Comprehensive Analysis of Graph Coloring Algorithms

Algorithmic Comparison of Graph Coloring Techniques: Evaluating Solutions and Strategies

Venkata Tejaswi Battula
CS-584-002 Winter 2024
Algorithm Design and Analysis
Portland State University

***Abstract*—This paper delves into the Graph coloring Optimization Problem, where the objective is to find the minimum number of colors needed to color a graph such that no two adjacent vertices share the same color. We explore various algorithms, including Greedy Sequential, Welsh-Powell, DSATUR (Degree of Saturation), Backtracking, Backtracking with Forward Checking, CSP (Constraint Satisfaction Problem). The research addresses the inherent complexities of the problem. The paper aims to provide a comprehensive analysis of these algorithms. The ultimate goal is to shed light on these methodologies, constraints, and performance metrics and contribute valuable insights to the field of graph theory and algorithmic optimization.**

I. Introduction

Graph coloring problem is described as a graph that needs to be colored such that no two adjacent vertices have the same color. Here the number of colors is usually represented by m. There are different flavors to this problem. When the number of colors is given, we need to determine whether these will be sufficient to color the graph or not, this is defined as the "M-coloring Decision problem". Another problem is where we need to find the minimum number of colors required to color the graph. This is called the "Graph coloring optimization problem". In this paper, we are concentrating on the "Graph coloring optimization problem". M-coloring Decision is an NP-complete problem which means we can find a solution whether the coloring is possible or not in polynomial time. On the other hand, the Graph coloring optimization problem is an NP-hard problem. We can not determine whether the number of colors used is minimal. The minimum number of colors used to color a graph is called "chromatic number". So Here our ultimate goal is to find the chromatic number or the number closest to the chromatic number of the given graph. the different approaches to solving the problem, their constraints, how they solve the problem, their complexities, and their analysis will be discussed.

II. Applications and Significance

Direct application for graph coloring is coloring the map where no two adjacent countries/states will be of the same color. Finding the minimum of colors will help in reducing the printing cost. But this is not the actual application. The graph coloring ensures that connected entities which are represented as nodes in a graph, do not share the same resource or attributes. Resource allocation problems, assigning frequencies to wireless communication channels, and distributing tasks among parallel processors are some of the applications for graph coloring.

Take the example of resource allocation. Imagine an educational institution wants to schedule exams for multiple courses and no student should have exam timings which are conflicting with other exams. This can be modeled as a graph where each node represents an exam and the edge connecting the nodes indicates that those exams can not be conducted at the same time. We can solve this problem by applying graph coloring and scheduling the exams.

Considering these applications it is important to implement ways to solve this problem with less complexity. Graph coloring has always been a research area. Therefore, it is worthwhile to analyze the algorithms used to solve graph coloring problems.

III. Greedy

Greedy algorithms are usually chosen to tackle graph coloring problems because of their simplicity and efficiency. The core idea is to iterate through vertices and assign them colors based on particular criteria. While greedy algorithms may not always produce optimal results, they provide practical and fast solutions.

Here we explore three different greedy algorithms. the Greedy Sequential Algorithm, the Welsh-Powell Algorithm, and the DSATUR (Degree of Saturation) Algorithm. Each of them implement a greedy paradigm yet implements different strategies.

A. Greedy Sequential Algorithm

This technique uses the fundamental greedy algorithm. Each vertex is assigned to the lowest numbered available color (colors are represented via integers and assigned to corresponding colors at the end) that none of its adjacent vertices have used before. This is a greedy approach because it minimizes the usage of number of colors at every iteration, Ultimately minimizing the count for the complete input set. However, this approach won't always give the most optimal solution,

but it does offer an upper bound d+1. This indicates that the algorithm will not use more than d+1 colors, here d is the maximum degree of a vertex in the graph. Although it does not assure an optimal solution, it ensures a solution.

Algorithm 1: Greedy Sequential Algorithm

Data: adjacencyList, numberOfVertices
Result: Result array

```
result ← array of size numberOfVertices initialized with −1;
result[0] ← 0;
available ← array of size numberOfVertices initialized with False;
for u in range(1, numberOfVertices) do
    for i in adjacencyList[u] do
        if result[i] ≠ −1 then
            available[result[i]] ← True;
    cr ← 0;
    while cr < numberOfVertices do
        if available[cr] == False then
            break;
        cr ← cr + 1;
    result[u] ← cr;
    for i in adjacencyList[u] do
        if result[i] ≠ −1 then
            available[result[i]] ← False;
return result;
```

Explanation: This algorithm starts by initializing an array 'result' of size V with -1, where V is the number of vertices. This array contains the final assigned colors to corresponding vertices. For the first vertex, we assigned the first color. Also, we have taken an available color boolean array with size V. 'available[i]=True' indicates that ith color is already taken by the adjacent nodes of the current node and we can't assign this color to our current vertex. The algorithm loops to all the vertices, processes the availability, and assigns the next available color to the current node. Finally resets the availability array.

Correctness: Let's discuss the corner cases. If the graph is empty the algorithm works as V will be 0. It terminates without further processing.

As we are already assigning the first color to the first node the algorithm terminates without looping if a graph containing a single node.

Even if the graph is complex with numerous vertices and edges the algorithm still works as it executes in polynomial time.

As the algorithm is iterating through all the nodes it works for the graph with disconnected nodes.

B. Welsh-Powell Algorithm

This algorithm is used to find the chromatic number of the graph. It follows the greedy approach as it takes a node with a greater degree (number of edges connected to a graph) first to color. After sorting the vertices based on the degree, it iterates through this list and assigns colors. For the first node, it assigns a color and iterates the list to use the same color for other nodes. Once a node is found that does not conflict with the adjacent color rule, it assigns that color. And proceeds to do the same for other nodes.

Algorithm 2: Welsh-Powell Coloring Algorithm

Data: adjacencyList, numberOfVertices
Result: Result array

```
sortedVertices ← sortVerticesByDegree(adjacencyList, numberOfVertices);
colors ← array of size numberOfVertices initialized with −1;
assignedColors ← empty set;
for vertex in sortedVertices do
    neighborsWithColors ← set of colors of neighbors of vertex;
    availableColors ← assignedColors − neighborsWithColors;
    if availableColors is not empty then
        colors[vertex] ← min(availableColors);
    else
        newColor ← max(colors) + 1;
        colors[vertex] ← newColor;
        assignedColors.add(newColor);
```

Explanation: The adjacency list generated from the graph is sorted based on the degree. An array of size V is initialized with -1 which stores the result. An *assignedColor* set is taken to store all the colors used. Now, the algorithm loops through all the vertices from the sorted list, and for the current node, it checks the neighbor's colors and adds all of them to *neighbors_with_colors* list. Then we will find the *availableColors* that we can use to color the current node. This can be calculated by removing all the *neighbors_with_colors* from *assignedColors*. We can get the list of colors that we can use which is stored in *availableColors*. If this list is empty, then we will assign a new color to the node. Otherwise, we will take the least color from the *availableColor* list and assign it to our current node.

Correctness: Similar to the greedy sequential algorithm it works well for empty, single-node, complex, and disconnected graphs.

If multiple nodes have the same degree we are proceeding with the node with the lowest index. Because of this, we can not always guarantee optimal results. There might be a possibility that taking another node with the same degree results in less number of colors.

In the above crown graph our algorithm colors based on the ordering. Consider the second case, 4 colors are used which is more than expected. In cases like these, *Welsh Powell*

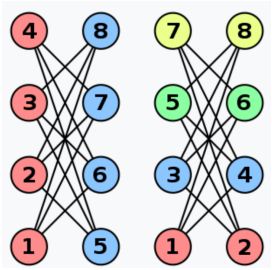

Fig. 1. Crown Graph

algorithm as well as *Greedy Sequential Algorithm* fails to give optimal results.

C. DSATUR (Degree of Saturation) Algorithm

DSatur will work similarly to greedy except it picks the next vertex more intelligently. It is the extension of the Welsh-Powell algorithm. It is expected to generate optimal results compared to the greedy sequential and Welsh-powell algorithms. This algorithm processes the vertices based on the degree of saturation. It chooses the uncolored vertices with the highest number of different adjacent colors(i.e. highest saturation degree). If two vertices are having the same degree of saturation it chooses the vertex with the highest degree even if degrees are the same it chooses the vertex order. Likewise, it overcomes the problem in the Welsh Powell algorithm where we can't choose a proper vertex if the vertices have the same degree.

Explanation: This algorithm loops through all the vertices and initializes the *color* array, which stores the final colors assigned to each vertex, the *d* array, which contains the degree of all vertices, the *adj_cols* list where *adj_col[i]* contains the set of colors used by the adjacent vertices of vertex i, and *Q*, which is a priority queue that automatically sorts the vertices based on saturation degree, then degree (if the saturation degree is the same), then vertex order (if degrees are the same). After initializing all these, we loop through the priority queue and pop the top element. Then check the colors used by these adjacent vertices and update the *use* array (which keeps track of colors used by adjacent vertices of the current vertex). Based on the data from the *use* array, get the first available color and color the current vertex. Then update the saturation degree and degree of its adjacent vertices in *Q*. The degree of the adjacent vertices is decremented because the current vertex no longer contributes to the degree of its neighbors. This ensures the correct execution of the algorithm by choosing the correct vertex in the next steps.

Correctness: The correctness of this algorithm is similar to the Welsh-Powell Algorithm except this algorithm gives even more optimal solutions than the other greedy algorithms. This algorithm overcomes the crown graph ordering issue mentioned above.

Algorithm 3: DSatur Algorithm

```
Initialize u to 0
Initialize use array of size V with False values
Initialize color array of size V with -1 values
Initialize d array of size V with degrees of vertices
Initialize adj_cols array of sets of size V
Initialize an empty priority queue Q
for u in range(V) do
    Set color[u] to -1
    Set d[u] to the degree of vertex u
    Initialize adj_cols[u] as an empty set
    Push VertexInfo(0, d[u], u) into the priority queue Q
end for
while Q is not empty do
    Pop the maximum priority element max_ptr from Q
    Set u to max_ptr.vertex
    for each color_index in range(length(use)) do
        if a color is not used then
            Set use[color_index] to True
            Break the loop
        end if
    end for
    Set color[u] to color_index
    for each adjacent vertex v of u do
        if vertex v is uncolored then
            Create a new VertexInfo object named
            new_vertex_info with parameters
            (len(adj_cols[v]), d[v], v)
            Filter out new_vertex_info from Q and update
            Q
            Add color_index to the set adj_cols[v]
            Decrease d[v] by 1
            Push VertexInfo(len(adj_cols[v]), d[v], v) into the
            priority queue Q
        end if
    end for
end while
return color
```

IV. Backtracking

Backtracking is a problem-solving technique of exploring all the possibilities to find a solution and backtracking to another path if the current does not work. We will solve the graph coloring problem using backtracking by assigning colors one by one. But before assigning colors we will check the adjacent color constraint. We will then try to find the minimum number of colors required to color the graph.

A. Simple Backtracking

In this method, we will increment the number of colors starting from 1, until it finds a pattern that can color the given graph. For finding the pattern we use backtracking. This method picks a vertex and a color and checks whether that color can be assigned to the vertex. If yes it moves to the next vertex and a color starting from the first color. If the color can't be assigned it moves to the next color. If in some cases there is no valid coloring found then it backtracks, assigns a different color,

and proceeds to do the same until it finds a valid pattern. After finding the first valid pattern this breaks and returns the minimum colors used.

Algorithm 4: Backtracking Graph Coloring Algorithm

```
Function is_valid(graph, vertex, color, assignment):
    for each neighbor in graph[vertex] do
        if neighbor is in assignment and assignment[neighbor] == color then
            return False;
        end
    end
    return True;
Function graph_coloring(graph, num_colors, assignment, vertex):
    if vertex is not in graph then
        return True;
    end
    for each color in range from 0 to num_colors - 1 do
        if is_valid (graph, vertex, color, assignment) then
            assignment[vertex] = color;
            if graph_coloring (graph, num_colors, assignment, next_vertex(graph, vertex)) then
                return True;
            end
            assignment[vertex] = None;
        end
    end
    return False;
```

Explanation: The *num_colors* variable is initially set to 1 denoting to find a graph coloring pattern with chromatic number 1. Then *graph_coloring* is called with the parameters *graph*, *assignment* array containing the colors for each vertex if a solution exists, current *vertex*, and *num_colors*. In *graph_coloring* function it loops through each color and checks the validity using *is_valid*. If valid, it assigns the color and recursively calls *graph_coloring* with the next vertex. If it did not find valid color it reassigns a null value to current vertex and backtracks to the previous step and tries different color. Likewise, it checks all the possible cases. It returns false if it is not able to find the coloring then the *graph_coloring* is again called with incremented *num_colors*.

Correctness: This algorithm takes care of single node and null node cases (check out the full code). As this algorithm traverses through whole possibilities the complexity is exponential and will not work efficiently for the graphs with a large number of nodes. As this algorithm traverses the vertices one by one it works well for isolated vertices and disconnected graphs. This gives the most optimal chromatic number compared to the previous algorithms.

B. Backtracking with Forward Checking

Forward checking is a technique of detecting the branches that do not contain a solution earlier than backtracking. This reduces the search tree. In turn, reducing the complexity of finding chromatic numbers without compromising the optimality. To explain how this works more clearly let's take an example.

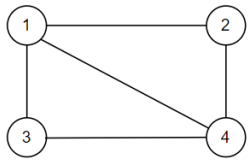

Fig. 2. Forward Checking Example

In this method, we declare the domain for each vertex. Vertices can only take values from these domains.

$V = \{1, 2, 3, 4\}$ All vertices

$D = \{\text{Red}, \text{Green}, \text{Blue}\}$ Total domain value set

$C = \{1 \neq 2, 1 \neq 3, 1 \neq 4, 2 \neq 4, 3 \neq 4\}$ Constraints

TABLE I
INITIAL DOMAIN ASSIGNMENT

Vertex	**1**	**2**	**3**	**4**
Initial Domain	R,G,B	R,G,B	R,G,B	R,G,B
1 = R	R	G,B	G,B	G,B
2 = G	R	G	G,B	B
3 = B	R	G	B	B

The first vertex is assigned with Red, and for the rest of the vertices, Domains are updated. And the same while vertex 2 is assigned with Green. But when I want to update Green to vertex 3 there is no legal color left for vertex 4. Hence this step will be backtracked. And vertex 3 will be assigned to another color. The difference between backtracking and this method is that In Backtracking domains are not considered and vertex 3 will take R, G then B which increases the search space.

Explanation: The *state* variable contains the vertex numbers and helps in changing other variables based on a particular state. *neighbours*[i] contains the neighbors of the ith vertex, *colors*[i] contains the colors added to the ith vertex, and *domain*[i] contains the set of domain values, only from which the ith vertex's color can be assigned. After initializing all these variables, Forward Checking loops through the domain of the current vertex and assigns a color, updates the state. Then checks whether the domain for neighbors can be reduced, and if yes, then it calls its recursion for its neighbors. If the domain for adjacent vertices is empty, then it removes the assignment of current colors and goes to the next color and continues the process.

Correctness: The correctness of this algorithm is similar to the backtracking approach. This algorithm just reduces the state space so the only difference comes from complexity but optimality remains the same.

Algorithm 5: Forwardcheck Algorithm

```
Function Forwardcheck(states, neighbours, colors, domain):
    if all values in colors are not equal to 'Nil' then
        return "Success";
    end
    currentState ← states[0];
    currentNeighbors ← neighbours[currentState];
    occupiedColors ← map colors.get for each neighbor in currentNeighbors;
    if 'Nil' is in occupiedColors then
        Remove 'Nil' from occupiedColors;
    end
    for each color in domain[currentState] do
        if color is not in occupiedColors then
            Assign color to currentState;
            Remove currentState from states;
            if Forwardcheck(states, neighbours, colors, domain) is not equal to "Failure" then
                return "Success";
            end
            Restore currentState to states;
            Unassign color from currentState;
        end
    end
    if currentState's color is still 'Nil' then
        return "Failure";
    end
```

C. Constraint Satisfaction Problem

Constraint satisfaction problem is an extension of forward checking. This involves heuristics in decision-making for the path to be chosen. In simple words, It decides which vertex to choose next intelligently based on some metrics which ultimately tends to provide optimal solutions. Here we choose the variable with the most constraints on the remaining variables.

To be more clear, consider an example below

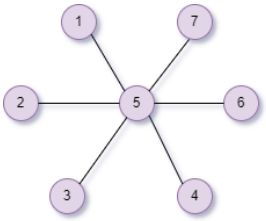

Fig. 3. Star Graph

Forward Checking is applied to this graph with a color set $\{\text{Red}, \text{Green}\}$. It starts with node 1 and assigns Red so for other nodes except for node 5 domain will be $\{\text{Red}, \text{Green}\}$. So there will be $2^6 + 1 = 65$ possibilities.

But if we initially start with node 5 and assign Red color then for all other nodes the domain changes to Green. Here we can directly assign a Green color to other nodes. Selecting the node that needs to be explored next intelligently is reducing the search space and even solutions can be found within less time.

Algorithm 6: CSP

```
Function Forwardcheck(states, neighbours, colors, domain):
    if all colors are assigned then
        return "Success";
    end
    currentState ← selectVariable(states, domain, neighbours);
    for each color in orderDomain(currentState, domain, neighbours) do
        if color not in getOccupiedColors(neighbours[currentState], colors) then
            assign color to currentState;
            remove currentState from states;
            result ← checkConstraints(color, neighbours[currentState], colors, domain);
            if not result then
                prevDomain ← copy of domain;
                reduceDomain(color, neighbours[currentState], colors, domain);
                singleton ← reduceSingletonDomain(neighbours[currentState], neighbours, colors, domain);
                if singleton then
                    recursively call ForwardcheckWithSingletonPropogationAndHeuristics;
                end
                restore domain if failure;
                undo assignment of color;
                add currentState back to states;
            end
        end
    end
    if currentState is still unassigned then
        return "Failure";
    end
```

Explanation: All the variables used here have similar functions to that of forward checking. After initializing all the variables, instead of directly looping, it calls the 'leastConstrainingValueHeuristic' function to select a vertex. Then, similar to Forward Checking, it loops through the domain, assigns colors, backtracks and re-assigns colors. But the extra thing here is it calls the 'reduceSingletonDomain' function to further reduce the search space.

Correctness: This algorithm assigns correct colors satisfying the constraints. It works well for all types of graphs.

V. Experimentation

The experimentation code to analyze the algorithms and algorithmic implementation code is in the following repository:
https://github.com/TejaswiBattula/Graph-Coloring-Algorithms-Analysis

Basic Code Walk-through

To obtain all experimentation results, it is necessary to run `start.py`. After the algorithms are executed, the results are sent to `plot.py` and corresponding plots are generated. These plots will be analyzed further. If you need to test a specific algorithm separately, you can use `test.py`. Specify the type of input graph and the algorithm you want to run. This script will provide a visual representation of the colored output graph, which is retrieved from `output.py`. `input.py` generates random inputs with different types of graphs, facilitating experimentation with various graph structures.

Configuration And Setup

1) Required version: Python version 3.11.5
2) For dependencies, check `requirements.txt` in the specified repository.
3) For setup, check the README file from the repository.
4) For testing individual algorithms, check `test.py`.

How is experimentation conducted?

Certain input graph types are taken into consideration. For each of the graph types, all the algorithms are executed. Time taken by these algorithms and the chromatic number are stored and plotted as line graphs (duration vs input, chromatic number vs input). As the backtracking algorithms take so much time compared to greedy algorithms, plotting them together won't give better results because the lines on the graph for greedy will overlap. Hence we are separately considering greedy algorithms and backtracking algorithms after analyzing them. We run all algorithms together with less input set and can compare them thoroughly. While running the input types we vary the size because some graphs like linear graphs take less time so we can take large input to properly analyze them while graphs like dense graphs take a lot of time executing hence we take relatively smaller inputs.

Strategies for Ensuring Uncorrupted Results

Experimentation is conducted on gcloud virtual machines which are specifically started to run the analysis. These are isolated environments without background processes. Hence, these help in giving correct results. Input size is changed according to the algorithm so that the algorithms with less complexity take more input size and vice versa. Monitoring system resources throughout the process.

Input Types - Consideration Factors

The input graph types for graph coloring are unweighted, undirected graphs. Hence the following input graph types are selected.

Linear graph: Linear graphs represent simple structures. This helps in demonstrating the efficiency of algorithms in handling linear structures.

Dense graph: Dense graphs have a high density of edges and vertices. Analyzing the algorithms on dense graphs is important because they challenge the efficiency of algorithms and help in finding the algorithms that are giving optimal results as there will be a large number of conflicts while coloring the graph.

Sparse Graph: These have a low density of edges analyzing these will help us understand how algorithms are performing in situations where there are fewer potential conflicts.

Regular Graph: Regular Graphs have the same degree for all vertices. These are useful in analyzing because they provide a uniform distribution of constraints and help to compare the algorithms.

Planar Graph: Planar Graphs are graphs that can be drawn on a plane. It does not have crossing edges. There is a four-color theorem that states that a planar graph can take the utmost 4 colors. Hence the chromatic number can not exceed 4. Taking this graph we can also check whether the algorithms are satisfying this requirement.

Disconnected Graph: Disconnected graphs contain multiple disconnected components. Analyzing these helps in observing how algorithms are handling separate regions with different coloring constraints.zz

Graphs with bridges and cut vertices: These are the best graphs to check which algorithms are giving optimal coloring. Also, this helps in comparing algorithms like DSatur and CSP with others as these algorithms have heuristics in selecting the vertices intelligently.

Crown graph: The crown graph is a complete bipartite graph with horizontal edges removed. As mentioned in greedy algorithms the efficiency of the algorithm changes based on the ordering of vertices. This graph helps to strengthen the fact that chromatic numbers change based on the ordering.

Analysis Metrics

For graph coloring, we must consider completeness and optimality, thus necessitating the evaluation of:

- Time complexity
- Space complexity
- Chromatic number

We analyze both Input vs. time complexity and Input vs. chromatic number for all the algorithms. Both of these metrics are equally important, as sometimes the solution needs to be produced instantly, in which case algorithms with minimum complexity are preferable. Other times, we require the best solution possible, and thus, we must analyze the tradeoffs.

Theoretical complexity

Number of vertices V
Number of edges E
Chromatic number C

TABLE II
TIME COMPLEXITY - GREEDY

Algorithm	Greedy Sequential	Welsh-Powell	DSatur
Worst Time Complexity	$O(V^2)$	$O(V \log V)$	$O((V + E) \log V)$
Space Complexity	$O(V)$	$O(V)$	$O(V \log V)$

TABLE III
TIME COMPLEXITY - BACKTRACKING

Algorithm	Backtracking	Forward Checking	CSP
Worst Time Complexity	$O(C^V)$	$O(C^V)$	$O(C^V)$
Space Complexity	$O(V)$	$O(V)$	$O(V)$

Theoretical Optimality

Optimality depends on the type of graph but this is given as a general case. Ordering the algorithms based on the optimal result (minimum chromatic number) generation.

Backtracking $>$ CSP $>$ DSatur $>$ Forward-Checking $>$ Welsh-Powell $>$ Greedy Sequential

VI. OBSERVATIONS AND RESULTS

From the experimentation, we will analyze the graphs to derive meaningful insights. Our objective is to analyze the performance of the mentioned algorithms and identify patterns in their behavior across the mentioned input types.

Greedy Algorithms - Analysis

Dense And Sparse Graphs Analysis:

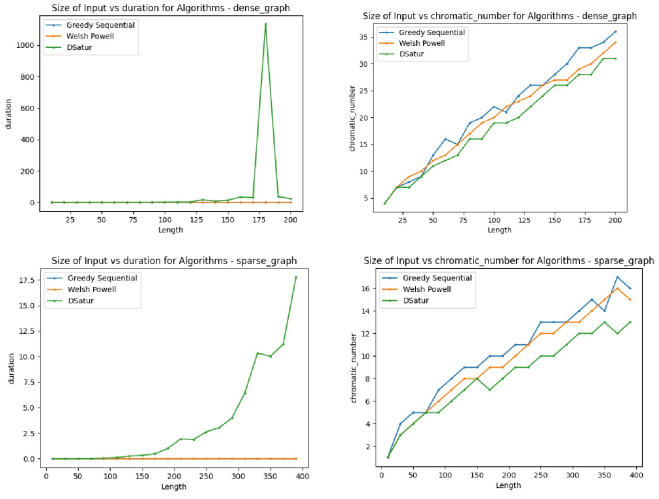

Fig. 4. Dense And Sparse Graphs Analysis

For dense and sparse graphs the algorithms are behaving normally. In both cases greedy sequential produced greater chromatic numbers compared to others and DSatur produced comparatively optimal chromatic numbers. But DSatur has taken so much execution time. Although the time taken increased gradually in the sparse graph, in the dense graph there is unexpected rise and fall which determines that the complexity for DSatur purely depends on the kind of the graph.

Graph with Bridges And Regular Graphs Analysis:

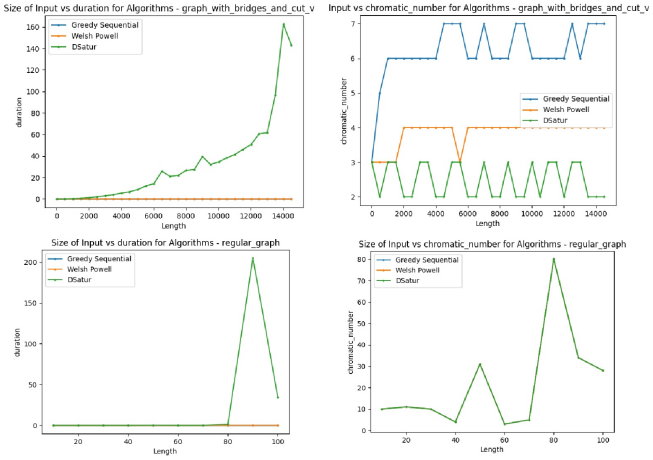

Fig. 5. Graph with Bridges And Regular Graphs Analysis

The time complexity for the algorithms is similar to the previous analysis, DSatur for regular graphs even performed similarly to that of dense graphs, this again proves the dependency of the algorithm on the input case. In the graph with bridges and cut vertices DSatur gave optimal results followed by Welsh-Powell. This is expected as DSatur selects the vertices based on heuristics and it performed well for graphs with bridges. Interestingly, for regular graphs, all the algorithms gave the same results.

Disconnected And Crown Graphs Analysis:

The time complexity for the disconnected graph is similar to previous graphs. Here major insight is from the input vs chromatic number graph where unexpectedly Dsatur Algorithm has produced greater chromatic numbers. The Welsh-Powell Algorithm for disconnected graphs gave the most optimal result. It is interesting to note that even Greedy Sequential gave more optimal results than DSatur. From this, we can conclude DSatur is not a better algorithm for disconnected graphs. For the crown graph, the time complexity for DSatur increased exponentially and the other two algorithms did not take much time to execute. The chromatic number produced by all the algorithms are same as the ordering of the crown graph is in such a way that all the algorithms produce the same results.

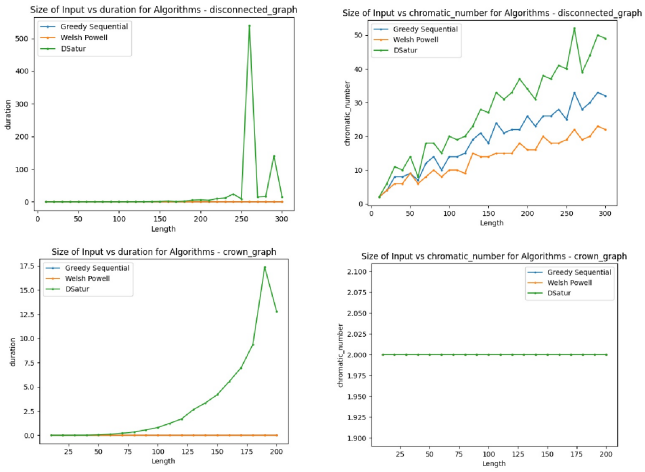

Fig. 6. Disconnected And Crown Graphs Analysis

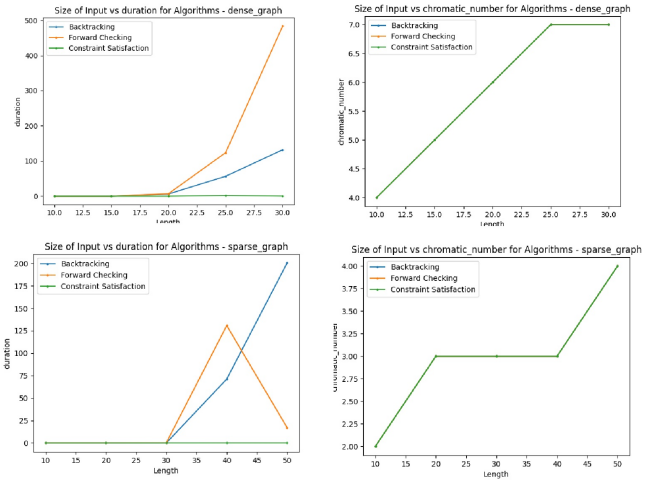

Fig. 8. Dense And Sparse Graphs Analysis

Backtracking Algorithms - Analysis

Linear And Disconnected Graphs Analysis:

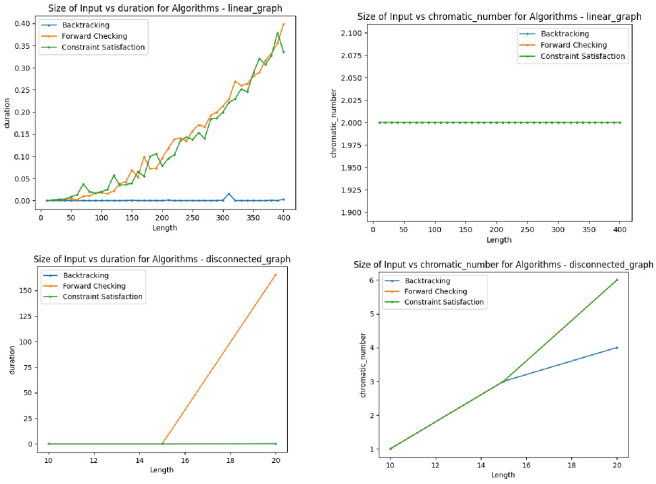

Fig. 7. Linear And Disconnected Graphs Analysis

For the linear graph, backtracking is taking linear time. This is expected behavior as it finds a solution following the first path without backtracking. Forward Checking and CSP have taken almost the same time, the majority of this time is contributed by calculating heuristics rather than that of by backtracking. All algorithms gave the same chromatic number. For the Disconnected graph, Forward Checking exhibited unexpected behavior and has taken longer than expected. Backtracking gave optimal results compared to other algorithms.

Dense And Sparse Graphs Analysis: As these algorithms take exponential time, we have only taken less input size. It is hard to analyze the optimality and derive correct results. For both dense and sparse graphs, the algorithms have given the same chromatic number. We can not judge optimality here. Coming to complexity, forward checking in a dense graph has taken a longer time, followed by backtracking. Insightfully, CSP has taken polynomial time. Similarly, for sparse graphs, backtracking and forward checking have taken longer, but it depends on the input type.

Combined Analysis

Linear And Planar Graphs Analysis:

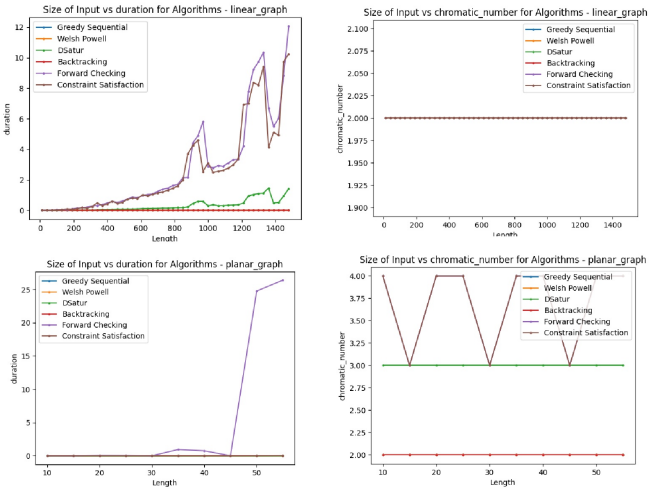

Fig. 9. Linear And Planar Graphs Analysis

For linear graphs, Forward checking is taking longer time followed by CSP. Backtracking has taken the least time. In backtracking, for the first possible path, the solution was found in a linear graph, hence the time taken is lower. All the other algorithms which do not follow any heuristics have taken less time. The chromatic number is the same for all algorithms. In planar graphs, surprisingly Forward Checking has taken a very long time. It might be because this is

stuck with checking domains are backtracking. For chromatic numbers, although some algorithm's complexity is not visible we can see the pattern for CSP. Backtracking and DSatur have produced constant results. But this graph proves that the planar graph's chromatic number won't exceed 4.

Dense And Sparse Graphs Analysis:

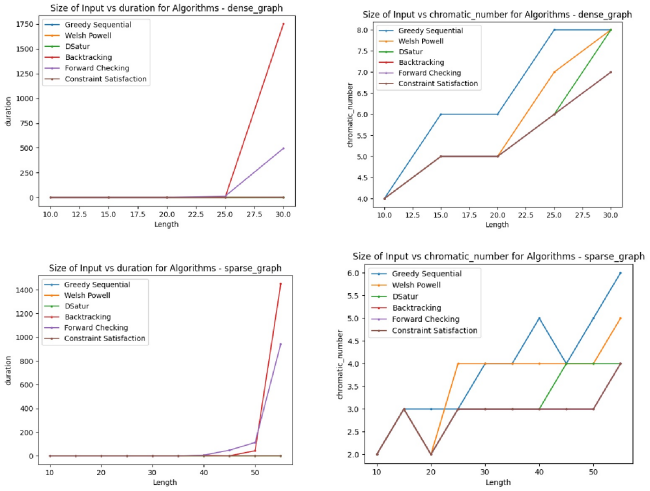

Fig. 10. Dense And Sparse Graphs Analysis

In the Dense graph, the duration of backtracking is very high. This shows the actual exponential growth of time complexity for backtracking. Forward checking follows the backtracking. The rest of the algorithms have taken less time. This is expected behavior. The graph for chromatic numbers is little overlapped, but based on the visible graph, Greedy Sequential gives greater chromatic numbers. CSP produces the optimal results. Algorithms on the Sparse graph behave similar to the dense graph. As usual, backtracking has taken a longer time, followed by forward checking. And CSP has produced optimal results. We can conclude that using CSP for dense graphs and sparse graphs will be efficient as it gives optimal results with less execution time. Followed by DSatur.

Graph with bridges And disconnected graph Analysis:

In graphs with bridges and cut edges, Backtracking has taken a longer time. Welsh-Powell produced a constant chromatic number for all the inputs. CSP showed optimal results. CSP is better for this graph because it produces optimal results in less time. While analyzing backtracking algorithms previously, Forward Checking has taken a longer time for graphs with bridges, even here its execution time is longer. It proves that Forward Checking is not a suitable algorithm for disconnected graphs. Even though it's time complexity is more it failed to produce optimal results. Welsh-Powell gave optimal results while analyzing disconnected graphs for Greedy algorithms and here it is also producing optimal results. This proves that Welsh-Powell is a better algorithm

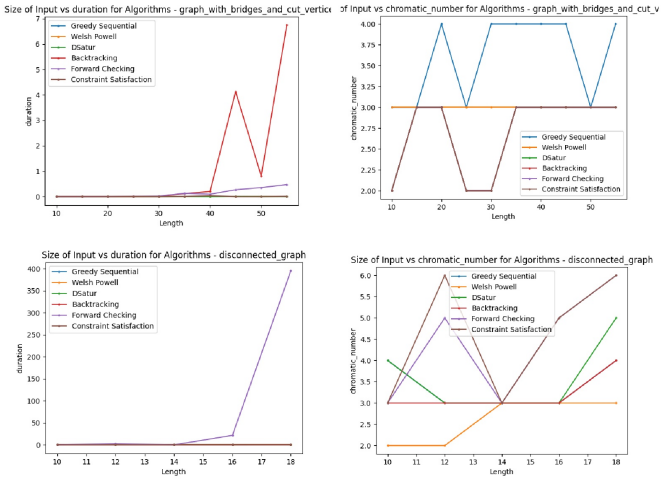

Fig. 11. Graph with bridges And disconnected graph Analysis

for disconnected graphs.

Planar and Regular Graphs Analysis:

For the planar graph we can not analyze properly because of the overlapping in the chromatic numbers and Forward Checking is taking a lot of time. CSP does not produce optimal results. So, here Forward Checking and CSP are not taken into consideration and the rest of the algorithms are analyzed. Similarly, for regular graphs, Forward Checking is taking more time than expected. Hence, from regular graphs also Forward Checking is removed for analysis. At the same time From Greedy Algorithms analysis. All the greedy Algorithms are producing the same results but DSatur is comparatively taking more time. Hence DSatur is excluded from the analysis.

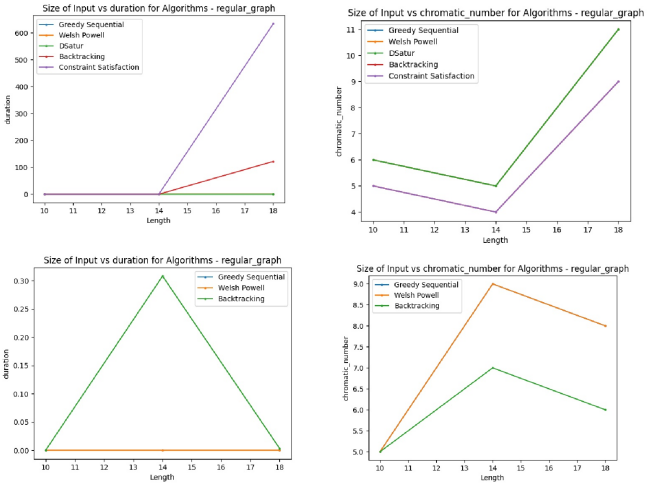

Fig. 12. Regular graph Analysis

From the graphs, we can see CSP is producing optimal results but at the same time taking longer execution time. Even Backtracking provides similar results but takes comparatively

less time. In the second set of graphs, it can be noticed that Backtracking provides optimal results but takes longer time than greedy algorithms. Hence the best algorithm can not be determined. If Backtracking is considered time will be the tradeoff. If Greedy Sequential or Welsh-Powell is considered optimality will be the tradeoff.

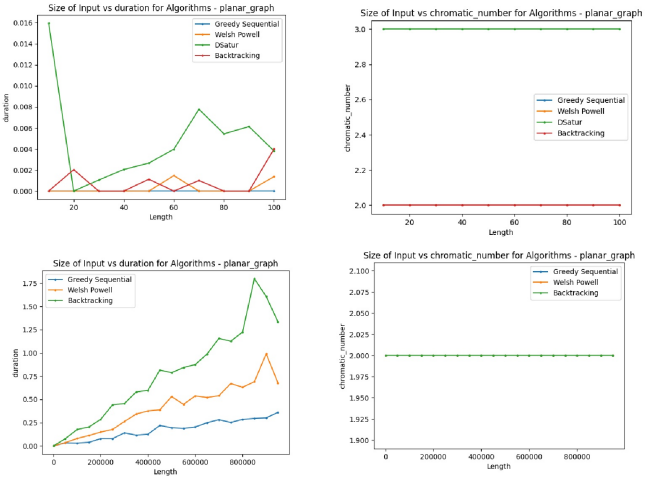

Fig. 13. Planar graph Analysis

The first sample graph is plotted to check which algorithms are taking longer time. Found that DSatur is taking more time and also not giving optimal results. Hence DSatur is removed and analysis is continued with the rest of the algorithms. From the graph we can see although all the algorithms are producing the same results, Greedy Sequential is taking less time to execute. Hence we can conclude that the Greedy Sequential Algorithm is the best algorithm for planar graphs.

VII. Conclusion

This provides a summary and insights to choose the most suitable algorithm based on the resources and problem requirements.

For dense and sparse graphs CSP exhibited better performance followed by DSatur. CSP provided optimal results with less time complexity. This can be chosen for dense and sparse graph types.

Although it is known that for linear graphs chromatic number is not more than 2. Backtracking solved this within less time. Hence we can state that Backtracking is a better solution for linear graph types. Backtracking with the combination of some other algorithms can help in solving problems with linear graph structure combined with other graph types.

For the graphs with bridges and cut vertices, CSP provides optimal solutions within less time as it uses degree heuristics to choose the vertices. It chooses to color vertices within bridges first.

Interestingly, Welsh-Powell is the best algorithm for disconnected graphs. As it sorts the vertices and goes on coloring. Coloring for all the isolated vertices is done in very little time. Whereas, other algorithms take time in calculating the heuristics and choose which path to take.

Because of the ordering of the Crown graph all the algorithms performed the same. If the ordering is horizontal then DSatur performs better.

For regular graphs, Backtracking gives an optimal solution but takes a longer time. Greedy Sequential and Welsh-Powell algorithms are faster but do not give comparatively optimal solutions. Hence these need to be chosen based on the requirements.

For planar graphs, Greedy Sequential algorithm is the best as it is producing fewer chromatic numbers with less time.

Selecting the right graph depends on the requirements. This analysis helps to identify a suitable algorithm.

In conclusion, this paper provides a thorough exploration of graph coloring algorithms and their performance. This comprehensive analysis provides valuable insights to graph theory and algorithmic optimization.

References

[1] ”What are some alternative methods that could be used to solve the coloring problem?,” Quora, [Online]. Available: https://www.quora.com/What-are-some-alternative-methods-that-could-be-used-to-solve-the-coloring-problem.

[2] ”An exact approach for the Vertex Coloring Problem,” *ScienceDirect*, 2010, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S157252861000054X.

[3] T. Ballard and S. Myer, ”A Simulated Annealing Approach to Graph Coloring,” 1995, [Online]. Available: https://www.gcsu.edu/sites/files/page-assets/node-808/attachments/ballardmyer.pdf.

[4] ”Graph coloring,” Wikipedia, Wikimedia Foundation, January 22, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Graph_coloring.

[5] ”DSatur Algorithm for Graph Coloring,” *GeeksforGeeks*, [Online]. Available: https://www.geeksforgeeks.org/dsatur-algorithm-for-graph-coloring/.

[6] ”Vertex Coloring Welsh-Powell Algorithm,” 2009, [Online]. Available:

[7] ”Welsh Powell Graph colouring Algorithm,” *GeeksforGeeks*, [Online]. Available: https://www.geeksforgeeks.org/welsh-powell-graph-colouring-algorithm/.

[8] ”CSP Mapping Color,” [Online]. Available: https://github.com/gadgil-devashri/csp-map-coloring/blob/main/code/mapcoloring.py.