                    newSat = len({c[w] for w in range(self.n) if self.adj_matrix[v][w] > 0 and c[w] != -1})
                    newDeg = d[v]
                    heapq.heappush(Q, (-newSat, -newDeg, v))

        #the printing statement
        print("The assigned colors are as follows:")
        for i in range(self.n):
            print(f"Vertex: {i}  -> Color: {c[i] + 1}")
        print("Coloring is possible for this graph!\n")


        return c

    def visualize(self, colorArray):
        G = nx.Graph()
        G.add_nodes_from(range(self.n))
        for i in range(self.n):
            for j in range(i + 1, self.n):
                if self.adj_matrix[i][j] > 0:
                    G.add_edge(i, j)

        colors = ['red', 'green', 'blue', 'yellow', 'orange', 'purple', 'brown']
        node_colors = [colors[color] for color in colorArray]

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


if __name__ == '__main__':


    for graph_index, graph_matrix in enumerate(stored_graphs):
        print(f"\nGraph {graph_index + 1}:")
        graph = Graph(graph_matrix)
        colorArray = graph.DSatur()
        graph.visualize(colorArray)
```

Figure 14: Code of DSATUR algorithm

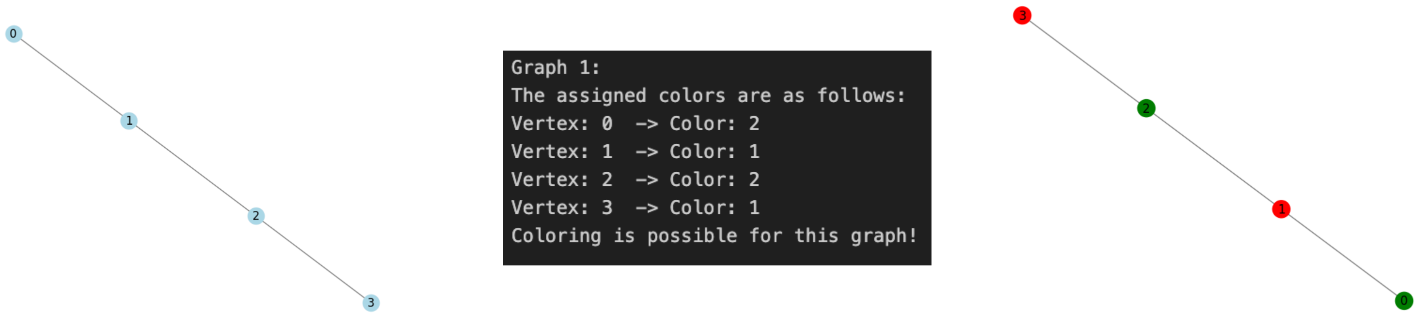
1. V = 1
2. V = 2



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

Figure 15: Initial graph, output, and the coloured graph for V =2

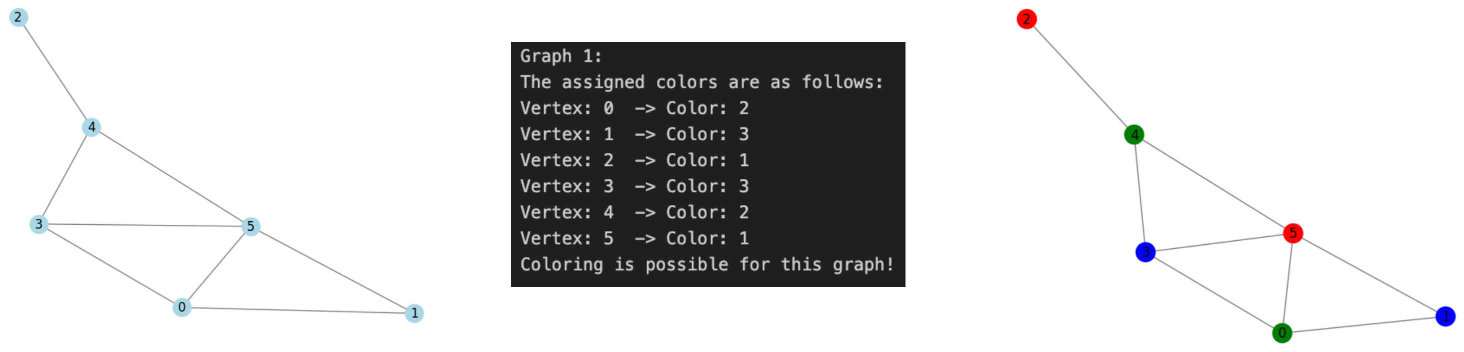
3. V = 3
4. V = 4



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

Figure 16: Initial graph, output, and the coloured graph for V =4

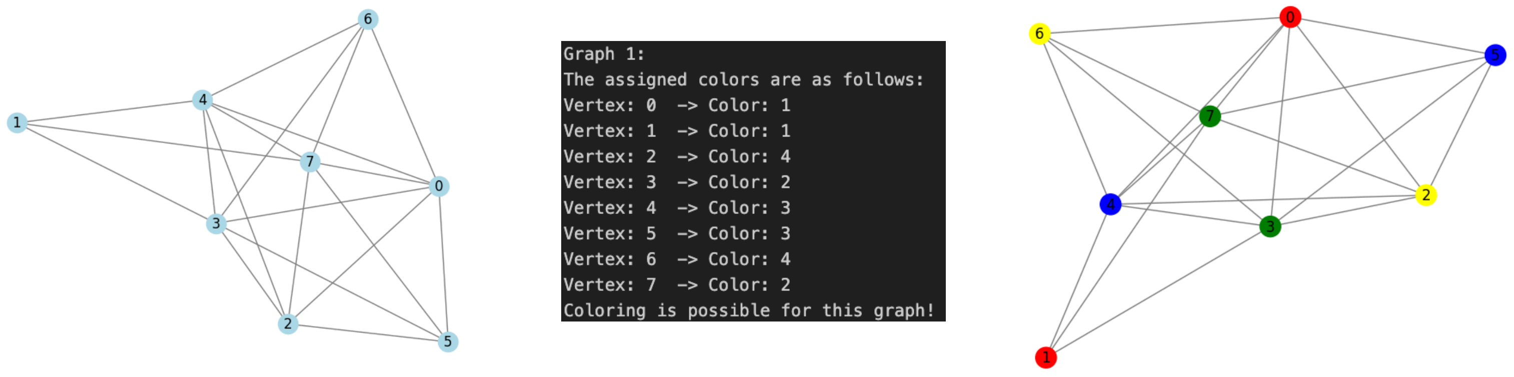
5. V = 5
6. V = 6



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

Figure 17: Initial graph, output, and the coloured graph for V =6

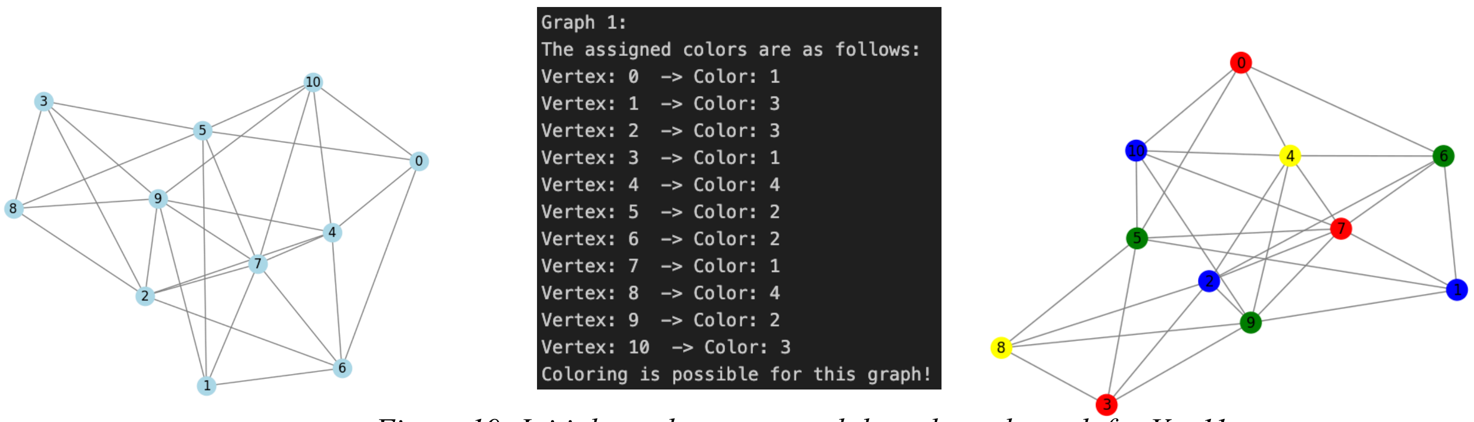
7. V = 7
8. V = 8



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

Figure 18: Initial graph, output, and the coloured graph for V =8

9. V = 9
10. V = 11



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

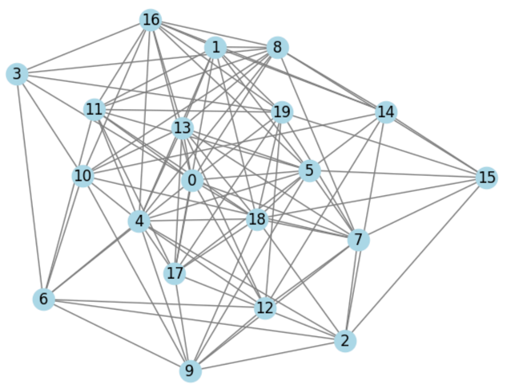
Figure 19: Initial graph, output, and the coloured graph for V =11

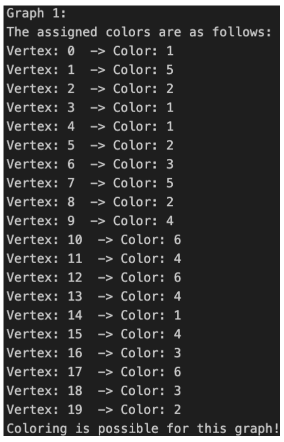
11. V = 14

12. V = 15

13. V = 18

14. V = 20





```
Graph 1:
The assigned colors are as follows:
Vertex: 0  -> Color: 1
Vertex: 1  -> Color: 5
Vertex: 2  -> Color: 2
Vertex: 3  -> Color: 1
Vertex: 4  -> Color: 1
Vertex: 5  -> Color: 2
Vertex: 6  -> Color: 3
Vertex: 7  -> Color: 5
Vertex: 8  -> Color: 2
Vertex: 9  -> Color: 4
Vertex: 10  -> Color: 6
Vertex: 11  -> Color: 4
Vertex: 12  -> Color: 6
Vertex: 13  -> Color: 4
Vertex: 14  -> Color: 1
Vertex: 15  -> Color: 4
Vertex: 16  -> Color: 3
Vertex: 17  -> Color: 6
Vertex: 18  -> Color: 3
Vertex: 19  -> Color: 2
Coloring is possible for this graph!
```

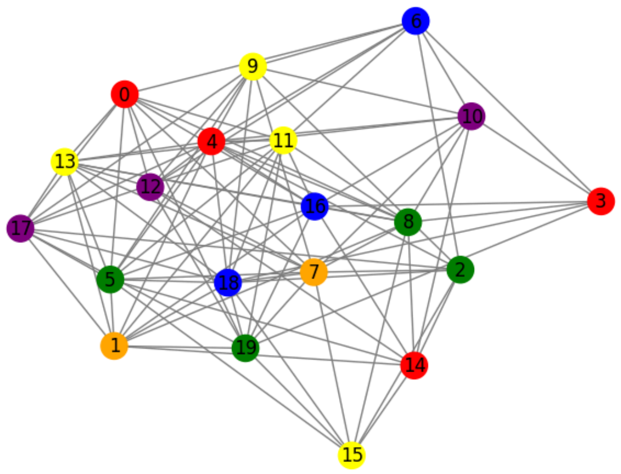
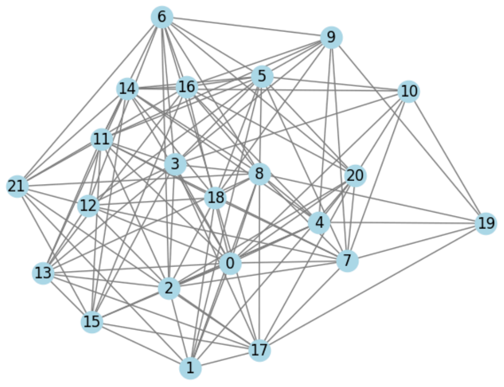
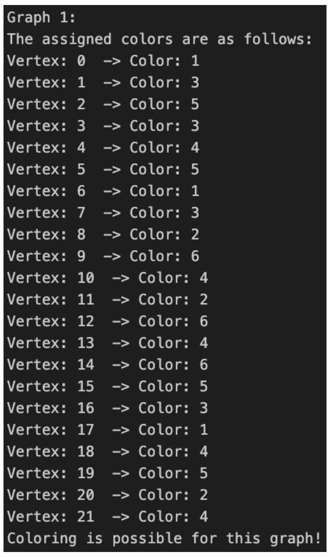


Figure 20: Initial graph, output, and the coloured graph for V =20

15. V = 22





```
Graph 1:
The assigned colors are as follows:
Vertex: 0  -> Color: 1
Vertex: 1  -> Color: 3
Vertex: 2  -> Color: 5
Vertex: 3  -> Color: 3
Vertex: 4  -> Color: 4
Vertex: 5  -> Color: 5
Vertex: 6  -> Color: 1
Vertex: 7  -> Color: 3
Vertex: 8  -> Color: 2
Vertex: 9  -> Color: 6
Vertex: 10  -> Color: 4
Vertex: 11  -> Color: 2
Vertex: 12  -> Color: 6
Vertex: 13  -> Color: 4
Vertex: 14  -> Color: 6
Vertex: 15  -> Color: 5
Vertex: 16  -> Color: 3
Vertex: 17  -> Color: 1
Vertex: 18  -> Color: 4
Vertex: 19  -> Color: 5
Vertex: 20  -> Color: 2
Vertex: 21  -> Color: 4
Coloring is possible for this graph!
```

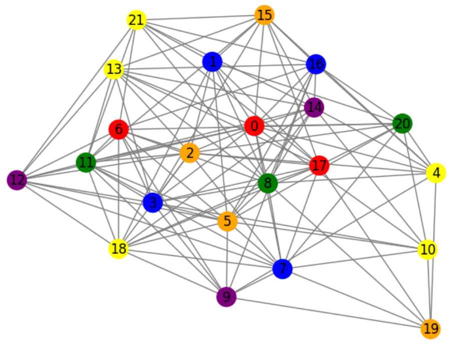


Figure 21: Initial graph, output, and the coloured graph for V =22

6. Experimental Analysis of The Performance (Performance Testing)

In this part of the term project, we will explain the findings of our experimental analysis of performance. We were urged to conduct our own analysis, because the analysis conducted in section 3.2 does not specifically state performance in real life, it just gives a worst case scenario of time and space complexity. We will present the findings of our experiment results of our heuristic DSatur algorithm both as a graph and code. For the sake of reliability, at each input size, we are making the number of test samples as 200 (k = 200), and the run amount as 10, meaning we run each size with 200 samples for 10 times in a row. The input size that we use here explains the concept of vertex amount in a graph.

In this part, we have used step sizes 1, 10, 20, 50, 75, 100, and used a logarithmic plot to visualise the results. Our figure shows the results of these sizes. The equation for the fitted line is y=0.0000032 . x^1.42 . Given that T(n) = n^a + lower order terms, where a is the slope in the log-log plot. The slope 1.42, meaning the practical time complexity of the DSatur algorithm is O(n^1.42). We can see that this result is better than what previously defined, O(N^2).

We must also consider that we used an iteration size of 200 and a confidence level of %90. All average values for our input sizes, fall within the range of mean + standard error * t value with our selected confidence interval. The narrowness being always below 0.1 indicates the reliability of our results.

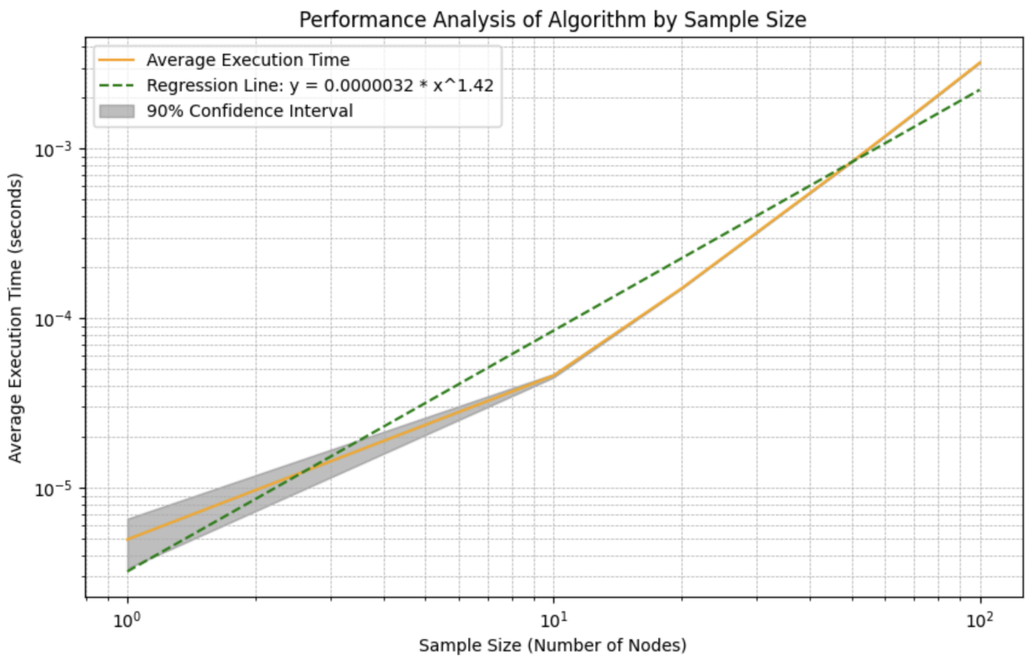


Figure 22: The Graph of performance analysis of algorithm by sample size

7. Experimental Analysis of the Quality

Input Size	**Brute Force Algorithm Minimum Color Count**	**Heuristic Algorithm Minimum Color Count**	**Ratio of Heuristic/Exact**
1	**1**	**1**	**1**
4	**2**	**2**	**1**
5	**2**	**2**	**1**
6	**3**	**3**	**1**
8	**3**	**4**	**1.33**
10	**3**	**4**	**1.33**
12	**5**	**6**	**1.2**
13	**5**	**6**	**1.2**
15	**5**	**7**	**1.4**

Table 1

The Table 1 compares the brute force and heuristic (DSATUR) algorithms with respect to the minimum number of colours used for various input sizes. For smaller input sizes, it can be seen that both brute force and heuristic algorithms perform identically, outputting the same colour count. The ratio of heuristic/exact is 1 in these cases. When the input size gets bigger, the heuristic algorithm uses more colour than the brute force algorithm, reflected in the increasing ratio. From this result, it can be said that the heuristic algorithm does not always yield an optimal solution. Optimal refers to whether it identifies the minimum cardinality vertex cover here. Moreover, because of the complexity of the brute force algorithm, it was impossible to run a sample size larger than 15 in a reasonable amount of time. As a result, it can be said that even though the heuristic algorithm looks less optimal in minimum colour count, it is better for huge input sizes due to the brute force's running time. The graph (figure 23) is the representation of the obtained result from the trials. The trend line indicates that the heuristic algorithm tends to provide slightly less optimal solutions as the input size increases. The confidence interval highlights the variability in the ratio, showing that the heuristic algorithm's performance varies more significantly for larger input sizes.

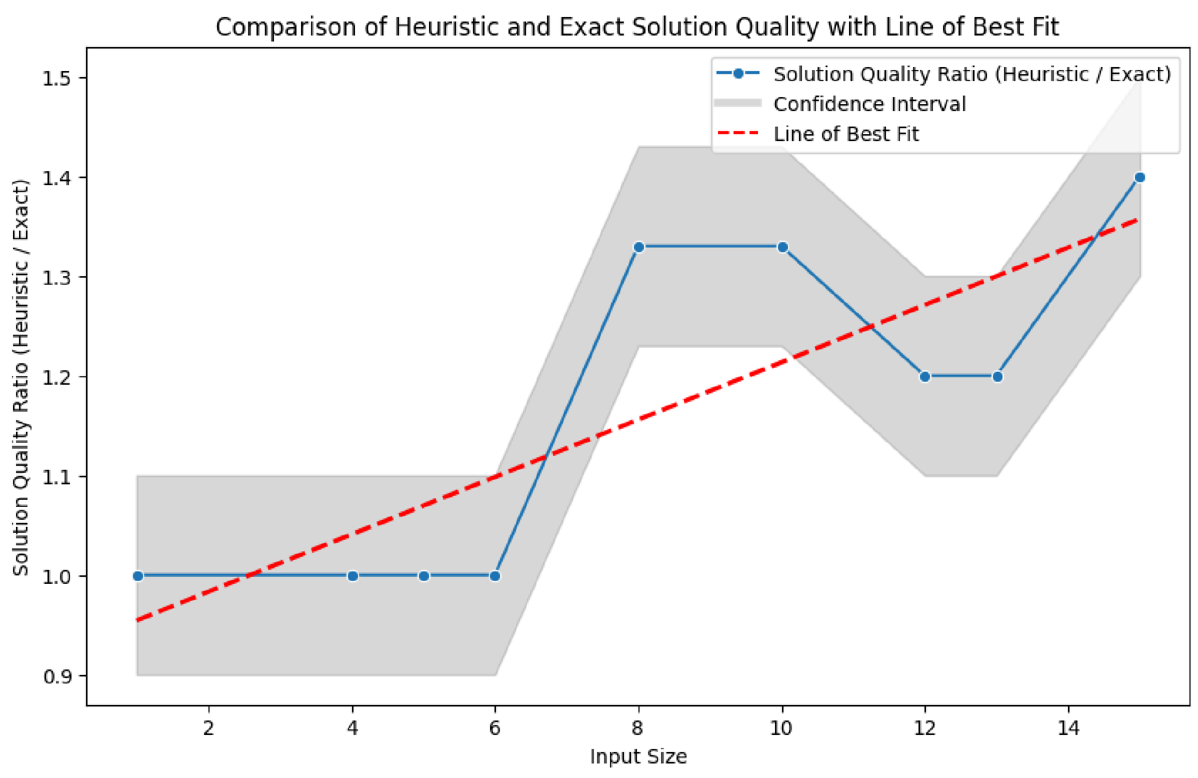


Figure 23: Comparison of Heuristic and Exact Solution Quality with Line of Best Fit Graph

The graph below (figure 24) shows the comparison of the heuristic and brute force algorithms for a graph with a single vertex over 200 iterations. As seen, the solution quality ratio remains at 1 throughout all iterations. This indicates that for graphs with only one vertex, both the heuristic and exact algorithms consistently produce the same minimum vertex cover, which is always 1. The results clearly demonstrate the reliability of both algorithms in this specific case, as no discrepancies occur between the heuristic and exact solutions.

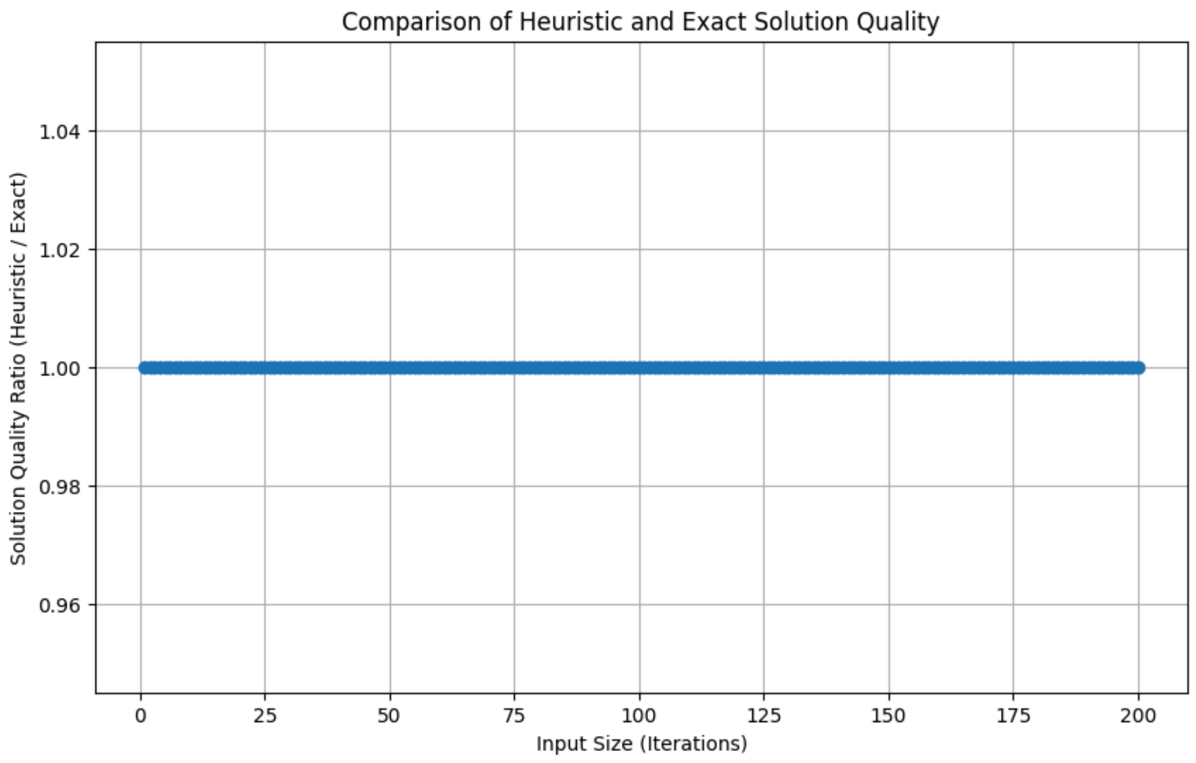


Figure 24: Comparison of Heuristic and Exact Solution Quality Graph

In summary, the brute force and heuristic (DSATUR) algorithms perform identically for smaller input sizes, yielding the same minimum color counts. However, as the input size increases, the heuristic algorithm tends to use more colors than the brute force approach, reflected in the rising heuristic/exact ratio. For graphs with a single vertex, both algorithms consistently produce the same minimum vertex cover, demonstrating their reliability in this specific scenario. So, the size and the structure of the graph affects the performance of different algorithms significantly.

8. Experimental Analysis of the Correctness of the Implementation (Functional Testing)

Black Box Testing for the Heuristic DSatur Algorithm of Graph Colouring

Black box testing is the type of software testing technique that tests the integrity of the code without knowing the inner workings. In this part of the homework, we applied 6 test cases to our algorithm.

Test Case 1: Empty Graph

- This case is for understanding the behaviour of the algorithm when the given graph is empty.
- Graph: {}
- Expected Result: []
- Since there isn’t any vertices, the result should be empty

Test Case 2: A Graph Without Any Edge

- This case is for understanding the behaviour of the algorithm when the given graph does not have any edge.
- Graph: {A': [], 'B': [], 'C': []}
- Expected Result: [0,0,0]
- Since the vertices are all on their own, the algorithm should return a set containing all the vertices.

Test Case 3: A Graph With Both Multiple Vertices and Edges

- This case is for understanding the behaviour of the algorithm when the given graph has both multiple vertices and edges.
- Graph: {'A': ['B', 'C'], 'B': ['A'], 'C': ['A']}
- Expected Result: {'A'}
- Here A gets one colour, and the other ones get a different colour.

Test Case 4: A Graph With Isolated Vertices

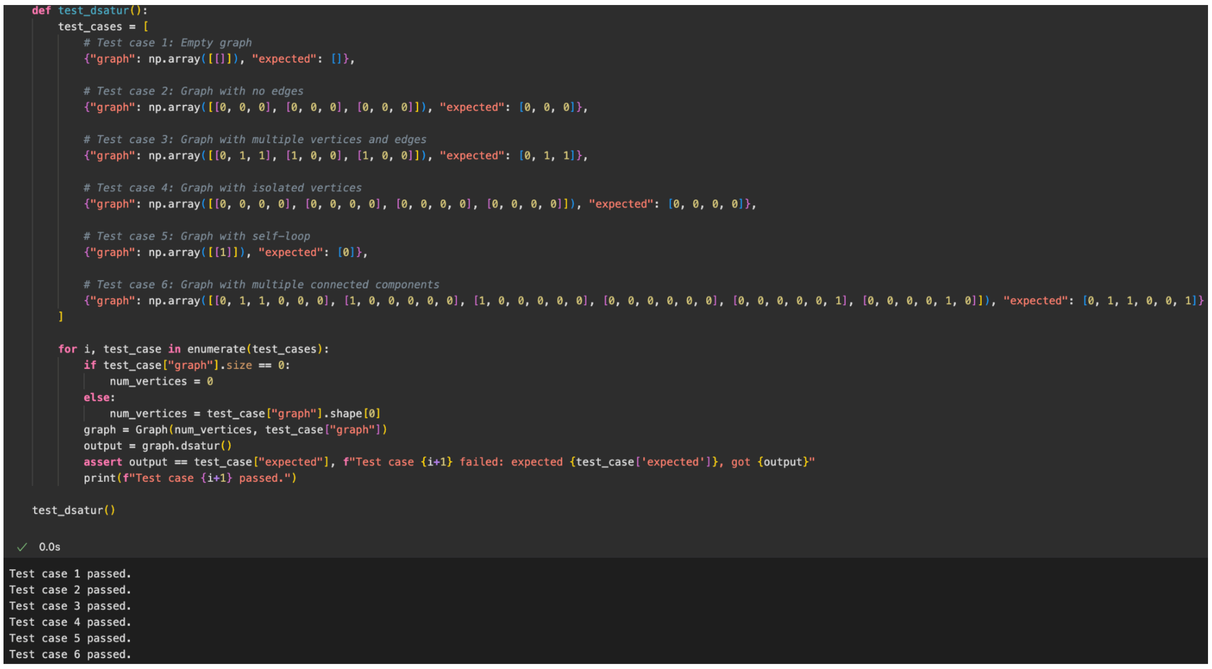
- This case is for understanding the behaviour of the algorithm when the given has isolated vertices.
- Graph: {'A': [], 'B': [], 'C': [], 'D': []}
- Expected Result: [0,0,0,0]
- Since the vertices are all isolated, they all should get the same colour.

Test Case 5: A Graph With Self Loop

- This case is for understanding the behaviour of the algorithm when the given graph is empty.
- Graph: {'A': ['A']}
- Expected Result: [0]
- Since there is a single vertex, it does not change the colour assignment.

Test Case 6: A Graph with Multiple Connections

- This case is for understanding the behaviour of the algorithm when the given graph has many connections.
- Graph: {'A': ['B', 'C'], 'B': ['A'], 'C': ['A'], 'D': [], 'E': ['F'], 'F': ['E']}
- Expected Result: [0,1,1,0,0,1]
- Each component here is coloured independently.



```python
def test_dsatur():
    test_cases = [
        # Test case 1: Empty graph
        {"graph": np.array([[]]), "expected": []},

        # Test case 2: Graph with no edges
        {"graph": np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]]), "expected": [0, 0, 0]},

        # Test case 3: Graph with multiple vertices and edges
        {"graph": np.array([[0, 1, 1], [1, 0, 0], [1, 0, 0]]), "expected": [0, 1, 1]},

        # Test case 4: Graph with isolated vertices
        {"graph": np.array([[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]), "expected": [0, 0, 0, 0]},

        # Test case 5: Graph with self-loop
        {"graph": np.array([[1]]), "expected": [0]},

        # Test case 6: Graph with multiple connected components
        {"graph": np.array([[0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 1, 0]]), "expected": [0, 1, 1, 0, 0, 1]}
    ]

    for i, test_case in enumerate(test_cases):
        if test_case["graph"].size == 0:
            num_vertices = 0
        else:
            num_vertices = test_case["graph"].shape[0]
        graph = Graph(num_vertices, test_case["graph"])
        output = graph.dsatur()
        assert output == test_case["expected"], f"Test case {i+1} failed: expected {test_case['expected']}, got {output}"
        print(f"Test case {i+1} passed.")

test_dsatur()
```

✓ 0.0s

```
Test case 1 passed.
Test case 2 passed.
Test case 3 passed.
Test case 4 passed.
Test case 5 passed.
Test case 6 passed.
```

Figure 25: Test Cases of Blackbox Training

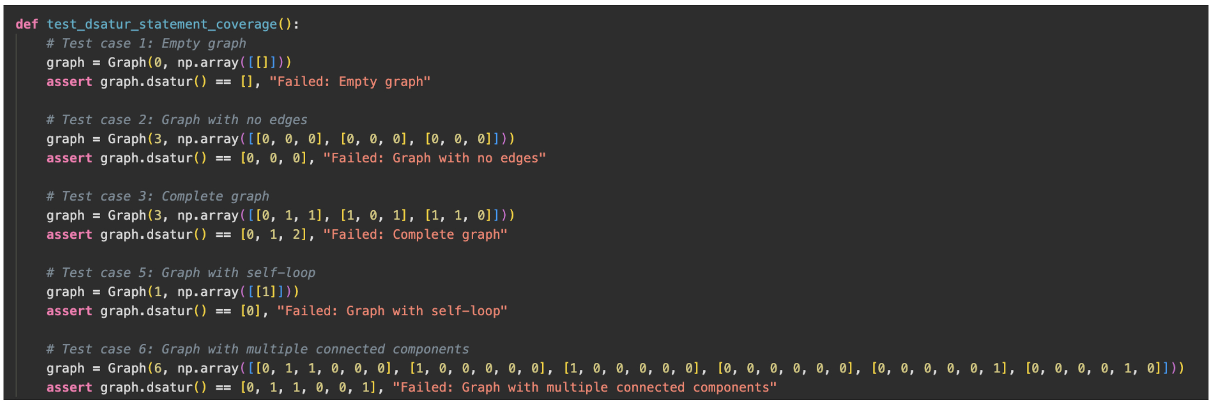
Conclusion:

In our figure, we can see the implementation of our black box testing for the heuristic DSatur algorithm. As we can understand from the result, we have passed all the test cases. Black box testing of our algorithm ensures that the correctness of the proof we have provided in the section 3.4. By considering the cases we have used on the testing part, the results validate its capacity to colour vertices correctly.

White Box Testing for the Heuristic Vertex Colour Algorithm DSatur

Statement Coverage:

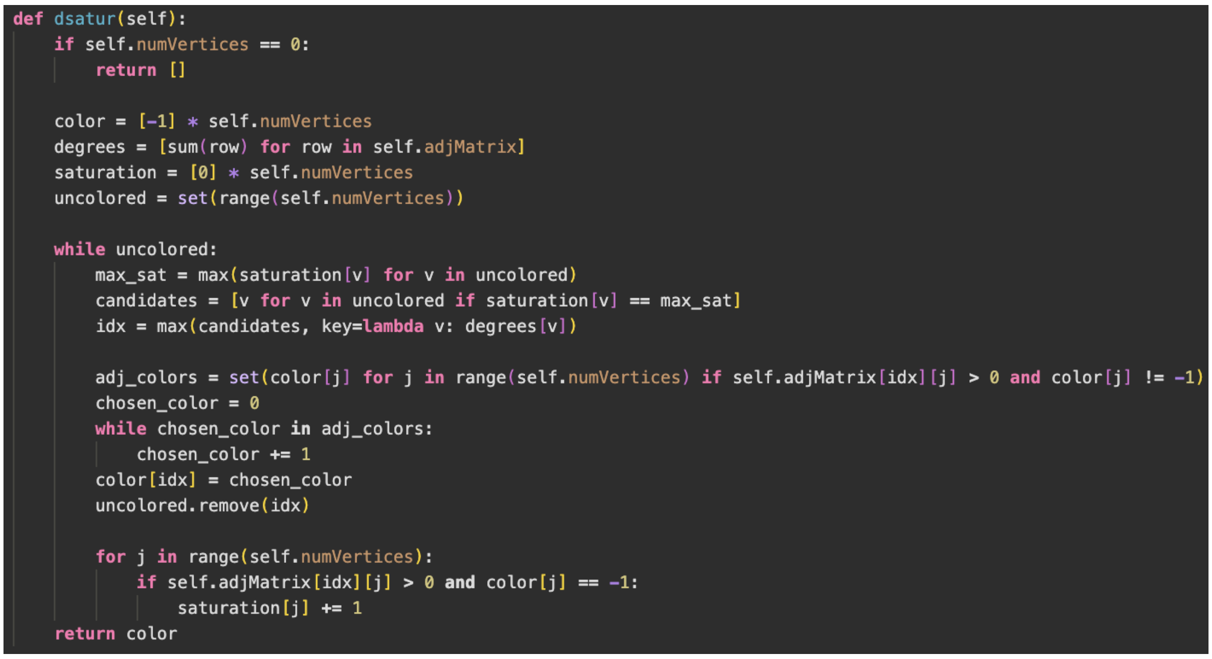
Statement coverage is used to show that the lines in the algorithm are executed at least once. For our algorithm DSatur, this is used to show each branches are covered. To check this, we implemented test cases such as:



```python
def test_dsatur_statement_coverage():
    # Test case 1: Empty graph
    graph = Graph(0, np.array([[]]))
    assert graph.dsatur() == [], "Failed: Empty graph"

    # Test case 2: Graph with no edges
    graph = Graph(3, np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]]))
    assert graph.dsatur() == [0, 0, 0], "Failed: Graph with no edges"

    # Test case 3: Complete graph
    graph = Graph(3, np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]]))
    assert graph.dsatur() == [0, 1, 2], "Failed: Complete graph"

    # Test case 5: Graph with self-loop
    graph = Graph(1, np.array([[1]]))
    assert graph.dsatur() == [0], "Failed: Graph with self-loop"

    # Test case 6: Graph with multiple connected components
    graph = Graph(6, np.array([[0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 1, 0]]))
    assert graph.dsatur() == [0, 1, 1, 0, 0, 1], "Failed: Graph with multiple connected components"
```

Figure 26: Test Cases of Statement Coverage

To check if we utilise every if and while statement in the algorithm:



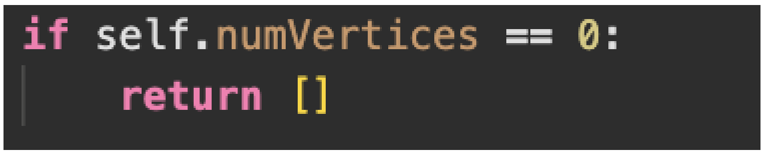
```python
def dsatur(self):
    if self.numVertices == 0:
        return []

    color = [-1] * self.numVertices
    degrees = [sum(row) for row in self.adjMatrix]
    saturation = [0] * self.numVertices
    uncolored = set(range(self.numVertices))

    while uncolored:
        max_sat = max(saturation[v] for v in uncolored)
        candidates = [v for v in uncolored if saturation[v] == max_sat]
        idx = max(candidates, key=lambda v: degrees[v])

        adj_colors = set(color[j] for j in range(self.numVertices) if self.adjMatrix[idx][j] > 0 and color[j] != -1)
        chosen_color = 0
        while chosen_color in adj_colors:
            chosen_color += 1
        color[idx] = chosen_color
        uncolored.remove(idx)

        for j in range(self.numVertices):
            if self.adjMatrix[idx][j] > 0 and color[j] == -1:
                saturation[j] += 1
    return color
```

Figure 27: Code DSATUR Algorithm

Since the code passed all the coverage tests, we can acknowledge that using these cases, we ensured that the code covered all the statements in the algorithm. However we must note that statement coverage does not always imply neither path nor decision coverage.

Decision Coverage:

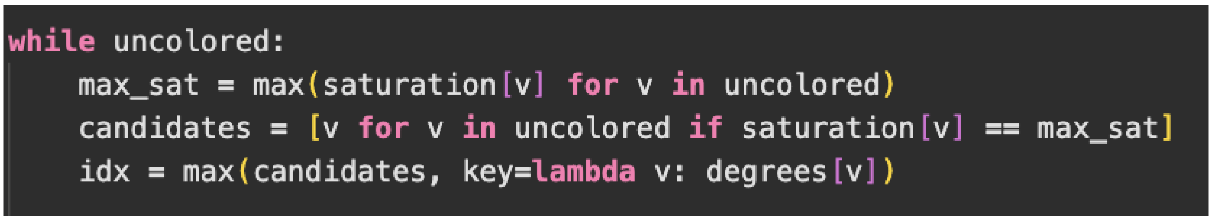
Decision coverage ensures that the decision points in the code are tested for test cases. For our algorithm, we need to first identify the decision points.



```
if self.numVertices == 0:
    return []
```

Figure 28: Outer if statement

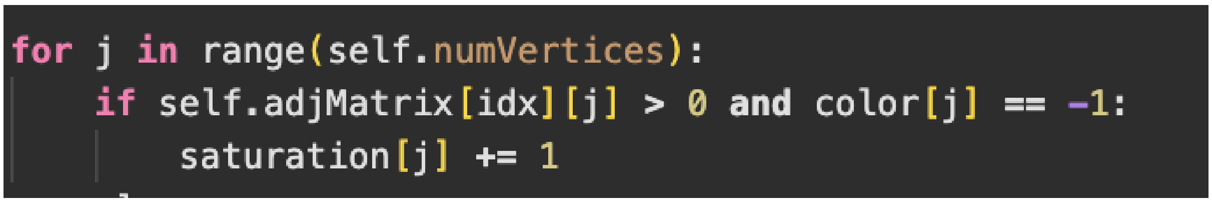
This if statement returns true when the graph has no vertices.



```
while uncolored:
    max_sat = max(saturation[v] for v in uncolored)
    candidates = [v for v in uncolored if saturation[v] == max_sat]
    idx = max(candidates, key=lambda v: degrees[v])
```

Figure 29: Inner while loop

This while loop returns true when there are still uncolored vertices.

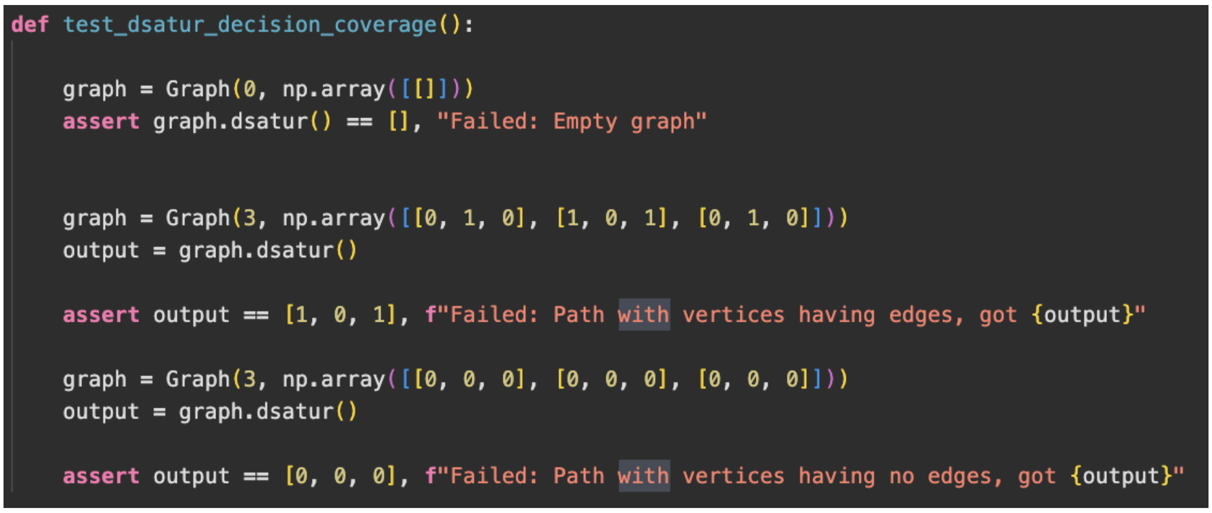


```
for j in range(self.numVertices):
    if self.adjMatrix[idx][j] > 0 and color[j] == -1:
        saturation[j] += 1
```

Figure 30: Inner if statement

This if statement returns true when there is an edge between the current vertex and vertex j and vertex j is uncolored.

To check each case for the decision coverage, we implemented these test cases:



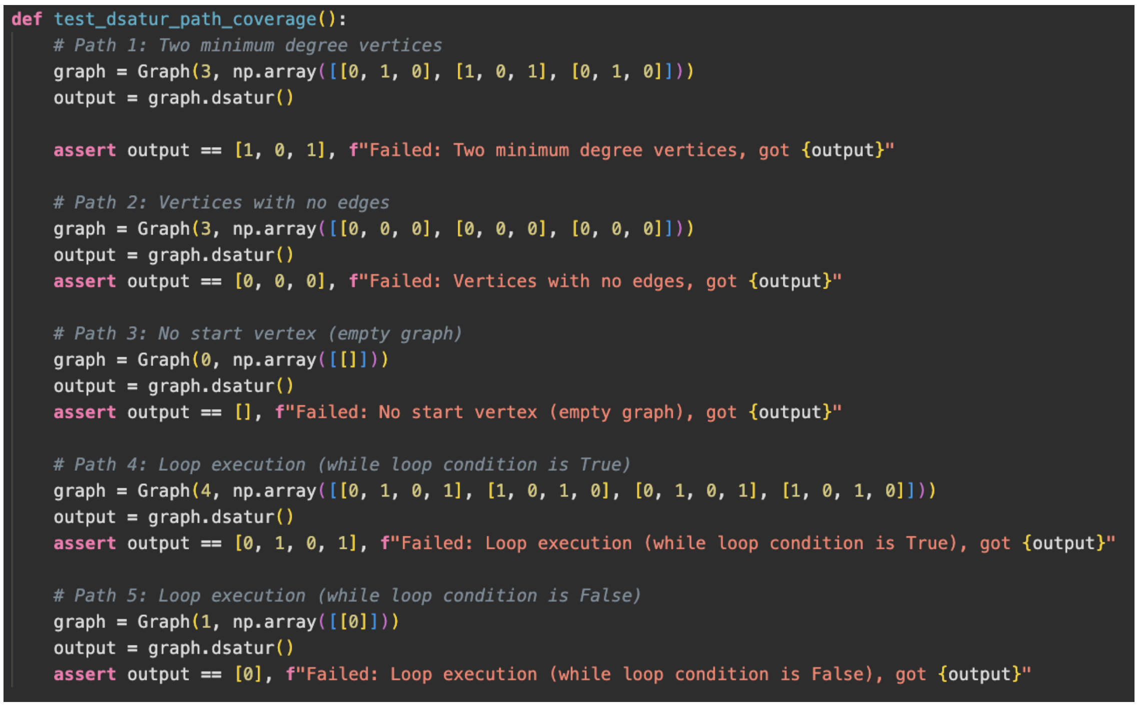
```python
def test_dsatur_decision_coverage():

    graph = Graph(0, np.array([[]]))
    assert graph.dsatur() == [], "Failed: Empty graph"


    graph = Graph(3, np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]]))
    output = graph.dsatur()

    assert output == [1, 0, 1], f"Failed: Path with vertices having edges, got {output}"

    graph = Graph(3, np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]]))
    output = graph.dsatur()

    assert output == [0, 0, 0], f"Failed: Path with vertices having no edges, got {output}"
```

Figure 31: Decision coverage cases

Since our code passed from each test case, we can say that we cover each decision node in our code.

Path Coverage:

Path coverage is used to test all execution paths through an algorithm. Since it may be impractical for large algorithms to be used for, we only focused on the key paths representing different scenarios.



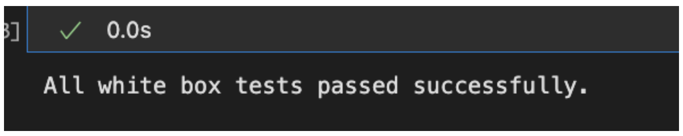
```python
def test_dsatur_path_coverage():
    # Path 1: Two minimum degree vertices
    graph = Graph(3, np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]]))
    output = graph.dsatur()

    assert output == [1, 0, 1], f"Failed: Two minimum degree vertices, got {output}"

    # Path 2: Vertices with no edges
    graph = Graph(3, np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]]))
    output = graph.dsatur()
    assert output == [0, 0, 0], f"Failed: Vertices with no edges, got {output}"

    # Path 3: No start vertex (empty graph)
    graph = Graph(0, np.array([[]]))
    output = graph.dsatur()
    assert output == [], f"Failed: No start vertex (empty graph), got {output}"

    # Path 4: Loop execution (while loop condition is True)
    graph = Graph(4, np.array([[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 1, 0]]))
    output = graph.dsatur()
    assert output == [0, 1, 0, 1], f"Failed: Loop execution (while loop condition is True), got {output}"

    # Path 5: Loop execution (while loop condition is False)
    graph = Graph(1, np.array([[0]]))
    output = graph.dsatur()
    assert output == [0], f"Failed: Loop execution (while loop condition is False), got {output}"
```

Figure 32: Path Coverage cases

As can be seen from the code, we have selected 5 different paths for testing purposes.

Path 1. Empty Graph
Path 2. Vertices with no Edges
Path 3. No Start Vertex
Path 4. Loop Execution while condition is true
Path 5. Loop execution while condition is true

However, we must also specify that these are not the only paths that our algorithm will go, we just wanted to check if there may be errors for us to fix.



```
3]  ✓  0.0s
All white box tests passed successfully.
```

Figure 33: Output for the white box tests

As we can see from the output, our algorithm passed from all tests.

9. Discussion

At the end of our article, we would like to lastly mention the results obtained from each part, and the parts where we think would greatly improve the future works of the problem of graph colouring. If we were to start our discussion section, we must first acknowledge that the brute force solution for the graph colouring problem always gives the minimum number of chromatic numbers possible for the colouring of a graph. But because of its nature, it does this on a way that produces exponential time complexity. This tendency makes it very hard to analyze and see the result of big test cases, (in our case, more than 15 vertices makes it run for at least half an hour) thus the theoretical time complexity of the algorithm, like we have stated at the part 3.2, is O(m^v).

Since we also proved it to be an NP-Hard problem, we also know that there isn't any optimal and correct solution other than brute force. At this point, since partially solving a problem is better than not solving it at all, only option that we had at our hand is to implement and use a heuristic greedy algorithm named DSatur, which uses saturation number to determine the next vertex that it will colour and with which colour. Although heuristic algorithm DSatur may look correct up until this point, due to its greedy nature, at higher input numbers (amount of vertices), the correctness level of the algorithm decreases as we have also showed it at the 7th part. The ratio between heuristic and brute force tend to increase, meaning that it starts to fail at correctly estimate the amount of colours to use. It also highly depends on the shape of the graph, not only on the vertex count. As for the time complexity, our tests with more than 7 different vertex counts, with 200 iterations and 10 runs, results a O(n^1.42), compared to the theoretical time complexity of O(n^2). On the examination part, we tested our heuristic algorithm both with blackbox and whitebox cases, for the hopes of finding corner-case bugs, but the results suggest that it successfully passed all cases. As with the testing, and the actual measurement of our algorithm suggest, actual time complexity is less than the theoretical one, as we know it is a case that occurs very rarely.

For the future works of heuristic algorithms of graph colouring, we would like to point out to the importance of input graphs, as the tendency to give errors increases when the randomness of the graph increases compared to the sample inputs specifically designed for algorithm to solve. Overall, we believe that the findings and the process of our algorithm of DSatur provides both information and insights for the ones that seek to understand and solve the controversial problem of graph colouring.

References

Brelaz, Daniel. “New Method to Color the Vertices of a Graph.” *Communications of the ACM* 22, no. 4 (1977): 251–56.

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