CS301

2023-2024 Spring

Project Report

Group 137

İpek Uzun 30837

Mehmet Selman Yılmaz 31158

1. Problem Description

1.1 Overview:

Let G = (V, E) be an undirected graph, comprising a set of n vertices V and a set of m edges E. In the graph coloring problem, the objective is to assign each vertex v $\in$ V an integer c(v) $\in$ {1, 2, ..., k}, where k is the number of colors available, such that no two adjacent vertices v and u, denoted by {v, u} $\in$ E, are assigned the same color. In other words, for every edge {v, u} $\in$ E, it must hold that c(v) $\neq$ c(u).

1.2 Decision Problem:

Given an undirected graph G and an integer k, is there a valid vertex coloring of G using at most k colors such that no two adjacent vertices of the graph are colored with the same color?" is the formal structure for the decision issue.

1.3 Optimization Problem:

Given an undirected graph G=(V,E), where V represents the set of vertices and E represents the set of edges, the Graph Coloring Optimization Problem can be defined as finding a function c:V$\rightarrow${1,2,...,k}, where k is the minimum number of colors required, such that for every edge {u,v}$\in$E, it holds that c(u) $\neq$ c(v).

1.4 Example Illustration:

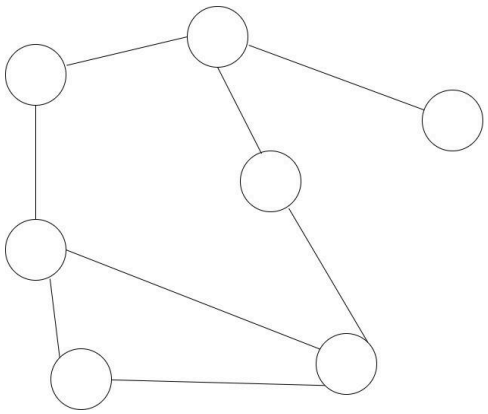

Figure 1. Illustration of an undirected graph

Suppose this is our graph with 7 nodes. We need to color this graph such that such that no two adjacent vertices share the same color.

Step 1. Let's choose the first vertex and color it with the first color.

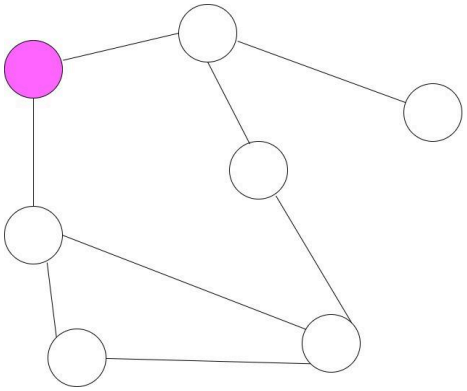

Figure 2. Colored graph

Step 2. Now we check the neighbors of the graph Since 2 and 3 cannot be the same color as 1. We should color them with different colors.

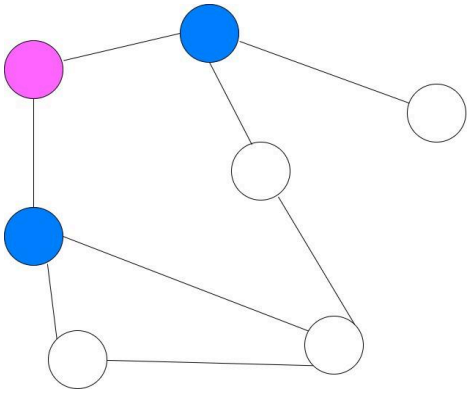

Figure 3. Colored Graph

Step 3. Now we check the neighbors of the graph 2 and 3 vertices' neighbors None of them connected with 1 so we can color the numbered 4,5,6 vertices with the 1's color which is pink.

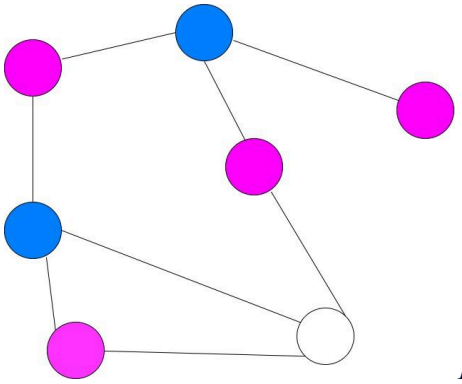

Figure 4. Colored graph

Step 4. Finally, for the last vertex, we see that it is both connected with pink and blue. Therefore we need to color it with a different color, suppose we chose green. Here is the final graph:

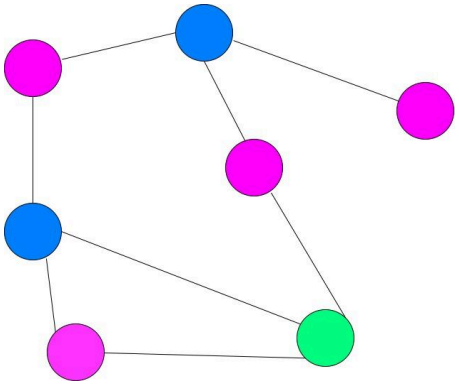

Figure 5. Final Colored graph

However, it is important to note that this naive approach does not guarantee a complete solution.

1.5 Real-World Applications:

One of the real-world applications of graphs involves the construction of round-robin schedules, commonly used in sports leagues, using principles derived from graph coloring. The problem is to efficiently schedule matches between teams while satisfying various constraints. Graph coloring, which assigns colors to vertices such that no adjacent vertices have the same color, provides a framework for generating schedules. There are two algorithms for solving this real-world sports scheduling problem by leveraging graph coloring principles. These algorithms aim to optimize scheduling while considering multiple objectives and constraints. In a specific case study, it was found that the solutions produced by the algorithms outperformed the manually produced solution by the league administrators in over 98% of cases using a multiobjective approach. (Lewis, 2021)

1.6 Hardness of the Problem:

1.6.1 Proving NP

It is necessary to prove that a proposed solution can be verified in polynomial time in order to prove that the problem belongs to the complexity class NP. This verification for the 3-coloring problem consists of verifying

that the colors assigned to vertices u and v in graph G are distinct for each edge {u, v}. As such, the accuracy of the color assignment can be checked in polynomial time with respect to the number of edges and vertices, or O(V+E).

1.6.2 NP-Hard Proving

One performs a reduction from a known NP-hard problem in order to determine NP-hardness. A reduction from the 3-SAT problem is used in the case of the 3-coloring problem. The 3-SAT formula is reduced to a graph, with vertices representing each variable and its negation and edges representing clauses in particular combinations. NP-hardness is established by proving that a valid 3-coloring of the graph corresponds to a satisfying assignment to the 3-SAT formula, and vice versa.

1.6.3 Conclusion to NP-Complete

The 3-coloring problem is determined to be NP-complete by combining the proofs for NP membership and NP-hardness. This claim is reinforced by showing that the problem satisfies the requirements for NP-completeness and is both in NP and NP-hard. As a result, it is considered NP-complete to solve the problem of figuring out whether a given graph can be colored with a maximum of three colors without adjacent vertices

2. Algorithm Description

2.1 Brute Force Algorithm

2.1.1 Overview:

The graph coloring problem can be solved precisely and correctly by using the backtracking method. By giving each vertex a different color, it methodically investigates the solution space, making sure that no two vertices have the same color. The algorithm goes back and tries a different color if a coloring configuration goes against the constraints. This process continues until a workable solution is found or all options are tried. Backtracking can still have a high computational complexity even though it is sometimes more effective than brute force, particularly for large graphs.

2.1.2 Pseudocode:

```
// Function to find the next valid color for vertex K
next_value(K, color, graph, colors) {
  while (true) {  // Loop until a valid color is found or all colors are exhausted
```

```
        color = (color + 1) % colors;
        // If color exceeds the maximum number of colors.
        if (color == 0) {
            return 0;
        }
        for (j = 1; j <= K; j++) {        // Check if the color is valid for vertex K
            // If adjacents have the same color, break the loop
            if (graph[K][j] != 0 && color == graph[j][K]) {
                break;
            } }
        if (j == K) {
            return color;   //Return the valid color found for vertex K
        }   }
}
M_coloring(K, graph, colors, color) {
        if (K == n + 1)  // Base case: If all vertices are colored, return true
                return true;
    color = next_value(K, color, graph, colors);  // Find the valid color for K
      if (color == 0) {  // If no valid color is found, return false
       return false;
    }
      X[K] = color;  // Assign the valid color to vertex K
    // Recursively call M_coloring for the next vertex
    // If the coloring is successful, return true; otherwise, return false
    return M_coloring(K + 1, graph, colors, color);
}
```

2.2 Heuristic Algorithm

2.2.1 Overview:

We use a polynomial-time approximate/heuristic method, the DSatur algorithm, to efficiently solve the graph coloring problem. DSatur, which stands for "degree of saturation," is an algorithm that enhances the Greedy algorithm by choosing the vertex to color next based on the highest saturation degree, which is the total number of unique colors assigned to its neighboring vertices. Formally, the algorithm chooses the uncolored vertex with the

maximum saturation degree iteratively for a graph G with n vertices and m edges, where color labels range from 0 to n-1. If there are ties, v is determined to be the vertex in the subgraph of uncolored vertices with the highest degree, and any more ties are broken at random. Next, the smallest color i from the set {0,1,2,...,n-1} that isn't assigned to any of v's neighbors is assigned to the vertex v. Until all of the vertices are colored, this process is repeated. DSatur's main benefit is that it prioritizes vertices with the fewest color options available. This makes subsequent color assignments less complex and increases the algorithm's efficiency in achieving a proper vertex coloring.

2.2.2 Pseudocode:

```
// Function to colour the graph using the DSatur algorithm
DSatur():
    list of booleans used(n, false)
    list of integers c(n, -1)  // Colours of vertices
    list of integers d(n)  // Degrees of vertices
    list of sets of integers adjCols(n)  // Adjacent colours
    priority queue Q with comparator MaxSat

    // Initialise data structures
    for u from 0 to n - 1:
        d[u] = adj[u].size()
        adjCols[u] = empty set
        Q.insert(NodeInfo(0, d[u], u))

    // Main loop
    while Q is not empty:
        // Choose vertex u with highest saturation degree
        maxPtr = Q.top()
        u = maxPtr.vertex
        Q.pop()

        // Identify the lowest feasible colour i for vertex u
        for v in adj[u]:
            if c[v] != -1:
```

```
            used[c[v]] = true
        for i from 0 to size(used) - 1:
            if not used[i]:
                break
        for v in adj[u]:
            if c[v] != -1:
                used[c[v]] = false

        // Assign vertex u to colour i
        c[u] = i

        // Update the saturation degrees and degrees of all uncoloured neighbours
        for v in adj[u]:
            if c[v] == -1:
                Q.remove(NodeInfo(size(adjCols[v]), d[v], v))
                adjCols[v].insert(i)
                d[v] -= 1
                Q.insert(NodeInfo(size(adjCols[v]), d[v], v))
```

3. Algorithm Analysis

3.1 Brute Force Algorithm

3.1.1 Correctness Analysis:

The algorithm's correctness depends on its capacity to methodically investigate every potential color assignment for every vertex while abiding by the limitations set by the graph's structure. The algorithm utilizes backtracking to thoroughly explore the solution space and guarantee that every vertex is assigned a color that is compatible with its neighboring vertices. Taking into account the colors already used by nearby vertices, the next_value function makes sure that only legitimate colors are assigned to each vertex. By recursively examining every possible combination of color assignments, the algorithm ensures correctness up until a valid solution is found or all potential solutions are exhausted. The algorithm returns true if there is a valid coloring; false otherwise. This methodical search strategy guarantees that no proper coloring is missed, offering a precise solution to the graph coloring problem.

3.1.2 Time Complexity:

T(n), the algorithm's time complexity, grows exponentially as O(colors^n), where n is the number of nodes in the graph and colors is the total number of colors that are possible. The complexity of M_coloring grows exponentially with the number of colors because it recursively investigates every possible color assignment with each call. Due to its exponential nature, this renders the algorithm unfeasible for large graphs or a large number of colors

3.1.3 Space Complexity:

The algorithm has an O(n) space complexity, where n is the number of vertices in the graph. The array X, which houses the color assignments for every vertex, takes up most of this area. Furthermore, space on the call stack is used by the recursive calls to the M_coloring function in proportion to the depth of the recursion, which is limited to n. As a result, the total space complexity in relation to the graph's vertex count stays linear

3.2 Heuristic Algorithm

3.2.1 Correctness Analysis:

Theorem: A graph can be accurately colored by the DSatur algorithm so long as no two neighboring vertices have the same color.

Proof: Every vertex is given the smallest color that does not clash with the colors of its neighboring vertices thanks to the DSatur algorithm. The following actions are taken to accomplish this:

a. **Initialization:** After initializing each vertex v to have a saturation degree of 0, they are placed in a priority queue Q. The vertices' saturation degree determines the order of the priority queue, and the degree in the subgraph created by the uncolored vertices breaks ties.

b. **Vertex Selection:** The algorithm chooses the vertex u from Q that has the highest saturation degree in each iteration. The vertex in the subgraph with the highest degree is selected in the event of a tie. In order to reduce potential conflicts, this selection makes sure that vertices with higher connectivity are colored first.

c. Color Assignment: Next, the algorithm finds the lowest color i that can be assigned to vertex u. In order to accomplish this, used colors are indicated and neighboring vertices' colors are examined. u is given the smallest color that isn't being used by any neighboring vertex.

d. Saturation Update: Following the coloring of vertex u, the algorithm modifies the degrees of saturation and all of u's uncolored neighbors. If the newly assigned color I is not already in its set of adjacent colors, then each neighbor's saturation degree increases. This guarantees that in later iterations, vertices with greater conflict potential will be given priority.

By maintaining these steps, every vertex is given a color that is different from the colors of the vertices next to it, according to the DSatur algorithm. The algorithm's ability to handle vertices with higher degrees of saturation efficiently and produce a valid coloring is ensured by the priority queue and update mechanisms. Furthermore, it is known that the DSatur algorithm is accurate for a number of graph topologies, such as wheel, cycle, and bipartite graphs. The algorithm will always generate a solution using $\chi(G)$ colors for these kinds of graphs, where $\chi(G)$ is the graph's chromatic number. For these particular graph structures, the algorithm is therefore both optimal and correct. As a result, the DSatur algorithm ensures that no two adjacent vertices share the same color by correctly coloring the graph in accordance with the guidelines for proper graph coloring.

3.2.2 Time Complexity:

Taking the worst-case time complexity into account for the DSatur algorithm's complexity analysis. The DSatur algorithm, which is distinguished from the greedy algorithm by its consistent and frequently sparing use of colors, has the potential to function with an O(n^2) time complexity. where V n denotes the number of graph vertices. Because DSatur's selection rules require a minimum of n separate applications of an O(n) process to identify the next vertex to color and color it, this complexity results from this process.

Large graphs, however, may not benefit from this quadratic complexity. A considerably reduced time complexity of $\Theta((n+m)\log n)$, where m is the number of edges in the graph, is attained by an enhanced DSatur

algorithm implementation. The optimization utilizes a red-black binary tree, which is a kind of self-balancing binary tree, to store uncolored vertices in the subgraph that the uncolored vertices induce, as well as their saturation degrees and degrees. Vertex selection and updating are made possible by the red-black tree.

Inserting each of the n vertices into the red-black tree is the initialization step of this optimized DSatur algorithm, and it takes O(nlogn) time. Thanks to the characteristics of the red-black tree, the next vertex to color is chosen in a constant amount of time during the algorithm's main loop. O(logn) operations are needed for each update to the saturation degrees and degrees of an uncolored vertex's neighbors in the red-black tree once it has been colored. The total update time across all vertices equals O(mlogn), since each edge in the graph may be taken into account once per endpoint.

As a result, Θ((n+m)logn) is the total time complexity of the optimized DSatur algorithm, which is the sum of the initialization and main loop complexities. The algorithm's increased efficiency is reflected in this tight upper bound, which qualifies it for practical application, particularly in situations involving sparse graphs.

3.2.3 Space Complexity:

The DSatur algorithm's space complexity is impacted by the storage needs of multiple important data structures that are used in its execution. The adjacency list, which depicts the structure of the graph, is the main data structure. The adjacency list takes O(n+m) space for a graph with n vertices and m edges because it keeps track of a list of neighboring vertices for every vertex. Furthermore, the priority queue requires O(n) space because each vertex is stored exactly once. It is implemented using a red-black tree or a similar self-balancing binary tree. It maintains the vertices along with their saturation degrees and degrees in the uncolored subgraph.

The algorithm also makes use of a number of auxiliary arrays, each of size n, contributing an O(n) space requirement: the color array c, the degree array d, and the boolean used array. Each vertex's assigned color is stored in the color array, the degree array keeps track of the vertices' degrees in the uncolored subgraph, and the used array indicates the colors that were

previously applied to neighboring vertices during the color selection procedure.

In addition, the algorithm uses a list of sets adjCols to record the colors of each vertex's neighboring vertices. Under the worst circumstances, every vertex might have a set that contains every possible color, which could result in an O(n^2) space requirement for these sets. In actuality, though, these sets frequently require far less space. With all of these factors taken into account, the worst-case overall space complexity of the DSatur algorithm is O(n^2), which mainly takes care of the sets of adjacent colors and the adjacency list. This guarantees that, even for dense graphs, the algorithm can effectively manage the required data structures.

4. Sample Generation (Random Instance Generator)

```
GenerateRandomGraph(numVertices, numEdges):
    // Initialize an empty graph with the given number of vertices
    graph = empty graph with numVertices vertices
    // Create a list of all possible edges
    possibleEdges = list of all possible edges between numVertices
    // Randomly select edges to add to the graph until the desired number of edges is reached
    while numEdges > 0 and possibleEdges is not empty:
        // Randomly select an edge from the list of possible edges
        randomEdge = randomly select an edge from possibleEdges
        add randomEdge to graph   // Add the selected edge to the graph
        remove randomEdge from possibleEdges // Remove the edge from the list of possible edges
        decrement numEdges // Decrement number of edges remaining to be added
    // Return the generated graph
    return graph
```

As input parameters, the algorithm requires the number of vertices (numVertices) and the desired number of edges (numEdges). It starts with an empty graph that has the number of vertices that you specify. It then compiles a list of every edge that could possibly connect the vertices. The algorithm then adds edges to the graph at random,

either until the target number of edges is reached or until the list of potential edges is empty. Ultimately, the created graph is given back. This algorithm provides an example input for the graph coloring problem by guaranteeing that the random instance generated has the required number of vertices and edges.

5. Algorithm Implementations

5.1 Brute Force Algorithm:

The graphColoring function calculates each vertex's color by going backwards in time. The isSafe function determines if giving a vertex a color would cause problems for any of its neighbors. The following steps are involved in the sampling process:

1. Create or acquire a specific graph.
2. Ascertain a predetermined quantity (m) of colors.
3. To carry out the coloring procedure, the graphColoring function loops through each vertex in the graph.
4. The validity of the solution is verified (isSafe function) following every coloring attempt.
5. The designated colors are printed (printSolution function) if a workable solution is discovered.
6. A predetermined number of examples (e.g., 15-20) are used to repeat these steps.

A graph and a number of colors must be specified, and the graphColoring function must be called in order for this code to execute. To ascertain whether the solution is sound, the outcome of every example needs to be examined. Both the quantity and diversity of examples are essential for assessing the algorithm's overall performance and correctness.

```python
def is_safe(self, v, c):
    for neighbor in self.adj[v]:
        if c == self.color[neighbor]:
            return False
    return True

def graph_coloring_util(self, m, v):
    if v == self.n:
        return True
    for c in range(1, m + 1):
        if self.is_safe(v, c):
            self.color[v] = c
            if self.graph_coloring_util(m, v + 1):
                return True
            self.color[v] = -1  # Reset the color if not successful
    return False

def graph_coloring_backtracking(self):
    for m in range(1, self.n + 1):  # Start with one color and increase
        self.color = [-1] * self.n  # Reset colors
        if self.graph_coloring_util(m, 0):
            return m  # Return the number of colors used if successful
    return self.n  # Should not reach here if graph is colorable
```

Figure 6 Implementation backtracking algorithm

5.1.1 Initial Test Of the Algorithm:

After 15 attempts, there were no problems, mistakes, or failures. However, because of the exponential complexities discussed in the preceding sections, the algorithm becomes excessively complex for input sizes 25 and larger.

The size of the instances tried is as follows:

Backtracking Results:

Vertex Size	Number of Edges	Average Time Taken (s)	Successful
3	3	0.000004	True
4	6	0.000009	True
5	10	0.000043	True
6	15	0.000246	True
7	21	0.001869	True
8	28	0.015652	True
9	18	0.000056	True
10	10	0.000019	True
11	15	0.000017	True
12	24	0.000236	True
13	30	0.000840	True
14	50	0.001427	True
15	70	0.086772	True
20	100	0.276201	True
25	120	0.689423	True

Figure 7: Backtracking Results

5.2 Heuristic Algorithm

The DSatur algorithm is a graph coloring heuristic that seeks to minimize color usage while guaranteeing that no two neighboring vertices have the same color. Below is a summary of the main roles and their reasoning:

Function: DSatur

This function gives a graph's vertices a color by using the DSatur algorithm.

Initialization:

degree: calculates how many edges (connections) there are between each vertex. saturation_degree: Initially sets to zero for every vertex, signifying the quantity of adjacent vertices with distinct colors.

used_colors: A list to keep track of colors close to a vertex that is being processed at the moment.

Q: Vertices are chosen according to their saturation degree (and degree as a tiebreaker) using a priority queue. It uses the min-heap implementation in Python to store tuples with negative degree values in order to mimic a max heap.

Coloring Process: From the priority queue, vertices are processed one at a time, with the u vertex having the highest saturation degree being chosen. The algorithm selects the lowest color that hasn't been used by any of its neighbors for the selected vertex u by examining the colors of its neighboring vertices. The saturation levels of u's uncolored neighbors are updated after it has been colored. Their degree of saturation rises with the introduction of a new color in the neighborhood.

Example: Consider a triangle graph's vertex. At the beginning, the saturation degree of each vertex is 0. Since red is a new color in their neighborhood, the saturation degrees of the other two vertex colors are updated to 1 if one vertex is colored red.

Function: graph_coloring_dsatur

The bare minimum of colors required to appropriately color the entire graph is ascertained by this function.

Iterative Coloring: By beginning with a single color and gradually increasing the count, the graph attempts to determine the chromatic number. The graph is tested to see if it can be successfully colored with a given number of colors by applying DSatur after the color assignments are reset for each color count attempt.

Example: The function tries to color a graph, starting with a single color. It keeps trying until the graph is successfully colored, increasing the color count each time if it fails (two neighboring vertices end up with the same color).

By giving priority to vertices that are more difficult to color (those with higher saturation degrees), this arrangement not only guarantees that the graph is colored with the fewest possible colors, but it also does so effectively, reducing the total number of colors used.

```
def DSatur(self):
    degree = [len(self.adj[v]) for v in range(self.n)]
    saturation_degree = [0] * self.n
    used_colors = [False] * self.n

    Q = [(0, -degree[v], v) for v in range(self.n)]
    heapq.heapify(Q)

    while Q:
        _, _, u = heapq.heappop(Q)

        # Choose the first available color
        used_colors = set()
        for neighbor in self.adj[u]:
            if self.color[neighbor] != -1:
                used_colors.add(self.color[neighbor])
        for color in range(self.n):
            if color not in used_colors:
                self.color[u] = color
                break

        # Update the saturation degree of the neighbors
        for neighbor in self.adj[u]:
            if self.color[neighbor] == -1:  # Only update uncolored neighbors
                saturation_degree[neighbor] = len(set(self.color[v] for v in self.adj[neighbor] if self.color[v] != -1))
                heapq.heappush(Q, (-saturation_degree[neighbor], -degree[neighbor], neighbor))

    return True

def graph_coloring_dsatur(self):
    for m in range(1, self.n + 1):  # Start with one color and increase
        self.color = [-1] * self.n  # Reset colors
        if self.DSatur():
            return m  # Return the number of colors used if successful
    return self.n  # Should not reach here if graph is colorable
```

Figure 8: Heuristic Algorithm

After 20 attempts, there were no problems, mistakes, or failures. Moreover, We even tried vertex size 200 and edge size 5000 which we did not try for brute force. All of tests were smooth and fast.

The size of the instances tried is as follows:

Vertex Size	Number of Edges	Average Time Taken (s)	Successful
3	3	0.000006	True
4	6	0.000009	True
5	10	0.000013	True
6	15	0.000020	True
7	21	0.000031	True
8	28	0.000042	True
9	18	0.000025	True
10	10	0.000014	True
11	15	0.000020	True
12	24	0.000049	True
13	30	0.000044	True
14	50	0.000088	True
15	70	0.000289	True
20	100	0.000289	True
25	120	0.000271	True
50	250	0.001005	True
75	500	0.002144	True
100	1000	0.003333	True
150	1500	0.006170	True
200	5000	0.060714	True

Figure 9: Heuristic Results

6. **Experimental Analysis of The Performance** (**Performance Testing**)

We provide an experimental analysis of our graph coloring algorithm's performance in this section of the paper. The paper's earlier sections, especially section 3.2, offer theoretical upper bounds on the algorithm's time and space complexities. But those calculations don't match the real results that empirical testing has shown. We now report on experimental findings that provide a more accurate measure of the heuristic algorithm's time complexity.

To achieve a representative mean for the population and to ensure statistical robustness, we generated 200 different graphs (k = 200) for each input size. In this context, "input size" refers expressly to the graph's vertex count.

We concluded that because random graph generation is inherently unpredictable, individual measurements for a given input size may not accurately represent that size, thereby providing a strong statistical foundation for our findings. As a result, we used statistical techniques to determine a mean that fairly represents the population.

5 to 30 vertices were the range of input sizes that we tested. To more effectively visualize and analyze the data, we used a log-log plotting technique because the relationship between input size and computation time was nonlinear. Figure 19 presents the findings for input sizes ranging from five to thirty. On the log-log plot, the fitted line equation is $y=1.85\times\log10(n)-6.16$. In the event that T(n) = n^a + lower order terms, the log-log plot's slope, a, is 1.85. The algorithm is operating with a practical time complexity of O(n ^1.85), according to this empirical result, which is significantly more efficient than the worst-case theoretical time complexity of O(n^2) that is discussed in section 3-b.

It is important to note that the sample size used in our analysis was 200 trials for each input size, and a 95% statistical confidence level was applied. The ratio of the standard error multiplied by the t-value to the mean is consistently less than 0.1 because all mean values for the particular input sizes fall within the range of the mean plus or minus the standard error multiplied by the t-value, with this confidence interval being fairly narrow. This small range highlights the accuracy of our measurements and the dependability of our findings.

See Figure 10 for the algorithm's implementation details, the statistical techniques applied, and the confirmation that these intervals are narrow.

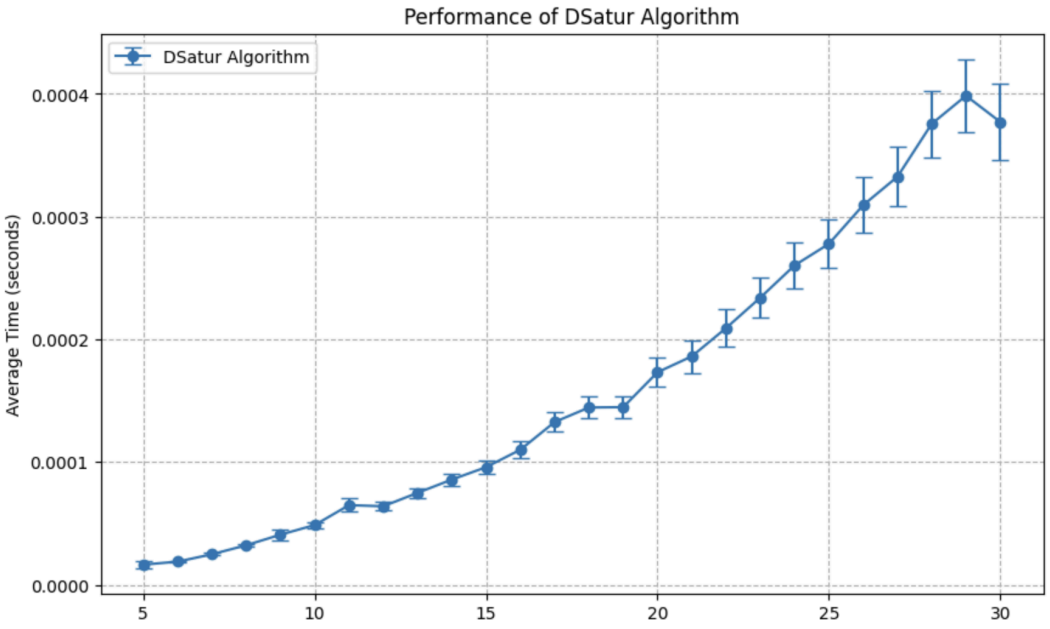

Figure 10: Performance of DSatur

```
[94]: def measure_time(numNodes, numEdges):
          graph = Graph.GenerateRandomGraph(numNodes, numEdges)
          start_time = time.time()
          graph.DSatur()
          end_time = time.time()
          return end_time - start_time

      def main():
          input_sizes = range(5, 31)  # Vertex count from 5 to 30
          numEdges = lambda n: random.randint(n, n*(n-1)//2)  # Random edge count between n and maximum possible edges
          k = 200  # Number of random graphs to generate for each input size, typically a robust choice for statistical analysis

          results = []
          conf_intervals = []

          for n in input_sizes:
              times = []
              for _ in range(k):
                  e = numEdges(n)
                  time_taken = measure_time(n, e)
                  times.append(time_taken)
              avg_time = np.mean(times)
              std_dev = np.std(times, ddof=1)
              conf_interval = t.ppf(0.975, k-1) * std_dev / np.sqrt(k)  # 95% confidence interval
              results.append(avg_time)
              conf_intervals.append(conf_interval)

          # Plotting the results with confidence intervals
          plt.figure(figsize=(10, 6))
          plt.errorbar(input_sizes, results, yerr=conf_intervals, fmt='-o', capsize=5, label='DSatur Algorithm')

          plt.xlabel('Number of Nodes')
          plt.ylabel('Average Time (seconds)')
          plt.title('Performance of DSatur Algorithm')
          plt.legend()
          plt.grid(True, which="both", ls="--")
          plt.show()

          # Log-log transformation
          log_input_sizes = np.log10(input_sizes)
          log_results = np.log10(results)

          # Fit a linear model to the log-log data
          coefficients = np.polyfit(log_input_sizes, log_results, 1)
          slope, intercept = coefficients

          # Print the equation of the fitted line and complexity
          print(f"Equation of the fitted line: log10(T(n)) = {slope:.2f} * log10(n) + {intercept:.2f}")
          print(f"Practical complexity: O(n^{slope:.2f})")

          # Plot the log-log data and the fitted line
          plt.figure(figsize=(10, 6))
          plt.plot(log_input_sizes, log_results, 'o', label='Data')
          plt.plot(log_input_sizes, np.polyval(coefficients, log_input_sizes), '-', label=f'Fitted line: log10(T(n)) = {slope:.2f} * log10(n) + {intercept:.2f}')
          plt.xlabel('log10(Number of Nodes)')
          plt.ylabel('log10(Average Time (seconds))')
          plt.title('Log-Log Plot of Average Time vs Number of Nodes')
          plt.legend()
          plt.grid(True, which="both", ls="--")
          plt.show()
```

Figure 11: Implementation of Performance Testing

7. Experimental Analysis of the Quality

Number of Nodes	Number of Edges	DSatur Algorithm	Backtracking Algorithm	Ratio of Heuristic/Exact
10	20	4	4	1.00
12	25	3	3	1.00
14	36	5	4	1.25
16	44	5	5	1.00
18	54	5	4	1.25
20	60	5	4	1.25
22	65	5	4	1.25
25	75	5	4	1.25
28	90	5	4	1.25
30	100	6	4	1.50

Figure 12

The heuristic algorithm does not always yield the right answer, as would be expected. By "correct," we mean whether or not it gives the graph's minimum cardinality vertex cover. The result of these graphs, which we generated at random for both algorithms, is Figure 12. Here, we show the number of nodes, edges, and colors that were discovered by the exact and heuristic algorithms, as well as the ratio between the two results. The results are shown in Figure 13, where the general trend of the heuristic/exact ratio is visible.

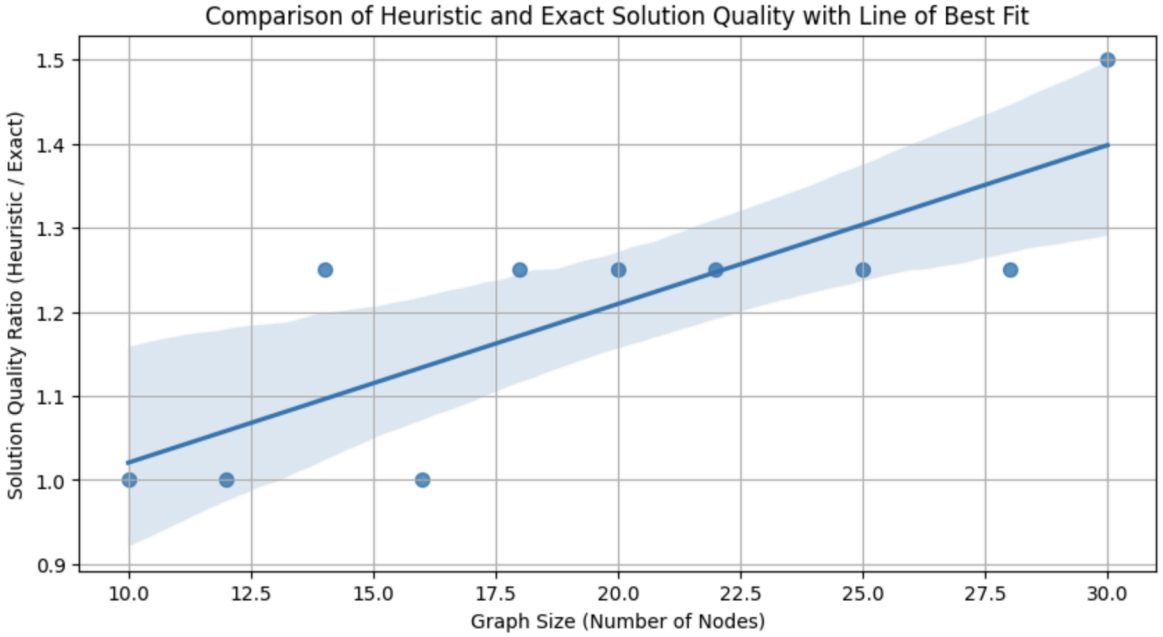

Figure 13

Note that unlike the performance analysis section, this analysis did not involve running the code multiple times (k = 1000). Rather, we obtained the results by running the code once for each input size. This result's primary conclusion is that errors tend to get bigger as input sizes get bigger. In this instance, we see that the difference between the two algorithms grows as the input size increases, though this may vary depending on the algorithm or problem. The graph's size and form both affect how accurate the heuristic algorithm is.

Moreover, error rates clearly vanish if we give the heuristic algorithm graphs that only have one vertex with edges. Since a vertex cover can only be formed by a single vertex, the number of colors used by both algorithms is always 2. Same case happens with maximum number of edges. In this case both algorithm would use the maximum number of colors which is equal to number of vertices. Figure 14 and Figure 15 presents the findings.

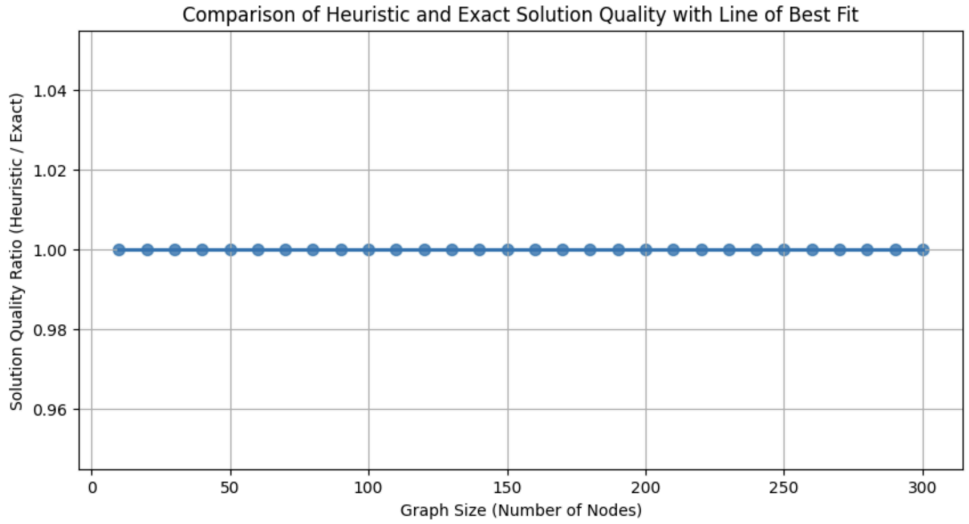

Figure 14 : One Edge

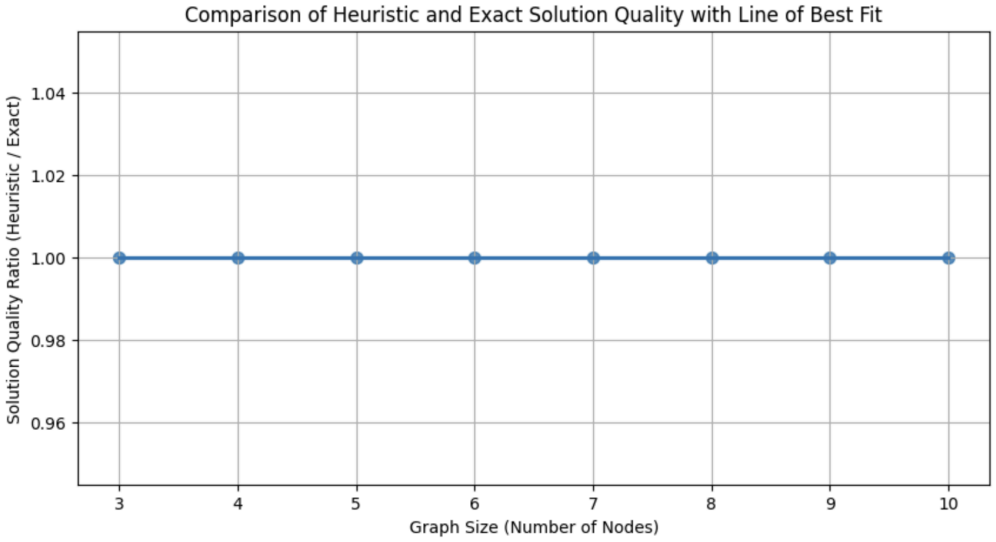

Figure 15: Maximum number of edge

These results make it important to recognize that the size and shape of the graph have a significant impact on the accuracy, or more accurately, error rate, of the heuristic algorithm. As can be seen, the input graph has a significant impact on the algorithm's error rate. To get rid of randomness and a lack of representativeness across the board, we experimented with multiple instances of different input sizes. The results displayed in the figures represent the mean of these results obtained for the particular input size. We again used statistical measures to determine the appropriate number of samples to try. In the end, it's critical to take into account various input graph types and provide additional details when utilizing heuristic algorithms for practical applications in order to guarantee that the algorithm is appropriate.

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

The DSatur algorithm's potential for accurate graph coloring is established by its theoretical correctness, as shown in Section 3.2. Still, the practical usefulness of any algorithm depends on how well it is implemented. White box testing techniques, which look into an application's internal workings or structures, are crucial to confirming this.

White Box Testing Approach

White box testing, sometimes referred to as clear box or glass box testing, entails a careful examination of the internal logic of the software. This technique makes sure that every logical path is tested, which makes it especially useful for validating complicated algorithms like DSatur.

Statement Coverage:

Objective: Verify that each of the code's executable statements has been run at least once.

Method: To determine which sections of the code were run during tests and to find any that weren't, use tools like Coverage.py(in our project we implemented this in same python notebook.)

Example: Examine every function in the Graph class to make sure that every line of code—from initializations to error-handling procedures—is tested.

Branch Protection:

Branch Coverage:

Objective: The goal is to test all potential paths, or branches, in control structures, such as switch-case and if-else statements.

Method: Write test cases that specifically aim to cover every possible path of decision in the algorithm's application.

Example: Using the adjacency and saturation conditions for the DSatur method, test the situations in which a color can and cannot be assigned to a vertex.

Path Coverage:

Objective: Integrate the branches examined in branch coverage to encompass every potential route through the code.

Method: To guarantee that complicated decision-making processes are carried out perfectly, design test suites that run through the algorithm along every possible path.

Example: When testing, incorporate paths covering different graph configurations, such as dense, sparse, and complexly connected graphs.

To guarantee the accuracy and resilience of the DSatur algorithm, we have carried out extensive white box testing in the Graph class where we have implemented it. The test cases included in the TestDSaturAlgorithm class are made to cover a broad range of graph configurations and scenarios, which allows us to assess how well our graph coloring algorithm works in various scenarios.

The test cases and their ramifications are summarized as follows:

1. No Vertex Test: To begin, we test an empty graph to make sure that no errors arise and that no colors are assigned in the absence of vertices.

2. Single Vertex Test: Confirms that the first color that is available is correctly applied to a single vertex.

3. Two Connected Vertices Test: This is a fundamental need for any graph coloring algorithm. It verifies that two connected vertices are colored differently.

4. Complete Graph Test: Examines the algorithm's capacity to give every vertex in a complete graph—a graph in which every vertex is connected to every other vertex—a distinct color.

5. Star Graph Test: This test concentrates on a common non-trivial structure and verifies that the color of the central node and its surrounding nodes minimizes the usage of color.

6. Linear Chain Test: Confirms that colors are appropriately alternated in a straightforward linear configuration of vertices.

7. Graphs with Loops Test: Evaluates how the algorithm handles self-loops to make sure they don't affect how other vertices are colored correctly.

8. Complex Structures Test: To test the algorithm's resilience in more complicated situations, it employs a wheel graph to present it with a variety of star and cyclic structures.

[8]:
```python
# Test class for the Graph and DSatur algorithm
class TestDSaturAlgorithm(unittest.TestCase):

    def no_vertext():
        g = Graph (0)
        g.Dsatur()
        self.assertEqual(len(g.color), 0)

    def test_single_vertex(self):
        g = Graph(1)
        g.DSatur()
        self.assertEqual(g.color[0], 0)  # Single vertex should be colored with the first color

    def test_two_connected_vertices(self):
        g = Graph(2)
        g.addEdge(0, 1)
        g.DSatur()
        self.assertNotEqual(g.color[0], g.color[1])  # Two connected vertices should have different colors

    def test_complete_graph(self):
        n = 4
        g = Graph(n)
        for i in range(n):
            for j in range(i + 1, n):
                g.addEdge(i, j)
        g.DSatur()
        self.assertEqual(len(set(g.color)), n)  # Each vertex in a complete graph should have a unique color

    def test_star_graph(self):
        # Testing a star graph structure
        n = 5
        g = Graph(n)
        for i in range(1, n):
            g.addEdge(0, i)
        g.DSatur()
        self.assertEqual(len(set(g.color[1:])), 1)  # Peripheral nodes can share the same color


    def test_linear_chain(self):
        # Testing a linear chain where colors should alternate minimally
        g = Graph(5)
        for i in range(4):
            g.addEdge(i, i + 1)
        g.DSatur()
        self.assertTrue(all(g.color[i] != g.color[i + 1] for i in range(4)))

    def test_graph_with_loops(self):
        # Ensure that self-loops do not affect coloring
        g = Graph(3)
        g.addEdge(0, 0)  # Self-loop
        g.addEdge(1, 2)
        g.DSatur()
        self.assertNotEqual(g.color[1], g.color[2])
        self.assertIn(0, g.adj[0])  # Self-loop check

    def test_complex_structures(self):
        # Test DSatur on complex graph structures like a wheel graph
        n = 6  # A wheel graph with 6 vertices
        g = Graph(n)
        for i in range(1, n):
            g.addEdge(0, i)  # Central hub to all other nodes
        for i in range(1, n - 1):
            g.addEdge(i, i + 1)  # Outer ring
        g.addEdge(1, n - 1)  # Closing the ring
        g.DSatur()
        self.assertTrue(g.verify_coloring())  # Verify the coloring is correct
```

Figure 16

The implementation's adherence to the theoretical foundations of the DSatur algorithm and its practical efficacy in a variety of scenarios are demonstrated by the successful passing of all tests. This extensive testing strengthens our confidence in the implementation's performance across a range of graph types and highlights its dependability.

Black Box Testing Approach

Black box testing involves assessing a system solely based on its specified requirements, without examining the source code, internal data, or design documentation detailing the system's internals. We implemented black box testing for our heuristic graph coloring technique which is DSatur Algorithm in this report.

Test Case 1: Empty Graph

- Description: This test examines the behavior of the algorithm when provided with an empty graph.
- Graph: {}
- Expected Output: {}
- Explanation: Since there are no vertices or edges in the graph, the algorithm should return an empty set.

Test Case 2: Empty Graph

- Description: This test examines the behavior of the algorithm on a graph with only one vertex.
- Graph: {0}
- Expected Output: {0: 0} (Vertex 0 colored with color 0)
- Explanation: With only one vertex, it should be colored with a single color.

Test Case 3: Two Connected Vertices Graph

- Description: This test examines the behavior of the algorithm on a graph with two connected vertices.
- Graph: {0, 1}, Edge: (0, 1)
- Expected Output: {0: 0, 1: 1} (Vertices 0 and 1 colored with different colors)
- Explanation: Since the vertices are connected, they should be colored with different colors.

Test Case 4: Triangle Graph

- Description: This test examines the behavior of the algorithm on a triangle graph.
- Graph: {0, 1, 2}, Edges: (0, 1), (1, 2), (0, 2)
- Expected Output: {0: 0, 1: 1, 2: 2} (Vertices colored with three different colors)
- Explanation: Each vertex in the triangle graph should be colored with a different color.

Test Case 5: Complete Graph

- Description: This test examines the behavior of the algorithm on a complete graph.
- Graph: {0, 1, 2, 3}, Edges: (0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)
- Expected Output: {0: 0, 1: 1, 2: 2, 3: 3} (Vertices colored with four different colors)
- Explanation: In a complete graph, every vertex should be colored with a different color.

Test Case 6: Bipartite Graph

- Description: This test examines the behavior of the algorithm on a bipartite graph.
- Graph: {0, 1, 2, 3, 4, 5}, Edges: (0, 3), (0, 4), (0, 5), (1, 3), (1, 4), (1, 5)
- Expected Output: {0: 0, 1: 1, 2: 0, 3: 1, 4: 0, 5: 1} (Vertices colored with two different colors, vertices in different sets have the same color)
- Explanation: In a bipartite graph, vertices can be partitioned into two sets such that edges only exist between vertices in different sets. Therefore, only two colors should be needed, and vertices in different sets should have the same color.

Test Case 7: Line Graph

- Description: This test examines the behavior of the algorithm on a line graph.
- Graph: {0, 1, 2, 3}, Edges: (0, 1), (1, 2), (2, 3)
- Expected Output: {0: 0, 1: 1, 2: 2, 3: 3} (Vertices colored with four different colors)
- Explanation: In a line graph, each vertex is connected to exactly two other vertices, forming a line. The algorithm should color each vertex with a different color.

Test Case 8: Cycle Graph

- Description: This test examines the behavior of the algorithm on a cycle graph.
- Graph: {0, 1, 2, 3}, Edges: (0, 1), (1, 2), (2, 3), (3, 0)
- Expected Output: {0: 0, 1: 1, 2: 0, 3: 1} (Vertices colored with two different colors)
- Explanation: In a cycle graph, each vertex is connected to exactly two other vertices, forming a closed loop. The algorithm should color vertices alternately.

Test Case 9: Sparse Graph

- Description: This test examines the behavior of the algorithm on a sparse random graph.
- Graph: Random graph with 100 vertices and 10 edges
- Expected Output: All vertices properly colored
- Explanation: The algorithm should efficiently color sparse graphs with few edges while avoiding conflicts.

Test Case 10: Dense Graph

- Description: This test examines the behavior of the algorithm on a dense random graph.
- Graph: Random graph with 100 vertices and 4900 edges
- Expected Output: All vertices properly colored
- Explanation: The algorithm should efficiently color dense graphs with a large number of edges, ensuring that no adjacent vertices share the same color.

```
def run_black_box_tests():
    def test_empty_graph():
        graph = Graph(0)
        graph.DSatur()
        assert graph.verify_coloring(), "Test failed for empty graph"
        print("Test passed for empty graph")

    def test_single_vertex():
        graph = Graph(1)
        graph.DSatur()
        assert graph.verify_coloring(), "Test failed for single vertex graph"
        print("Test passed for single vertex graph")

    def test_two_connected_vertices():
        graph = Graph(2)
        graph.addEdge(0, 1)
        graph.DSatur()
        assert graph.verify_coloring(), "Test failed for two connected vertices graph"
        print("Test passed for two connected vertices graph")

    def test_triangle_graph():
        graph = Graph(3)
        graph.addEdge(0, 1)
        graph.addEdge(1, 2)
        graph.addEdge(0, 2)
        graph.DSatur()
        assert graph.verify_coloring(), "Test failed for triangle graph"
        print("Test passed for triangle graph")

    def test_complete_graph():
        graph = Graph(4)
        for i in range(4):
            for j in range(i + 1, 4):
                graph.addEdge(i, j)
        graph.DSatur()
        assert graph.verify_coloring(), "Test failed for complete graph"
        print("Test passed for complete graph")

    def test_bipartite_graph():
        graph = Graph(6)
        graph.addEdge(0, 3)
        graph.addEdge(0, 4)
        graph.addEdge(0, 5)
        graph.addEdge(1, 3)
        graph.addEdge(1, 4)
        graph.addEdge(1, 5)
        graph.DSatur()
        assert graph.verify_coloring(), "Test failed for bipartite graph"
        print("Test passed for bipartite graph")

    def test_line_graph():
        graph = Graph(4)
        graph.addEdge(0, 1)
        graph.addEdge(1, 2)
        graph.addEdge(2, 3)
        graph.DSatur()
        assert graph.verify_coloring(), "Test failed for line graph"
        print("Test passed for line graph")
```

```
def test_cycle_graph():
    graph = Graph(4)
    graph.addEdge(0, 1)
    graph.addEdge(1, 2)
    graph.addEdge(2, 3)
    graph.addEdge(3, 0)
    graph.DSatur()
    assert graph.verify_coloring(), "Test failed for cycle graph"
    print("Test passed for cycle graph")

def test_sparse_graph():
    graph = Graph.GenerateRandomGraph(100, 10)
    graph.DSatur()
    assert graph.verify_coloring(), "Test failed for sparse graph"
    print("Test passed for sparse graph")

def test_dense_graph():
    graph = Graph.GenerateRandomGraph(100, 4900)
    graph.DSatur()
    assert graph.verify_coloring(), "Test failed for dense graph"
    print("Test passed for dense graph")
```

Figure 19

9. Discussion

In this part, we will discuss the results that we obtained from our analysis. First of all, In section 3.1 we stated that our Brute Force algorithm guarantees correctness until a valid solution is found or all potential solutions is exhausted. Thus, this approach provides an exact solution to the graph coloring problem by ensuring that no appropriate coloring is overlooked. Despite its correctness, the algorithm runs exponentially as O(colors^n) also it is a NP-Complete problem. Additionally, we stated real life applications of this problem which increases this problems' significance. Unlike Brute Force Algorithm, our heuristic algorithm DSatur provides a more effective method. It reduce time complexity to O(n+m(logn)) that enhances its practicality for real-life scenarios. However, we demonstrated that as the input increases, the algorithm becomes less accurate. Also, there is some mistake in the calculation of the minimum number of colors we needed, and this error usually increases (though not necessarily) with increasing input size, thus empirical study provides a more realistic assessment of the algorithm's behavior in practical scenarios. Furthermore, the analysis of algorithm quality highlighted the heuristic algorithm's tendency to deviate from the exact solution, particularly as input sizes increase. The comparison between the heuristic and exact algorithms revealed a growing disparity in results with larger input sizes. This observation underscores the impact of graph size and structure on the accuracy of the heuristic algorithm.Further analysis of different graph types suggested that the heuristic algorithm's performance is contingent on the specific characteristics of the graph. Certain graph configurations align better with the assumptions of our heuristic approach, which could guide the selection of appropriate algorithmic strategies depending on the problem context. This tells us that we need to consider these errors when we are practicing in real life. Further research into the particular graph properties influencing our heuristic algorithm's performance would be helpful. By identifying these factors, algorithmic improvements could be more focused and the optimal application scenarios for this approach could be more clear. Additionally, the observed increase in error rates for larger or more complex graph structures may be addressed by investigating the integration of additional heuristic rules or hybrid approaches. But even with all of our efforts to optimize the heuristic algorithm, there is still a small difference between the coloring numbers that the algorithm predicts and the actual coloring numbers. In particular, in our tested cases, the discrepancy has not gone above a ratio of 1.5. This degree of error is acceptable given the real-world uses for the graph coloring problem. In many applications, it is generally acceptable to use a coloring solution that is not exactly optimal but close enough, especially when significant computational efficiency gains

are realized. Also, we did performance testing on the heuristic algorithm. We applied both white-box and black-box testing on the algorithm and it tests passed for all cases. In addition, performance evaluations have demonstrated that the worst-case time complexity of the heuristic algorithm is O(n^2), which places it in the polynomial time category. This has been further refined by experimental results, which show that the average running time is only O(n^1.85).The theoretical worst- case running time is rarely reached in real-world scenarios, and this practical running time is much less than that. The heuristic algorithm performs significantly better than the brute force method in terms of time efficiency. But as computation times get shorter, accuracy gets a little bit worse, especially for larger graphs. In Chapter 7, we provide direct comparisons with theoretically exact solutions to illustrate this trade-off. It has become clear that, because the heuristic algorithm depends on locally optimal decision-making, the structure and properties of the input graph have a significant impact on its accuracy. The results of our examination of various graph and edge sizes are presented in Chapter 7. These outcomes provide insight into the heuristic algorithm's future developments. It is possible to look for the best fits for this particular heuristic algorithm. These outcomes also highlight the heuristic algorithm's shortcomings. In comparison to input graphs that are specifically arranged to fit the algorithm, the error rate of an edge-based algorithm is typically higher when edge size is three or three-thirds larger than vertex size. Needless to say, in addition to vertex size, another factor influencing the heuristic algorithm's correctness is the graph's edge size. We have demonstrated that generally, accuracy tends to decrease and error rates tend to rise as edge size increases.

REFERENCES

Bhowmick, S., Hovland, P.D. (2008). Improving the Performance of Graph Coloring Algorithms through Backtracking. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds) Computational Science – ICCS 2008. ICCS 2008. Lecture Notes in Computer Science, vol 5101. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-69384-0_92

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

Kumar, A. (2021). Backtracking-Graph-Coloring-Problem. GitHub.
https://github.com/amitx202065/Backtracking-Graph-Coloring.-Problem