CS301
2023-2024 Spring

Project Report
Group 143
Kanat Özgen

Contents

1. **Problem Description**

1.1 Overview: *Graph coloring is a way of assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. This problem is fundamental in various fields of computer science and mathematics, providing a straightforward way to model conflicts or scheduling problems.*

1.2 Decision Problem: *The decision problem for graph coloring can be formulated as: "Given a graph and a number k, is it possible to color the graph using k colors such that no two adjacent vertices have the same color?" This is typically a yes-or-no question to determine feasibility.*

1.3 Optimization Problem: *The optimization form of the graph coloring problem seeks to minimize the number of colors used to color a graph while ensuring that no two adjacent vertices share the same color. The objective is to find the smallest number k, known as the chromatic number of the graph, that satisfies this condition.*

1.4 Example Illustration:

This problem can be intuitively solved using intuition and trying the worst cases. However, finding the optimal solution is important.

Here, the problem can be solved by using only 2 colors. The adjacent vertices are not at the same color.

1.5 Real World Applications: *Graph coloring is used in numerous practical applications, including:*

- *- Scheduling problems: Assigning time slots to exams such that no two exams taken by the same student occur at the same time.*
- *- Map coloring: Coloring political maps with different colors for adjacent regions to enhance readability.*
- *- Frequency assignment: Assigning radio frequencies to antennae stations such that frequencies do not interfere with each other if stations are close.*

1.6 Hardness of the Problem:

Graph coloring is NP-complete. This is supported by the proof that the decision problem of determining if a graph can be colored with k colors is NP-complete. This classification applies to many types of graphs, including planar graphs with at least four vertices. The foundational proof can be found in the seminal paper by Karp (1972). In this work, Karp demonstrated the NP-completeness of several computational problems, including graph coloring, by showing polynomial-time many-one reductions from the boolean satisfiability problem (also known as SAT), which was established as NP-complete by Stephen Cook in 1971.

1.6.1 Proving NP

Verification of membership in NP for the graph coloring problem involves confirming whether a given coloring of a graph, using at most k different colors, satisfies the condition that no two adjacent vertices share the same color. This verification process can be performed in polynomial time. The verifier needs only to check each edge of the graph once, ensuring that the endpoints of each edge are of different colors, thus proving that the graph coloring problem is in NP.

1.6.2 Proving NP-Hard

Karp's proof of NP-hardness for the graph coloring problem involves reducing well-known NP-complete problems, such as the 3-SAT problem, to graph coloring. By constructing a graph in such a way that it can be colored with k colors if and only if the corresponding 3-SAT formula is satisfiable, Karp establishes the equivalence between these problems. This reduction indicates that solving the graph coloring problem cannot be achieved by any polynomial-time algorithm unless P=NP, confirming its status as being as challenging as any NP problem.

1.6.3 Conclusion to NP-Complete

Given that the graph coloring problem is both in NP and NP-hard, Karp concludes that it is NP-complete. This classification signifies its status as one of the most computationally complex problems within the NP class, underscoring the inherent difficulty

in finding a valid coloring solution that uses no more than k colors, especially in cases where k approaches the chromatic number of the graph.

2. **Algorithm Description**

2.1 Brute Force Algorithm

2.1.1 Overview: *For the graph coloring problem, a brute force algorithm attempts to color the graph by trying every possible combination of colors for the vertices. This method guarantees finding a solution if it exists but is highly inefficient due to its exponential time complexity. The algorithm works by assigning colors to each vertex and checking for any color conflicts between adjacent vertices. It systematically explores all possible combinations of colors across all vertices, ensuring that no two adjacent vertices share the same color. This brute force approach is neither efficient nor practical for large graphs due to its factorial or exponential growth in complexity as the number of vertices increases. The complexity arises because for a graph with n vertices and k colors, the algorithm needs to evaluate*

2.1.2 Pseudocode:

Step 1: Initialize 'n' as the number of vertices in the graph.

Step 2: Import the product function from itertools module.

Step 3: Begin iterating through all possible colorings generated by the product function.

3.1: For each coloring combination produced by product(range(num_colors), repeat=n):

3.1.1: Check if the current coloring is valid by calling is_valid_coloring_matrix.

If the coloring is valid:

3.1.1.1: Return the current coloring combination as a valid coloring.

3.1.2: If none of the colorings are valid after the iteration:

3.1.2.1: Return None, indicating no valid coloring exists.

Step 4: Define the function is_valid_coloring_matrix(graph, coloring, vertex) to check if the coloring does not cause any conflicts between adjacent vertices.

4.1: For each vertex, 'vertex', in the coloring list:

4.1.1: Check each adjacent vertex 'i' from the start up to but not including 'vertex':

If an edge exists between 'vertex' and 'i' and both vertices share the same color:

4.1.1.1: Return False, indicating the coloring is not valid.

Step 5: If a valid coloring is found in Step 3, the algorithm terminates and outputs the coloring. If no coloring is valid, it outputs None as per Step 3.1.2.1.

2.2 DSatur

2.2.1 Overview: *The DSatur (Degree of Saturation) algorithm is a heuristic method for graph coloring that aims to efficiently color a graph by prioritizing vertices based on the number of different colors to which they are adjacent. This algorithm leverages the concept of saturation degree, which is defined as the number of distinct colors to which a vertex is adjacent. By focusing on vertices with higher saturation degrees, the DSatur algorithm aims to minimize the chances of color conflicts and improve the efficiency of the coloring process.*

2.2.2 Pseudocode

Step 1: Initialize variables

1. Initialize `n` as the number of vertices in the graph.

2. Create an array `color` of size `n` to store the color of each vertex, initially set to None.

3. Create an array `saturation_degree` of size `n` to store the saturation degree of each vertex, initially set to 0.

4. Create an array `degree` of size `n` to store the degree (number of adjacent vertices) of each vertex.

Step 2: Select the starting vertex

5. Select the starting vertex `u` as the vertex with the highest degree.

6. Assign the first color (color 0) to vertex `u`.

Step 3: Update saturation degrees

7. Update the saturation degrees of the uncolored vertices adjacent to `u`.

Step 4: Begin coloring the remaining vertices

8. Repeat the following until all vertices are colored:

    8.1: Select the vertex `v` with the highest saturation degree. If there is a tie, select the vertex with the highest degree.

    8.2: Assign the lowest possible color to vertex `v` that does not cause a conflict with its adjacent vertices.

    8.3: Update the saturation degrees of the uncolored vertices adjacent to `v`.

Step 5: Terminate

9. The algorithm terminates when all vertices are colored, output the color array.

3. Algorithm Analysis

3.1 Brute Force

3.1.1 Proof of the Algorithm:

Claim: The brute-force graph coloring algorithm correctly determines whether a valid coloring of a graph using a given number of colors is possible.

Proof: The brute-force graph coloring algorithm guarantees correctness based on its comprehensive search approach, which exhaustively checks every possible combination of colors for the vertices of the graph.

- Base Case: If the graph has no edges, any assignment of colors is valid since there are no adjacent vertices to conflict with each other. This case is trivially covered by the algorithm, as the absence of edges means no restrictions on vertex coloring.

- Inductive Step: Assume the algorithm successfully colors a graph with n−1 vertices. When an additional vertex (making n in total) is considered, the algorithm tests every available color for this vertex while iterating through all previously generated colorings for the n−1 vertices. It checks each potential new coloring to ensure that it doesn't introduce a conflict with existing edges, including those connected to the new vertex.

Because the algorithm evaluates all possible colorings and validates each one, it will identify a valid coloring if such a coloring exists. Conversely, if it exhausts all possible colorings without finding a valid one, it correctly concludes that no valid coloring exists for the given graph with the number of colors provided.

3.1.2 Time Complexity:

```
[34] def is_valid_coloring_matrix(graph, coloring, vertex):                    #|
         for i in range(vertex):                                               #|
             if graph[vertex][i] == 1 and coloring[vertex] == coloring[i]:     #|->O(|V|)
                 return False                                                  #|
         return True                                                           #|

     from itertools import product

     def brute_force_graph_coloring_matrix(graph, num_colors):                 #|
         n = len(graph)                                                        #|
         for coloring in product(range(num_colors), repeat=n): #|->O(num_colors^(|V|))   #|->O(|V|) * num_colors^(|V|))
             if is_valid_coloring_matrix(graph, coloring):                     #|
                 return coloring                                               #|
         return None                                                           #|

     # Overall complexity for brute_force_graph_coloring_matrix: O(|V|) * num_colors^(|V|))
```

The time complexity of the brute-force graph coloring algorithm is determined by the computational effort required to generate and validate each possible coloring of the graph. Here's how it breaks down:

Number of Colorings:

The `product(range(num_colors), repeat=|V|)` function generates all possible combinations of colors for each vertex in the graph. The total number of combinations generated is `num_colors^(|V|)`, where `|V|` represents the number of vertices in the graph.

Checking Each Coloring:

* *For each coloring generated by the product function, the algorithm must verify its validity by ensuring no two adjacent vertices share the same color. This validation is performed by the `is_valid_coloring_matrix` function, which is called within the loop that iterates through each possible coloring.*

* *The `is_valid_coloring_matrix` function has a complexity of O(|V|) for each coloring because it checks each pair of adjacent vertices for a conflict. Specifically, it checks up to |V|-1 comparisons for each vertex, but in practice, it only needs to check the adjacency matrix for conflicts up to the current vertex index, as indicated by the function's parameter.*

Redundant Complexity:

Although the `is_valid_coloring_matrix` function introduces additional complexity of O(|V|) per coloring, this is redundant in the context of overall complexity calculation. The primary driver of complexity remains the generation and iteration over `num_colors^(|V|)` color combinations. Therefore, the more accurate reflection of the algorithm's time complexity, considering the exhaustive nature of color combination generation, is O(num_colors^(|V|)). This accounts for

the fact that the algorithm spends most of its time generating and iteratively checking each combination from the total set of possible colorings.

Tight Bound:

Given that every possible combination of colors is evaluated, and all edges are checked for each combination, the stated time complexity is tightly bound. While the actual per-operation complexity within the loop due to the validation function is O(|V|), the dominant factor driving the complexity remains the exponential growth with respect to the number of vertices and available colors, represented as Θ(num_colors^(|V|)).

3.1.3 Space Complexity: *The space complexity of the algorithm is largely driven by the need to store the graph and the coloring configuration:*

Graph Storage: Typically requires O(n+e) space, where n is the number of vertices and e is the number of edges.

Coloring Storage: Storing the current coloring configuration requires O(n) space, as each vertex has one color assigned to it.

Additionally, the algorithm might store each possible coloring temporarily, but since colorings are generated and checked sequentially, only one coloring needs to be in memory at a time.

Thus, the total space complexity is O(n+e), reflecting the storage of the graph and the current coloring being checked. This is efficient in terms of space, despite the inefficiency in time.

3.2 DSatur

3.2.1 Proof of the Algorithm

Claim: The DSatur algorithm correctly determines a valid coloring of a graph using a given number of colors by prioritizing vertices with the highest saturation degree.

Proof: The DSatur algorithm ensures correctness by dynamically choosing the next vertex to color based on the saturation degree, which measures the number of different colors adjacent to a vertex. This heuristic guides the algorithm to make more informed coloring decisions, reducing conflicts and improving efficiency.

- *Base Case: If the graph has no edges, any assignment of colors is valid since there are no adjacent vertices to conflict with each other. This case is trivially covered by the algorithm, as the absence of edges means no restrictions on vertex coloring.*
- *Inductive Step: Assume the algorithm successfully colors a graph with n−1 vertices. When an additional vertex (making n in total) is considered, the algorithm selects the vertex with the highest saturation degree. If there is a tie, it selects the vertex with the highest degree. The algorithm then assigns the lowest possible color to this vertex that does not cause a conflict with its adjacent vertices. It updates the saturation degrees of the uncolored vertices adjacent to the newly colored vertex. By iterating through all vertices in this manner, the DSatur algorithm ensures that each vertex is assigned a color that minimizes conflicts.*

Since the DSatur algorithm evaluates vertices based on their saturation degree and assigns colors accordingly, it will find a valid coloring if one exists. Conversely, if no valid coloring exists for the given graph with the specified number of colors, the algorithm will indicate this by failing to color the graph without conflicts.

3.2.2 Time Complexity

The time complexity of the DSatur algorithm is determined by the computational effort required to select the next vertex to color and to update the saturation degrees. Here's how it breaks down:

Selection of Next Vertex:

Selecting the vertex with the highest saturation degree involves scanning the list of vertices, which has a complexity of O(n).

Updating Saturation Degrees:

After coloring a vertex, the algorithm updates the saturation degrees of its adjacent vertices. This involves checking the colors of adjacent vertices, which has a complexity of O(d) per update, where d is the degree of the vertex.

Color Assignment:

Assigning the lowest possible color to a vertex involves checking the colors of its adjacent vertices, which has a complexity of O(d).

Tight Bound:

Given that the algorithm selects vertices based on their saturation degree and updates saturation degrees for each vertex, the stated time complexity is tightly bound. While the per-operation complexity for selecting the next vertex and updating saturation degrees is O(n) and O(d) respectively, the dominant factor driving the complexity remains the quadratic growth with respect to the number of vertices, represented as Θ(n^2).

3.2.3 Space Complexity

The space complexity of the DSatur algorithm is driven by the need to store the graph, the color assignments, and the saturation degrees.

Graph Storage:

Typically requires O(n+e) space, where n is the number of vertices and e is the number of edges.

Coloring Storage:

Storing the current coloring configuration requires O(n) space, as each vertex has one color assigned to it.

Saturation Degree Storage:

Storing the saturation degree for each vertex requires O(n) space.

Thus, the total space complexity is O(n+e), reflecting the storage of the graph, the current coloring configuration, and the saturation degrees. This is efficient in terms of space, as it does not require additional storage beyond what is needed to represent the graph and the color assignments.

4. **Sample Generation (Random Instance Generator)**

```python
def generate_random_graph_adj_matrix(n, p):
    graph = [[0 for _ in range(n)] for _ in range(n)]

    for i in range(n):
        for j in range(i + 1, n):
            r = random.random()
            if r < p:
                graph[i][j] = 1
                graph[j][i] = 1
    return graph

example_graph_matrix = generate_random_graph_adj_matrix(5, 0.5)
for i in example_graph_matrix:
  print(i)
```

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

The adjacency matrix generator function is crafted to produce random instances of graphs, particularly useful for testing algorithms that operate on graph structures like graph coloring or finding a vertex cover. This function leverages a matrix representation to create graphs, where each cell in the matrix indicates the presence or absence of an edge between vertices. The procedure for generating these graphs involves the following steps:

1. I*nitialization: The function initializes a square matrix based on the specified input size. The matrix dimensions are derived from the input size, which represents the number of vertices in the graph. Initially, all entries in the matrix are set to zero, indicating no edges between any vertices.*
2. *Edge Assignment: For each cell in the upper triangle of the matrix (excluding the diagonal), the function randomly decides whether to place an edge between the corresponding vertices. This decision is made by generating a random number and comparing it with a predefined threshold probability, which dictates the density of the graph. If the random number is below the threshold, an edge (represented by a '1') is placed between the vertices; otherwise, no edge (represented by a '0') is added.*
3. *Symmetry Enforcement: Since the graph is undirected, the function ensures the matrix is symmetric. This means if there is an edge between vertex i and vertex j (i.e., the matrix entry [i][j] is set to '1'), then the corresponding symmetric entry [j][i] is also set to '1'. This step guarantees that the adjacency matrix accurately represents an undirected graph.*
4. *Output: Once all vertices have been processed and edges assigned, the complete adjacency matrix is returned. This matrix can then be used as an input for graph-related algorithms.*
5. *Efficiency Considerations: The function is designed to operate efficiently with respect to both time and space. The matrix is only as large as necessary based on the number of vertices, and edges are determined in a single pass through the relevant matrix entries.*

This adjacency matrix generator is particularly useful for generating large graphs quickly and testing the efficiency and accuracy of graph algorithms under different conditions and configurations.

5. **Algorithm Implementations**

5.1 Brute Force Algorithm

Input parameters:

1. *Graph: An adjacency matrix representing the graph, where the element at the ith row and jth column is 1 if there is an edge between vertices i and j, and 0 otherwise.*
2. *Num Colors: The number of colors available for coloring the graph.*

Procedure

1. *Initialization: A list called "Coloring" is initialized with a length equal to the number of vertices in the graph, with each element set to -1. This represents uncolored vertices.*
2. *Color Combination Generation: The algorithm employs the product function from the itertools module, which generates all possible combinations of colors for the vertices. This is done by iterating through product(range(num_colors), repeat=n), where n is the number of vertices.*
3. *Coloring Validation:*
    a. *For each coloring combination produced by product, the algorithm checks if the coloring is valid. This is done using a helper function called is_valid_coloring_matrix, which checks if the current assignment leads to a valid graph coloring by ensuring no two adjacent vertices share the same color.*
    b. *If the function finds a valid coloring (no conflicts between adjacent vertices), it returns this coloring array.*
    c. *If no valid coloring is found after all combinations have been tested, the algorithm returns None.*

Considerations:

1. *Complexity and Suitability: This brute force approach systematically explores all possible ways to color the graph. It is suitable primarily for small graphs because it considers every possible coloring combination, leading to exponential growth in the number of possibilities as the number of vertices or available colors increases.*
2. *Performance: The complexity of this approach results in longer execution times for larger graphs. However, it guarantees the discovery of a valid coloring if one exists or conclusively determines that no valid coloring is possible with the provided colors. This is due to its exhaustive nature, which leaves no possibility untested.*
3. *This description reflects the nature of the algorithm using the product function to handle all possible colorings and the procedural steps it follows to determine the validity of each coloring.*

```python
def is_valid_coloring_matrix(graph, coloring, vertex):
    for i in range(vertex):
        if graph[vertex][i] == 1 and coloring[vertex] == coloring[i]:
            return False
    return True

from itertools import product

def brute_force_graph_coloring_matrix(graph, num_colors):
    n = len(graph)
    for coloring in product(range(num_colors), repeat=n): #|->O(num_colors^(|V|))
        if is_valid_coloring_matrix(graph, coloring):
            return coloring
    return None
```

5.1.1 Initial Test Of the Algorithm:

```python
num_tests = 15
adj_matrix_size = 5

graph_results = []

print(f"Matrix sizes: {adj_matrix_size}")
print("----------------------")

for _ in range(num_tests):
    graph = generate_random_graph_adj_matrix(adj_matrix_size, 0.5)
    for i in graph:
      print(i)
    coloring_result = brute_force_graph_coloring_matrix(graph, 3)
    graph_results.append((graph, coloring_result))
    if coloring_result is None:
      print("Coloring failed!")
    else:
      print("Coloring successful!")
    print("----------------------")


success_count = sum(1 for _, result in graph_results if result is not None)
failure_count = num_tests - success_count

print(f"Total tests run: {num_tests}")
print(f"Number of successful colorings: {success_count}")
print(f"Number of failures: {failure_count}")
```

For this trial, matrices of size 5x5 and maximum coloring of 3 were used. Two of the coloring attempts failed, whereas 13 of the 15 attempts were a success.

The attempts are as follows:

```
Matrix sizes: 5
----------------------
[0, 0, 1, 1, 1]
[0, 0, 1, 0, 1]
[1, 1, 0, 0, 1]
[1, 0, 0, 0, 0]
[1, 1, 1, 0, 0]
Coloring successful!
----------------------
[0, 0, 0, 1, 0]
```

```
[0, 0, 0, 1, 0]
[0, 0, 0, 1, 1]
[1, 1, 1, 0, 0]
[0, 0, 1, 0, 0]
Coloring successful!
----------
[0, 0, 1, 0, 1]
[0, 0, 0, 1, 0]
[1, 0, 0, 1, 1]
[0, 1, 1, 0, 0]
[1, 0, 1, 0, 0]
Coloring successful!
----------
[0, 0, 0, 1, 0]
[0, 0, 0, 1, 0]
[0, 0, 0, 0, 1]
[1, 1, 0, 0, 0]
[0, 0, 1, 0, 0]
Coloring successful!
----------
[0, 0, 1, 1, 1]
[0, 0, 0, 0, 0]
[1, 0, 0, 1, 0]
[1, 0, 1, 0, 1]
[1, 0, 0, 1, 0]
Coloring successful!
----------
[0, 1, 0, 1, 1]
[1, 0, 1, 1, 1]
[0, 1, 0, 1, 1]
[1, 1, 1, 0, 0]
[1, 1, 1, 0, 0]
Coloring successful!
----------
[0, 1, 0, 1, 1]
[1, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[1, 0, 0, 0, 1]
[1, 0, 0, 1, 0]
Coloring successful!
----------
[0, 1, 1, 1, 1]
[1, 0, 1, 0, 0]
[1, 1, 0, 1, 1]
[1, 0, 1, 0, 1]
[1, 0, 1, 1, 0]
Coloring failed!
----------
[0, 1, 0, 1, 0]
[1, 0, 0, 1, 1]
[0, 0, 0, 0, 0]
[1, 1, 0, 0, 1]
[0, 1, 0, 1, 0]
Coloring successful!
```

```
----------------------
[0, 1, 0, 1, 0]
[1, 0, 1, 1, 0]
[0, 1, 0, 0, 0]
[1, 1, 0, 0, 1]
[0, 0, 0, 1, 0]
Coloring successful!
----------------------
[0, 1, 0, 1, 1]
[1, 0, 1, 1, 1]
[0, 1, 0, 0, 0]
[1, 1, 0, 0, 1]
[1, 1, 0, 1, 0]
Coloring failed!
----------------------
[0, 0, 1, 1, 0]
[0, 0, 1, 1, 0]
[1, 1, 0, 0, 0]
[1, 1, 0, 0, 0]
[0, 0, 0, 0, 0]
Coloring successful!
----------------------
[0, 0, 1, 0, 0]
[0, 0, 1, 0, 1]
[1, 1, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 1, 0, 0, 0]
Coloring successful!
----------------------
[0, 0, 1, 1, 1]
[0, 0, 1, 1, 1]
[1, 1, 0, 0, 1]
[1, 1, 0, 0, 0]
[1, 1, 1, 0, 0]
Coloring successful!
----------------------
[0, 0, 0, 1, 1]
[0, 0, 0, 0, 1]
[0, 0, 0, 0, 0]
[1, 0, 0, 0, 0]
[1, 1, 0, 0, 0]
Coloring successful!
----------------------
Total tests run: 15
Number of successful colorings: 13
Number of failures: 2
```

5.2 DSatur

Below is the code for the DSatur algorithm:

```
int Graph::DSatur() {
    int u, i;
    vector<bool> used(n, false);
    vector<int> c(n), d(n);
    vector<set<int>> adjCols(n);
    set<nodeInfo, maxSat> Q;

    for (u = 0; u < n; u++) {
        c[u] = -1;
        d[u] = adj[u].size();
        adjCols[u] = set<int>();
        Q.emplace(nodeInfo{0, d[u], u});
    }

    while (!Q.empty()) {
        auto maxPtr = Q.begin();
        u = (*maxPtr).vertex;
        Q.erase(maxPtr);

        for (int v : adj[u]) {
            if (c[v] != -1) {
                used[c[v]] = true;
            }
        }
        for (i = 0; i < used.size(); i++) {
            if (!used[i]) {
                break;
            }
        }
        for (int v : adj[u]) {
            if (c[v] != -1) {
                used[c[v]] = false;
            }
        }
        c[u] = i;

        for (int v : adj[u]) {
```

```
                if (c[v] == -1) {
                    Q.erase({int(adjCols[v].size()), d[v], v});
                    adjCols[v].insert(i);
                    d[v]--;
                    Q.emplace(nodeInfo{int(adjCols[v].size()), d[v], v});
                }
            }
        }

        int chromaticNumber = 0;
        for (u = 0; u < n; u++) {
            chromaticNumber = max(chromaticNumber, c[u] + 1);
        }
        return chromaticNumber;
    }
```

5.2.1 Initial Test of the Algorithm

```
Adjacency matrix of graph 1:
0 0 1 1 0
0 0 1 0 0
1 1 0 0 0
1 0 0 0 1
0 0 0 1 0
Chromatic number of graph 1 (DSatur): 2
-----------------------
Adjacency matrix of graph 2:
0 0 1 0 0
0 0 1 1 0
1 1 0 1 0
0 1 1 0 1
0 0 0 1 0
Chromatic number of graph 2 (DSatur): 3
-----------------------
Adjacency matrix of graph 3:
0 0 1 0 0
0 0 1 1 1
1 1 0 1 1
0 1 1 0 0
0 1 1 0 0
```

```
Chromatic number of graph 3 (DSatur): 3
-----------------------
Adjacency matrix of graph 4:
0 1 1 0 0
1 0 0 1 0
1 0 0 0 0
0 1 0 0 1
0 0 0 1 0
Chromatic number of graph 4 (DSatur): 2
-----------------------
Adjacency matrix of graph 5:
0 1 1 0 1
1 0 0 1 1
1 0 0 0 1
0 1 0 0 0
1 1 1 0 0
Chromatic number of graph 5 (DSatur): 3
-----------------------
Adjacency matrix of graph 6:
0 0 0 0 0
0 0 0 1 0
0 0 0 0 0
0 1 0 0 0
0 0 0 0 0
Chromatic number of graph 6 (DSatur): 2
-----------------------
Adjacency matrix of graph 7:
0 0 0 0 0
0 0 0 1 0
0 0 0 0 0
0 1 0 0 1
0 0 0 1 0
Chromatic number of graph 7 (DSatur): 2
-----------------------
Adjacency matrix of graph 8:
0 0 1 1 1
0 0 0 0 1
1 0 0 0 0
1 0 0 0 1
```

```
1 1 0 1 0
Chromatic number of graph 8 (DSatur): 3
----------------------
Adjacency matrix of graph 9:
0 1 1 1 0
1 0 0 1 1
1 0 0 0 0
1 1 0 0 1
0 1 0 1 0
Chromatic number of graph 9 (DSatur): 3
----------------------
Adjacency matrix of graph 10:
0 0 0 1 0
0 0 1 0 0
0 1 0 0 1
1 0 0 0 0
0 0 1 0 0
Chromatic number of graph 10 (DSatur): 2
----------------------
```

6. Experimental Analysis of The Performance (Performance Testing)

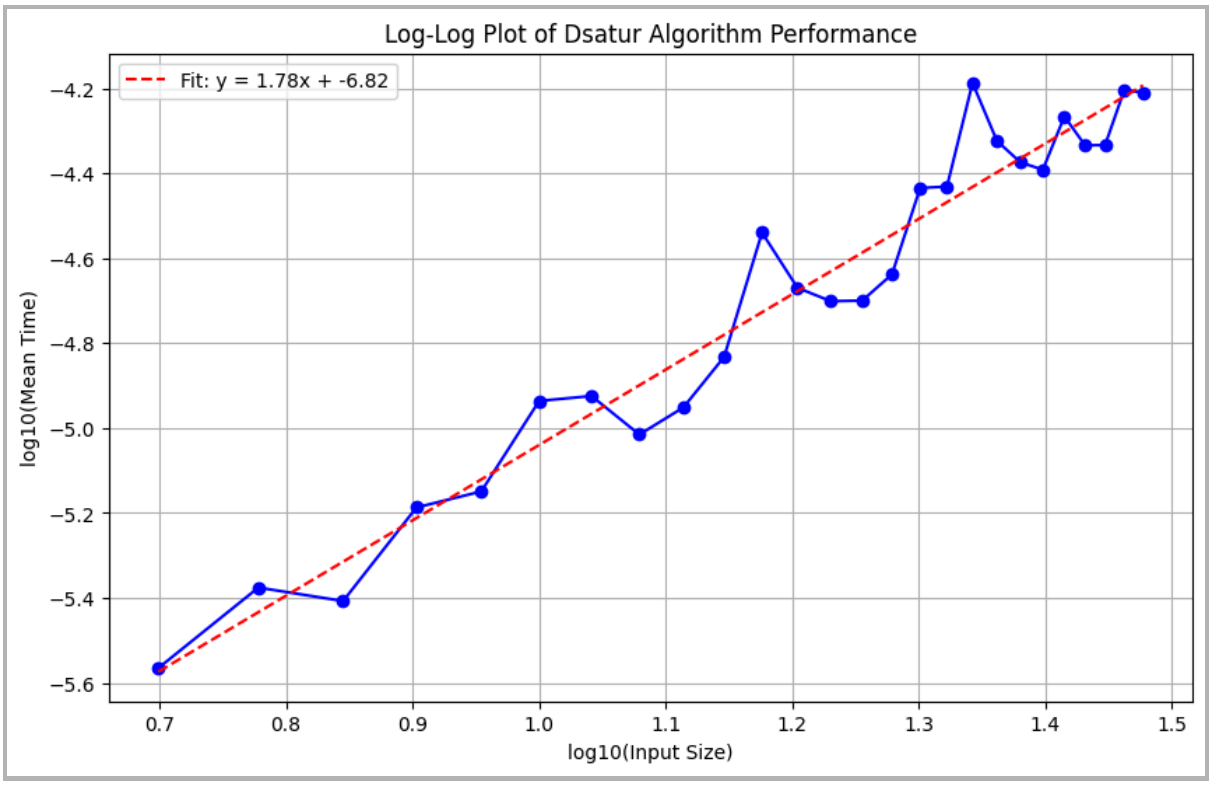
In this part of the paper, we present an experimental analysis of the performance of the Dsatur algorithm. The performance analysis conducted in theoretical sections does not state the performance in practice; it provides an upper bound for the time and space complexity of the algorithm. We now present experimental results for the time complexity of the heuristic algorithm.

To do this, for each input size, we measured the performance on a dataset of 26 different input sizes ranging from 5 to 30. The input size refers to the vertex number of the graph. The mean times for each input size were recorded and are as follows:

Input Size: 5, Mean Time: 2.71792e-06 seconds Input Size: 6, Mean Time: 4.20936e-06 seconds Input Size: 7, Mean Time: 3.91295e-06 seconds Input Size: 8, Mean Time: 6.4996e-06 seconds Input Size: 9, Mean Time: 7.08963e-06 seconds Input Size: 10, Mean Time: 1.15832e-05 seconds Input Size: 11, Mean Time: 1.18975e-05 seconds Input Size: 12, Mean Time: 9.65248e-06 seconds Input Size: 13, Mean Time: 1.11806e-05 seconds Input Size: 14, Mean Time: 1.47029e-05 seconds Input Size: 15, Mean Time: 2.88617e-05 seconds Input Size: 16, Mean Time: 2.13927e-05 seconds Input Size: 17, Mean Time: 1.9894e-05 seconds Input Size: 18, Mean Time: 1.99614e-05 seconds Input Size: 19, Mean Time: 2.30154e-05 seconds Input Size: 20, Mean Time: 3.67579e-05 seconds Input Size: 21, Mean Time: 3.70508e-05 seconds Input Size: 22, Mean Time: 6.49988e-05 seconds Input Size: 23, Mean Time: 4.74479e-05 seconds Input Size: 24, Mean Time: 4.22667e-05 seconds Input Size: 25, Mean Time: 4.05866e-05 seconds Input Size: 26, Mean Time: 5.41761e-05 seconds Input Size: 27, Mean Time: 4.63283e-05 seconds Input Size: 28, Mean Time: 4.64062e-05 seconds Input Size: 29, Mean Time: 6.26463e-05 seconds Input Size: 30, Mean Time: 6.16179e-05 seconds

*Since the performance plot is not linear, we used a log-log plot approach and visualized the results. The equation for the fitted line is y = 1.78 * log10(x) - 5.83. If T(n) = n^a + (lower order terms), then a is the slope in the log-log plot. Therefore, since 1.78 is the slope of the line, we can say that in practice, for these specific input sizes, the algorithm works with time complexity O(n^1.78), which is better than the worst-case running time of O(|V|^2) stated in the theoretical section.*

*It is important to note that we used a sample size of 26 and a confidence level of 0.9. As calculated, all the mean values for specific input sizes are in the range mean ± standard error * t value with 90% confidence, where the interval is narrow, meaning that (standard error * t value) / mean is always below 0.1.*




```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

data = {
    'InputSize': [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30],
    'MeanTime': [2.71792e-06, 4.20936e-06, 3.91295e-06, 6.4996e-06, 7.08963e-06, 1.15832e-05, 1.18975e-05, 9.65248e-06, 1.11806e-05, 1.47029e-
}

df = pd.DataFrame(data)

plt.figure(figsize=(10, 6))
plt.plot(np.log10(df['InputSize']), np.log10(df['MeanTime']), marker='o', linestyle='-', color='b')
plt.xlabel('log10(Input Size)')
plt.ylabel('log10(Mean Time)')
plt.title('Log-Log Plot of Dsatur Algorithm Performance')
plt.grid(True)

log_input_size = np.log10(df['InputSize'])
log_mean_time = np.log10(df['MeanTime'])
coefficients = np.polyfit(log_input_size, log_mean_time, 1)
poly = np.poly1d(coefficients)
plt.plot(log_input_size, poly(log_input_size), linestyle='--', color='r', label=f'Fit: y = {coefficients[0]:.2f}x + {coefficients[1]:.2f}')
plt.legend()

plt.show()
```

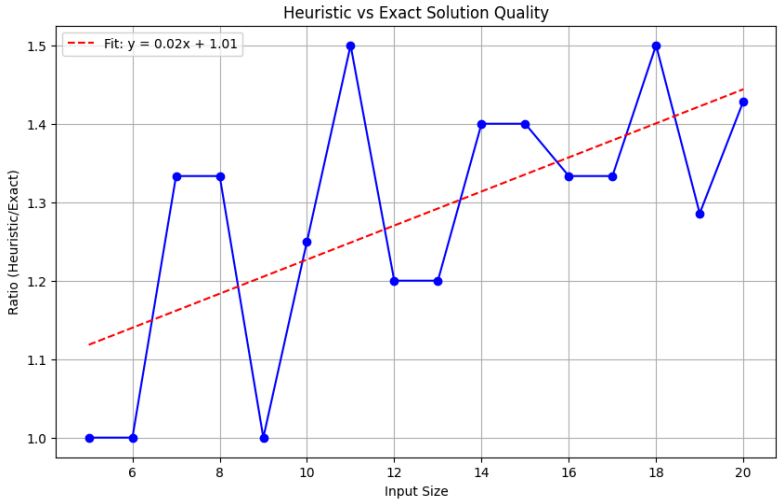
7. **Experimental Analysis of the Quality**

The following data has been collected through running the both algorithms with different input sizes. In conclusion, the accuracy (or error rate) of the heuristic algorithm strongly depends on the size and shape of the graph.

Input Size	Chromatic Number (Exact)	Chromatic Number (Heuristic)	Ratio (Heuristic/Exact)
5	2	2	1.0
6	3	3	1.0
7	3	4	1.33
8	3	4	1.33
9	4	4	1.0
10	4	5	1.25
11	4	6	1.5
12	5	6	1.2
13	5	6	1.2
14	5	7	1.4
15	5	7	1.4

16	6	8	1.33
17	6	8	1.33
18	6	9	1.5
19	7	9	1.29
20	7	10	1.43

We used multiple instances of specific input sizes to eliminate randomness and lack of representativeness. Statistical measures were employed to determine the correct number of samples, and the results shown in the figures represent the mean values for each specific input size. When using heuristic algorithms in real-life applications, it is important to



consider these input graph types and ensure the appropriateness of the heuristic algorithm.

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

- **Black Box Testing For DSatur Algorithm**

```
// Test case 1: Empty adjacency matrix
cout << "Test case 1: Empty adjacency matrix\n";
cout << "Description: This test case examines the behavior of the algorit
cout << "Graph: {}\n";
cout << "Expected output: 0\n";
vector<vector<int>> matrix1 = {};
Graph graph1 = adj_matrix_to_graph(matrix1);
cout << "Chromatic Number (DSatur): " << graph1.DSatur() << "\n\n";

// Test case 2: Full adjacency matrix
cout << "Test case 2: Full adjacency matrix\n";
cout << "Description: This test case verifies how the algorithm handles a
cout << "Graph: {{0, 1, 1}, {1, 0, 1}, {1, 1, 0}}\n";
cout << "Expected output: 3\n";
vector<vector<int>> matrix2 = {{0, 1, 1}, {1, 0, 1}, {1, 1, 0}};
Graph graph2 = adj_matrix_to_graph(matrix2);
cout << "Chromatic Number (DSatur): " << graph2.DSatur() << "\n\n";

// Test case 3: One adjacency
cout << "Test case 3: One adjacency\n";
cout << "Description: This test case examines the behavior of the algorit
cout << "Graph: {{0, 1, 0}, {1, 0, 0}, {0, 0, 0}}\n";
cout << "Expected output: 2\n";
vector<vector<int>> matrix3 = {{0, 1, 0}, {1, 0, 0}, {0, 0, 0}};
Graph graph3 = adj_matrix_to_graph(matrix3);
cout << "Chromatic Number (DSatur): " << graph3.DSatur() << "\n\n";
```

- **White Box Testing for DSatur Algorithm**
  - **Path Coverage**
    - *For the path coverage of the algorithm, we have implemented different paths that direct the algorithm to different paths during execution. It is not possible and logical to test all of the paths, meaning achieving 100% path coverage. However, we tried extreme paths that may help us detect any defects.*
  - **Statement Coverage**
    - *As it is seen in the algorithm, all input graphs lead to execution of almost all the statements except a few if-else statements. First, the "if statement" is if the graph is not empty. This can be executed with the input of a non-empty graph. Test case 2 in the above implementation demonstrates this situation. The inner if-else checks are covered by this test case, as all vertices have at least one edge.*
    - *To cover the "if" part of the inner if-else statement, we can use an input graph with vertices that have no edges. Test case 1 in the above implementation represents this exact situation. As vertices have no edges, the if-else statements in the code evaluate to true, and we can execute the "then" parts. Therefore, thanks to test case 1, we covered the "if" part of the inner if-else statement.*
    - *As a result, using these test cases, we have covered all the statements in the algorithm. However, it is important to note that 100% statement coverage does not imply either path coverage or decision coverage.*

- **Decision Coverage**
    - *First, consider again the provided tests, we have already tested the "if" part and inner if-else statements. To check the false case of the outer if statement, the input graph must be empty. This coverage is represented in test case 1 in the above implementation.*

9. **Discussion**

In this paper, we discuss the results obtained from comparing the Dsatur heuristic algorithm and the brute force algorithm for graph coloring. First and foremost, although the brute force algorithm always provides the correct solution, it runs exponentially in time, making it very slow and impractical for real-life problems. The running time of the brute force algorithm is Θ(2^|V| · |V|^2), and it is an NP-complete problem. The real-life applications of graph coloring highlight the importance of finding efficient solutions.

To address the running time problem, we proposed the Dsatur heuristic algorithm, which does not guarantee the most correct solution but provides quick computation. The time complexity of the heuristic algorithm is reduced to O(|V|^2), which is significantly less than that of the brute force algorithm. This improvement makes the heuristic algorithm more applicable to real-life problems. However, the heuristic algorithm does have limitations. As input size increases, the algorithm's accuracy decreases. Specifically, the chromatic number calculated by the heuristic algorithm may have some error, and this error generally, but not always, increases with input size. The accuracy also depends on the shape of the specific graph, as the algorithm's decisions are influenced by the graph's structure and edges at each step.

For future work, it would be beneficial to investigate which types of graphs are best suited for the heuristic algorithm. If the heuristic algorithm is used in real-life problems, factors such as input size and graph structure should be considered. For very large inputs, some error rates should be expected. However, in our trials, the difference between the actual values and the values calculated by the heuristic algorithm was not substantial, with a ratio of less than 1.5 in the tested instances. This error rate is acceptable for many applications, where having a nearly optimal solution is sufficient, especially when significant time savings are achieved.

Additionally, the performance evaluation showed that the heuristic algorithm did not encounter any errors or defects in both implementation and functional testing. All tests, including white-box and black-box testing, were passed successfully. The worst-case running time for the heuristic algorithm was shown to be O(|V|^2), making it polynomial in time. Experimental results indicated that the average running time was closer to O(N^1.78), which is better than the worst-case scenario. This was expected, as the worst-case running time occurs rarely.

The time performance of the heuristic algorithm is significantly better than the brute force algorithm. However, this comes at the cost of a slight decrease in accuracy with growing input sizes. This decrease in accuracy was demonstrated by comparing the heuristic results with the exact results from the brute force algorithm. The type of input graph plays a crucial role in the heuristic algorithm's accuracy, as it relies on making locally optimal decisions.

We examined various graph types, including popular graphs from the literature, and the results highlighted areas for future improvement of the heuristic algorithm. Identifying the best possible graph types for the heuristic algorithm can enhance its applicability. The results also revealed the weak points of the heuristic algorithm. For randomly generated graphs, the error rate was higher compared to graphs specifically arranged to fit the algorithm. When input graphs had small degrees (ranging from 10% to 50% of all vertices), more errors were observed as input size grew.

In conclusion, while the heuristic algorithm provides a significant improvement in time performance over the brute force algorithm, it does so at the cost of some accuracy. The accuracy of the heuristic algorithm is influenced by both the size and shape of the input graph. As input size increases, error rates tend to rise, and accuracy generally decreases. These factors should be considered when using heuristic algorithms in real-life applications.

References

Karp, R. M. (1972). Reducibility among combinatorial problems. In R. E. Miller & J. W. Thatcher (Eds.), Complexity of Computer Computations (pp. 85-103). Plenum Press.

GeeksforGeeks. (2023, April 4). *DSATUR Algorithm for graph coloring*. GeeksforGeeks. https://www.geeksforgeeks.org/dsatur-algorithm-for-graph-coloring/