Greedy vs Backtracking

A Comparative Study of Graph Vertex Coloring Algorithms with C++ Implementations

Jonathan Kris Wicaksono - 13524023
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: jonathankw2005@gmail.com,13524023@std.stei.itb.ac.id

***Abstract*—Graph vertex coloring is a classic problem in graph theory with numerous applications in computer science, scheduling, and resource management. This paper provides a comparative study between greedy and backtracking approaches for vertex coloring, highlighting their algorithmic principles, complexity, and practical performance. Through detailed implementation and experimentation, we analyze how each method performs on various graph instances, illustrating the trade-offs between computational efficiency and solution optimality. Our findings offer valuable insights for selecting appropriate algorithms based on problem requirements.**

***Keywords*—Graph coloring, Partition problem, Backtracking, Basic greedy, DSatur.**

I. INTRODUCTION

At its core, the graph vertex coloring problem seems quite straightforward: given a graph, how can we assign colors to its vertices so that (a) no two adjacent vertices share the same color, and (b) we use as few colors as possible? Even though the idea is simple, the problem itself is rich in both theory and real-world relevance. Its origins trace back to the well-known Four Color Theorem. Since then, graph coloring has become a fundamental topic in graph theory. Vertex coloring appears in many real-world situations, especially in problems where resources need to be assigned or organized efficiently without conflicts. Among the many algorithmic approaches developed to tackle vertex coloring, two broad paradigms stand out: greedy heuristics and backtracking-based exact methods. These approaches differ greatly in design philosophy, efficiency, and the quality of solutions they produce.

In the sections that follow, we will first formalize the definition of graph vertex coloring and review foundational concepts from elementary graph theory to ensure a shared baseline of understanding. We will then shift our focus to a comparative analysis of two major algorithmic paradigms used to solve the problem: greedy methods and backtracking-based approaches. We examine several representative algorithms—Basic Greedy and DSatur for the greedy category, alongside a standard backtracking algorithm for exact coloring—evaluating their computational complexity, implementation characteristics, and practical trade-offs. Through this comparison, we aim to clarify the strengths and limitations of each approach and provide insight into their behavior under different types of graph instances. In this paper, I aim to provide clarity and implementation that can be used right away. Thus, I will first talk about an algorithm in high-level pseudocode, and from there, I will implement the algorithm in C++.

II. THEORETICAL PREREQUISITE & BASIC TERMINOLOGY

A. *Elementary Graph Terminology*

Before diving straight to the heart of graph coloring, we will briefly review some elementary graph terminology that is necessary for understanding the content of this paper. Recall that a graph can be thought of as a collection of vertices along with a rule that tells us which pairs of vertices are connected. More formally, we define a graph G as an ordered pair (V, E), where V is the set of vertices and E is the set of edges that describe connections between those vertices. In general, we write $E \subseteq \{(u, v) : u, v \in V\}$ to represent this. Now, consider a graph that contains loops (a loop is an edge such that it connects a vertex to itself); this will automatically make our coloring problem on this graph impossible since there exists an edge $e = (u, u)$ (a loop) which violates the fundamental rule of proper vertex coloring (this will be defined below). In the vertex coloring case, multiple edges will not actually fail our coloring problem, but this will be problematic in the case of edge coloring. Thus, from now on when we mention a graph G, we will assume it's a simple graph unless stated otherwise. Formally, a simple graph is an undirected graph in which loops and multiple edges between vertices are forbidden. Consequently, each element of E in a simple graph is written as an unordered pair $\{u, v\}$ indicating the existence of an undirected edge between u and v with $u \neq v$ and $u, v \in V$.

A graph that contains a cycle can also help us understand how to determine a proper coloring. To define what a cycle is, we must first understand the concept of a path in graph theory. A *path* is a non-empty graph $P = (V, E)$ of the form

$$\begin{aligned} V &= \{x_0, x_1, \ldots, x_k\}, \\ E &= \{\{x_0, x_1\}, \{x_1, x_2\}, \ldots, \{x_{k-1}, x_k\}\}, \end{aligned}$$

where the vertices x_i are all distinct. Now, suppose P is such a path and $k \geq 2$, then we can form a cycle by adding an edge that connects the last vertex back to the first. That is, the graph $G := (V, E \cup \{\{x_k, x_0\}\})$ is called a cycle.

B. Graph Coloring

The graph coloring problem can be defined more formally as follows. Given a graph $G = (V, E)$, the goal is to assign each vertex $v \in V$ a color $c(v) \in C = \{1, 2, \ldots, k\}$ such that (a) for every edge $\{u, v\} \in E$, the colors of its endpoints are different, i.e., $c(u) \neq c(v)$ (no two adjacent vertices share the same color), and (b) the total number of colors k used is minimized. We define a *clash* as a situation in which a pair of adjacent vertices $u, v \in V$ are assigned the same color, i.e., $\exists\{u, v\} \in E$ such that $c(u) = c(v)$. A coloring is said to be *proper* if it contains no clashes; otherwise, it is *improper*. Unless stated otherwise, the term "coloring" in this paper will always refer to a proper coloring. Finally, we say that a graph G is k*-colorable* if there exists a proper coloring of G using at most k distinct colors. A k*-coloring* refers to a proper coloring that uses exactly k colors.

In the graph coloring problem, we can also seek out an equivalent problem (this is a common practice in solving hard problems, since looking at a problem from a different perspective may give us new insights). Observe that coloring a graph in a proper way is equivalent to a partitioning problem. A proper k-coloring solution is a class (collection of sets) $\mathcal{S}$ containing k colors, $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$, where the following properties must be obeyed by $\mathcal{S}$ to ensure a proper coloring:

i. the union of all sets in $\mathcal{S}$ must be equal to V:

$$\bigcup_{i=1}^{k} S_i = V,$$

ii. the sets in $\mathcal{S}$ must be pairwise disjoint:

$$\forall i, j \in \{1, 2, \ldots, k\},\ i \neq j, \quad S_i \cap S_j = \emptyset,$$

iii. adjacent vertices must be assigned to different sets:

$$\forall i \in \{1, 2, \ldots, k\},\ \forall u, v \in S_i, \quad \{u, v\} \notin E,$$

with k being minimized. This way of viewing the graph coloring problem as a partitioning problem will be of particular importance in the upcoming section.

C. Chromatic Number

We shall define the set of available colors of a coloring as $C = \{1, 2, \ldots, k\}$. We are not interested in the elements of C but rather in its size (since we are trying to minimize it). We define the chromatic number of a graph G as the smallest k such that G has a k-coloring. We denote the chromatic number of G as $\chi(G) = k$.

To make the theory that has been laid out so far more concrete, let us take a look at fig. 1.

With simple observation, it is obvious that $\chi(G) = 3$ for this particular graph since a 2-coloring is impossible. This follows from the structure of the graph, which contains odd cycles (cycles with an odd amount of vertices) that prevent a 2-coloring (also known as bipartite coloring). The coloring shown assigns one of three colors to each vertex such that no adjacent vertices share the same color, satisfying the requirements of a proper coloring.

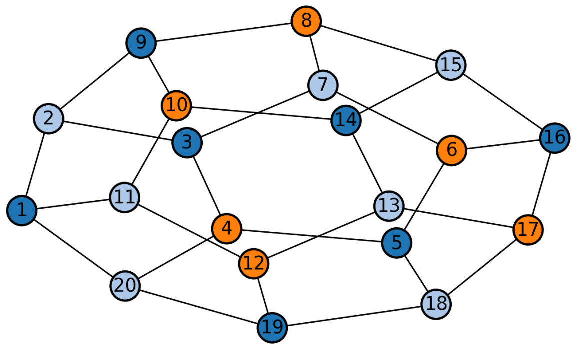

Fig. 1. Proper 3-coloring on a dodecahedron

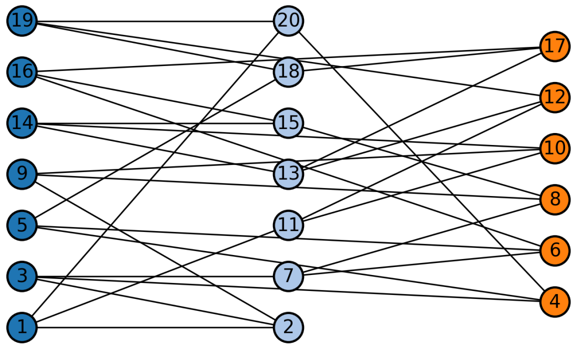

Fig. 2. Partitioning of fig. 1

We can also view the above graph as a partition of 20 vertices. This is shown in fig. 2. This figure is easier for checking proper coloring in a graph since it's just a partition of the graph into 3 columns. Reassembling a graph into a partition is generally easier for us to check whether a graph is properly colored or not.

III. COMPUTATIONAL COMPLEXITY OF GRAPH COLORING

A. How Hard Is It to Color a Graph?

At first glance, the graph coloring problem seems straightforward. Try assigning colors to each vertex and check whether the coloring is proper. However, as the size of the graph increases, the number of possible colorings grows extremely fast. For example, even with only three colors, a graph with n vertices has 3^n possible assignments to consider. In general, a brute-force approach considering all possible k-colorings from $k = 1$ to $k = n$ is at worst $\mathcal{O}(n^n)$. This can be reduced with lots of optimization and observation, but the time complexity in general is still exponential. Due to the discussion in [1], we can make a faster algorithm (although still exponential) by observing that an arrangement such as $c(1) = 1$, $c(2) = 2$, $c(3) = 1$, and $c(4) = 2$ is equivalent to $c(1) = 2$, $c(2) = 1$, $c(3) = 2$, and $c(4) = 1$. We say that such an arrangement is equivalent because, if we view the problem as a partitioning problem, the resulting solution class $\mathcal{S}$ remains the same (in this case, $\mathcal{S} = \{\{1, 3\}, \{2, 4\}\}$). Thus, we can reduce the complexity of our problem by observing that the problem is equivalent to finding the number of ways to partition the set of vertices V into non-empty, pairwise disjoint subsets in which their unions should be V. Recall that this is just the *Bell Number*, B_n, where n is the number of vertices. Remember that the Bell numbers are related to the Stirling numbers of the second kind $\left\{ {n \atop k} \right\}$ by

$$\begin{aligned} B_n &= \sum_{k=1}^{m} \left\{ {n \atop k} \right\} \\ &= \sum_{k=1}^{n} \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n. \end{aligned}$$

We can therefore employ the following algorithm: for each color starting from $k = 1$, we can check for all possible $\left\{ {n \atop k} \right\}$ colorings and check if they are proper. If a proper coloring is found, then we can halt the process immediately since this is obviously the minimum k. Unfortunately, such an algorithm is still at worst $\mathcal{O}(n^n)$. Therefore, trying all of them quickly becomes infeasible for larger graphs.

To simplify our problem, we may consider the decision version of graph coloring: given a graph G and an integer k, does there exist a proper coloring of G using at most k colors? While this version sounds simple, it turns out to be very difficult to solve in general.

In computer science, this problem is known to be *NP-complete*, meaning that no efficient (polynomial-time) algorithm is known, and it is widely believed that none exists. In other words, even powerful computers cannot solve arbitrary instances of the coloring problem quickly. This makes graph coloring one of the fundamental problems in the theory of computational complexity.

B. Approximation and Heuristics

Because finding the chromatic number is so difficult, researchers often focus on algorithms that give *good enough* colorings rather than optimal ones. These are known as *heuristic algorithms*. While they do not always produce the smallest possible number of colors, they can often find a proper coloring efficiently in practice.

In the next section, we will explore two well-known heuristic algorithms for graph coloring, which are the Basic Greedy Algorithm and DSatur. After that, we will do a comparative study for each algorithm on a set of graph instances. This will help us find out which algorithm we should use on specific graph instances.

IV. Graph Coloring Algorithms

A. Backtracking Approach

Backtracking is a classic brute-force paradigm in which we try every possible configuration in a solution space using a decision tree. This algorithm is suitable for problems that require us to look at all possible solutions (exhaustive search). For example, generating the powerset of a set, finding all possible permutations of a set of objects, solving sudoku puzzles, and many more. Although backtracking belongs to brute-force strategies, we don't refer to it as a brute-force algorithm in the context of graph coloring, since it behaves quite differently. A classic brute-force approach will take us $\mathcal{O}((V + E)k^V)$, but a backtracking approach will give a slightly better complexity of $\mathcal{O}(Vk^V)$, where V is the number of vertices in the graph, E is the number of edges in the graph, and k is the number of available colors [6]. This is because there are $\mathcal{O}(k^V)$ possible combinations of coloring and at worst ISSAFE will be $\mathcal{O}(V)$. The pseudocode for backtracking approach in graph coloring problem is provided in algorithm 1.

Algorithm 1 Graph Coloring via Backtracking

```
procedure GRAPH-COLORING($G = (V, E)$, $k$)
    for all $v \in V$ do
        $color[v] \leftarrow 0$                ▷ 0 means uncolored
    end for
    if COLOR-VERTEX(1, $G$, $color$, $k$) then
        return $color$
    else
        return "No valid coloring"
    end if
end procedure
procedure COLOR-VERTEX($v$, $G$, $color$, $k$)
    if $v > |V|$ then
        return true
    end if
    for $c \leftarrow 1$ to $k$ do
        if ISSAFE($v$, $c$, $G$, $color$) then
            $color[v] \leftarrow c$
            if COLOR-VERTEX($v + 1$, $G$, $color$, $k$) then
                return true
            end if
            $color[v] \leftarrow 0$            ▷ Backtrack
        end if
    end for
    return false
end procedure
procedure ISSAFE($v$, $c$, $G$, $color$)
    for all $u \in \text{Adj}[v]$ do
        if $color[u] = c$ then
            return false
        end if
    end for
    return true
end procedure
```

Here is an implementation of algorithm 1: (**Note.** For clarity and consistency with the pseudocode, this implementation uses 1-based indexing, where vertices are labeled from 1 to n.)

```cpp
#include <iostream>
#include <vector>

using namespace std;

// This implementation uses adjacency list
    representation
using Graph = vector<vector<int>>;

bool isSafe(int v, int c, const Graph &G, const
    vector<int> &color) {
   for (int u : G[v]) {
      if (color[u] == c)
         return false;
   }
   return true;
}

bool colorVertex(int v, const Graph &G, vector<int>
    &color, int k, int n) {
   if (v > n)
      return true;

   for (int c = 1; c <= k; ++c) {
      if (isSafe(v, c, G, color)) {
         color[v] = c;
         if (colorVertex(v + 1, G, color, k, n)) {
            return true;
         }
         color[v] = 0; // Backtrack
      }
   }
   return false;
}

vector<int> backtrackGraphColoring(const Graph &G,
    int k, int n) {
   vector<int> color(n + 1, 0); // 1-based indexing

   if (colorVertex(1, G, color, k, n)) {
      return color;
   } else {
      return {}; // No valid coloring
   }
}
```

Let us analyze this backtracking approach and show why this algorithm is guaranteed to find a proper coloring with k colors if one exists, and how it can be used to search for the chromatic number by incrementally increasing k. First note that the GRAPH-COLORING procedure takes a graph G and a number k as its parameter to check whether we can color G using at most k colors. We then must initialize the color array of G's vertices with 0 as a mark that these vertices are not colored yet. Next, we call the COLOR-VERTEX starting from vertex 1 which will then return a boolean value. If it's colorable using at most k colors, then we will return such coloring which is the *color* array. Else, we will return a message that no valid coloring is available.

Now, inside the COLOR-VERTEX procedure is where our main algorithms lie. First, we will take 4 parameter which is v (the vertex we are interested in), G, *color* (the array of colors for each vertex), and k. Next, we see that if the vertex v is already out of bounds, then we can safely return true—which means that we have successfully colored every vertex in our graph. If that is not the case, we still have some more vertices to color. We will then try to color our current v with color starting from 1 up to k. If a color c is safe (there are no neighbors of v that are colored with c), we can then try to color it using c and then move on to the next vertex by calling COLOR-VERTEX upon $v + 1$. If we get a true value, we can then return true; else, we try another color for the current vertex v (backtracking).

This algorithm gives us a valid coloring of G since we are trying recursively to find a valid coloring starting from vertex 1. We can also be sure that this coloring is the minimum coloring since we try to color every vertex starting from color $c = 1$. Hence, we have a correct algorithm for coloring a graph. But by our argument from the previous section, this algorithm is too slow in practice. Therefore, we consider heuristic greedy algorithms, which may not always give us the chromatic number of a graph but still produce valid colorings. In many practical scenarios, such approximations are more useful than exact but computationally expensive solutions like backtracking.

B. *Basic Greedy Algorithm*

We begin our discussion on greedy coloring heuristics with the Basic Greedy algorithm. Although simple, it serves as a fundamental approximation method for graph coloring.

The idea is straightforward: traverse the vertices one by one (in any fixed order), and for each vertex v, assign the smallest color label j that has not been assigned to any of its adjacent vertices. That is, $\mathit{color}[v] \leftarrow j$, where j is the smallest available color among v's neighbors. If all previously used colors are already taken by neighbors, a new color is assigned. The pseudocode for Basic Greedy algorithm is provided in algorithm 2 below.

Algorithm 2 Basic Greedy Graph Coloring

```
1:  procedure GREEDY-COLORING($G = (V, E)$)
2:      for all $v \in V$ do
3:          $color[v] \leftarrow 0$
4:      end for
5:      for all $v \in V$ do
6:          $used \leftarrow \emptyset$
7:          for all $u \in \text{Adj}[v]$ do
8:              if $color[u] \neq 0$ then
9:                  $used \leftarrow used \cup color[u]$
10:             end if
11:         end for
12:         $c \leftarrow 1$
13:         while $c \in used$ do
14:             $c \leftarrow c + 1$
15:         end while
16:         $color[v] \leftarrow c$
17:     end for
18:     return $color$
19: end procedure
```

Here is the implementation of algorithm 2:

```cpp
#include <iostream>
#include <vector>
#include <set>

using namespace std;

// This implementation uses adjacency list
    representation
using Graph = vector<vector<int>>;

vector<int> greedyColoring(const Graph &G, int n) {
   vector<int> color(n + 1, 0); // 1-based indexing

   for (int v = 1; v <= n; ++v) {
      set<int> used;
      for (int u : G[v]) {
         if (color[u] != 0)
            used.insert(color[u]);
      }

      int c = 1;
      while (used.count(c))
         ++c;

      color[v] = c;
   }

   return color;
}
```

Let us analyze this algorithm, starting with its time complexity. Since we attempt to color each vertex exactly once, the outer loop runs in $\mathcal{O}(V)$. Within this loop, we examine the colors of adjacent vertices to determine the smallest available color. Assuming the graph is represented using an adjacency list, the total number of iterations over neighbors across all vertices is proportional to the sum of degrees of all vertices, which is $2E$ in total. This is because each edge $e = (v_1, v_2)$ will be inspected twice—once when $v = v_1$ and once when $v = v_2$. Therefore, the overall time complexity of the algorithm is $\mathcal{O}(V + E)$.

Now, why is this algorithm not always going to produce the chromatic number of a graph? The answer lies in the fact that the greedy algorithm makes coloring decisions based solely on local information—that is, the colors of already-colored adjacent vertices. It does not look ahead or consider global structure. As a result, the final coloring heavily depends on the order in which the vertices are processed.

For instance, a poor ordering of vertices can lead to a coloring that uses far more colors than the chromatic number. In some cases, even graphs with a small chromatic number can be assigned significantly more colors simply due to an unfortunate ordering. In other words, the greedy algorithm is order-dependent.

To illustrate, consider a crown graph with even n vertices (also known as a bipartite graph with matching edges removed). The chromatic number of such a graph is 2, yet a bad ordering in greedy coloring can lead to using up to $n/2$ colors. Thus, while the greedy algorithm is efficient and guarantees a valid coloring, it cannot guarantee optimality in terms of the number of colors used. A figure of a crown graph with $n = 6$ vertices is shown in fig. 3 (non-optimal) and fig. 4 (optimal).

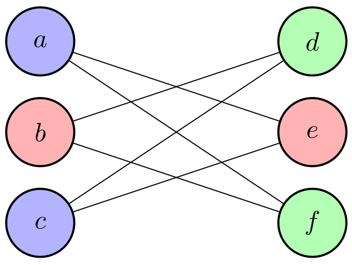

Fig. 3. Non-optimal greedy coloring on crown graph.

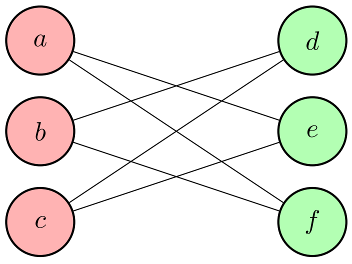

Fig. 4. Optimal 2-coloring on crown graph.

As we can see above, fig. 3 is not optimal since we can actually color the graph using only 2 colors. Such a result occurs due to a bad vertex ordering during the greedy coloring process. For instance, fig. 3 can be achieved if the vertices are visited in the following order:

$$b, e, d, f, a, c.$$

The Basic Greedy algorithm will then assign colors as follows:

- b gets color 1 (red)
- e gets color 1 (red)
- d is adjacent to $b \Rightarrow$ gets color 2 (green)
- f is adjacent to $b \Rightarrow$ gets color 2 (green)
- a is adjacent to both e and $f \Rightarrow$ needs color 3 (blue)
- c is adjacent to both c and $e \Rightarrow$ needs color 3 (blue)

Thus, the greedy algorithm ends up using 3 colors even though the chromatic number of the graph is only 2.

C. Degree of Saturation (DSatur) Algorithm

As we can see from the previous algorithm, the Basic Greedy algorithm heavily depends on the ordering of the vertices. This dependency can lead to unnecessarily high color usage on certain graphs, such as the crown graph we examined. To overcome this drawback, the DSatur algorithm introduces a more adaptive strategy: instead of following a fixed order, it selects vertices dynamically based on how "saturated" they are—that is, how many different colors appear in their neighborhood. By always choosing the vertex with the highest saturation degree, DSatur often produces colorings much closer to the chromatic number, even in challenging cases. The pseudocode for DSatur algorithm is provided in algorithm 3.

Algorithm 3 DSatur Graph Coloring

```
 1: procedure DSATUR($G = (V, E)$)
 2:     for all $v \in V$ do
 3:         $color[v] \leftarrow 0$
 4:         $sat[v] \leftarrow 0$
 5:         $deg[v] \leftarrow |\mathrm{Adj}[v]|$
 6:     end for
 7:     while UNCOLORED($color$) do
 8:         $v \leftarrow$ PICK($color,\ sat,\ deg$)
 9:         $color[v] \leftarrow$ FIRSTFIT($v,\ color$)
10:         for all $u \in \mathrm{Adj}[v]$ do
11:             if $color[u] = 0$ then
12:                 $sat[u] \leftarrow$ SATDEG($u,\ color$)
13:             end if
14:         end for
15:     end while
16:     return $color$
17: end procedure
18: procedure UNCOLORED($color$)
19:     for all $v \in V$ do
20:         if $color[v] = 0$ then
21:             return true
22:         end if
23:     end for
24:     return false
25: end procedure
26: procedure PICK($color,\ sat,\ deg$)
27:     $u \leftarrow$ any vertex with $color[u] = 0$
28:     for all $v$ with $color[v] = 0$ do
29:         if $(sat[v] > sat[u])$ or
30:           $(sat[v] = sat[u]$ and $deg[v] > deg[u])$ then
31:             $u \leftarrow v$
32:         end if
33:     end for
34:     return $u$
35: end procedure
36: procedure FIRSTFIT($v,\ color$)
37:     $used \leftarrow \emptyset$
38:     for all $u \in \mathrm{Adj}[v]$ do
39:         if $color[u] \neq 0$ then
40:             $used \leftarrow used \cup \{color[u]\}$
41:         end if
42:     end for
43:     $c \leftarrow 1$
44:     while $c \in used$ do
45:         $c \leftarrow c + 1$
46:     end while
47:     return $c$
48: end procedure
49: procedure SATDEG($v,\ color$)
50:     $s \leftarrow \emptyset$
51:     for all $u \in \mathrm{Adj}[v]$ do
52:         if $color[u] \neq 0$ then
53:             $s \leftarrow s \cup \{color[u]\}$
54:         end if
55:     end for
56:     return $|s|$
57: end procedure
```

Here is the implementation of algorithm 3:

```cpp
#include <iostream>
#include <vector>
#include <set>

using namespace std;

// This implementation uses adjacency list representation
using Graph = vector<vector<int>>;

bool uncolored(const vector<int> &color, int n) {
   for (int v = 1; v <= n; ++v)
      if (color[v] == 0)
         return true;
   return false;
}

int satDeg(int v, const Graph &G, const vector<int> &color) {
   set<int> s;
   for (int u : G[v])
      if (color[u] != 0)
         s.insert(color[u]);
   return (int)s.size();
}

int pick(const vector<int> &color, const vector<int> &sat, const vector<int> &deg, int n) {
   int u = -1;
   for (int v = 1; v <= n; ++v) {
      if (color[v] != 0) continue;
      if (u == -1 ||
         sat[v] > sat[u] ||
         (sat[v] == sat[u] && deg[v] > deg[u]))
         u = v;
   }
   return u;
}

int firstFit(int v, const Graph &G, const vector<int> &color) {
   set<int> used;
   for (int u : G[v])
      if (color[u] != 0)
         used.insert(color[u]);

   int c = 1;
   while (used.count(c))
      ++c;
   return c;
}

vector<int> dsaturColoring(const Graph &G, int n) {
   vector<int> color(n + 1, 0);
   vector<int> sat(n + 1, 0);
   vector<int> deg(n + 1, 0);

   for (int v = 1; v <= n; ++v)
      deg[v] = G[v].size();

   while (uncolored(color, n)) {
      int v = pick(color, sat, deg, n);
      color[v] = firstFit(v, G, color);

      for (int u : G[v])
         if (color[u] == 0)
            sat[u] = satDeg(u, G, color);
   }

   return color;
}
```

Now, let us analyze the time complexity of the DSatur algorithm. The DSatur algorithm processes the graph one vertex at a time, assigning colors based on the saturation degree—the number of different colors used by neighboring vertices. In each iteration of the main loop, the algorithm first checks whether any vertex remains uncolored using UNCOLORED, which takes linear time $\mathcal{O}(V)$ (this is negligible from what we are going to do inside the main loop). Next, the algorithm selects the next vertex to color using PICK, which scans all uncolored vertices and chooses the one with the highest saturation degree, breaking ties using the ordinary degree. This also requires $\mathcal{O}(V)$ time per iteration. Since we are iterating over V vertices, we already have $\mathcal{O}(V^2)$.

Once a vertex is chosen, FIRSTFIT determines the smallest color that is not used by its neighbors. It does this by collecting the colors of all adjacent vertices into a set and then scanning for the first unused color. For a vertex v with degree d_v, this takes $\mathcal{O}(d_v \log d_v)$ time due to the cost of inserting d_v items into a set data structure (from C++ STL). Thus, if we are iterating over V vertices, we would have

$$\sum_{v \in V} \mathcal{O}(d_v \log d_v) = \mathcal{O}(E \log V)$$

After the vertex is colored, the algorithm updates the saturation degree of each uncolored neighbor using SATDEG. For each such neighbor u, SATDEG scans all of u's neighbors and inserts their colors into a set to count the number of distinct ones. This operation takes $\mathcal{O}(d_u \log d_u)$ time for each neighbor u. Since this update is done for every uncolored neighbor of every vertex during the main loop, we have

$$\sum_{u \in \mathsf{Adj}[v]} \mathcal{O}(d_u \log d_u) = \mathcal{O}(E \log V)$$

Thus, the time complexity of the DSatur algorithm can be up to

$$\mathcal{O}(V^2 + E \log V + E \log V) = \mathcal{O}(V^2 + E \log V).$$

Note that this estimation reflects the behavior of the current implementation. Other implementations, particularly those that use a priority queue or heap to maintain the most saturated vertex, may reduce this overhead and achieve better performance. Although not the most efficient in terms of runtime, DSatur offers a good balance between performance and coloring quality, making it a popular heuristic for practical graph coloring tasks.

V. METHODOLOGY

To evaluate the practical performance of the backtracking, Basic Greedy, and DSatur algorithms, we designed and conducted a series of controlled experiments on a variety of synthetic graph instances. The goal is to assess both the efficiency and effectiveness of each algorithm under differing graph structures and sizes. This section details the experimental setup, including the computing environment used, the specific types of graph instances generated, and the performance metrics used to evaluate and compare the algorithms.

A. *Environment*

All programs were written in C++20 and compiled using the GNU Compiler Collection (g++) with the `-O2` optimization flag to enhance performance. Experiments were run on a MacBook Pro equipped with an Apple Silicon M4 chip and 16GB of RAM. The source code used in this paper, including the implementations of each algorithm and the benchmarking drivers, can be accessed here.

B. *Graph Instances*

To evaluate algorithm performance under different structural conditions, several types of graphs were generated:

- Random Graphs (Erdős–Rényi Model): Graphs generated using the $G(n, p)$ model, where n is the number of vertices and p is the probability of edge creation between each pair of vertices.
- Square Grid Graphs: 2D grid graphs of size $n \times n$, which are sparse and structurally regular.
- Bipartite Graphs: Graphs known to be 2-colorable, useful to test algorithm correctness on low-chromatic cases.
- Complete Graphs: Graphs where each vertex is connected to every other vertex, requiring exactly n colors for K_n.
- Crown Graphs: Structurally sparse yet challenging graphs for greedy approaches.

Graphs were tested across multiple sizes and each configuration was repeated 1001 times for each algorithm with the first iteration ignored since it's a warm-up for the cache.

C. *Metrics*

The algorithms were evaluated using the following metrics:

- Execution time: Measured in milliseconds using C++'s `chrono` library, representing the duration of the coloring process.
- Number of colors used: The total number of distinct colors assigned in the final solution.

VI. EXPERIMENTAL RESULTS AND DISCUSSIONS

This section presents and analyzes the experimental results obtained by running the backtracking, Basic Greedy, and DSatur algorithms on various types of synthetic graph instances. The goal is to provide a comprehensive comparison of their performance in terms of both execution time and the number of colors used. By assessing each algorithm under varying structural complexities and graph sizes, we gain insight into their scalability, practical efficiency, and suitability for different scenarios.

To ensure statistical reliability, each experiment was repeated 1001 times for every graph instance. The first run was discarded to eliminate cache warm-up effects, and the remaining results were averaged. The following subsections break down the findings by graph type.

A. *Performance on Random Graphs (Erdős–Rényi)*

Random graphs, generated using the Erdős–Rényi model $G(n, p)$ with $p = 0.2$, present an environment with moderate edge density and unpredictable structure. For this study, graphs of size $n = 5$, 25, and 100 were created using the `Boost` Graph Library [9], which offers a reliable and efficient random graph generator.

The performance metrics for each algorithm are summarized in Table I. We measure the average execution time in nanoseconds, as well as the average number of colors required to produce a valid coloring.

TABLE I
PERFORMANCE ON RANDOM GRAPHS ($G(n, 0.2)$)

Algorithm	**Time (ns)**	**Colors Used**	n
Backtracking	163.359	2	5
Basic Greedy	212.503	2	5
DSatur	465.822	2	5
Backtracking	833934	4	25
Basic Greedy	1647.36	4	25
DSatur	7583.13	4	25
Backtracking	∞	-	100
Basic Greedy	25264.7	10	100
DSatur	226542	8	100

At $n = 5$, all algorithms perform efficiently, using only two colors and negligible time. As n increases to 25, we begin to observe a divergence in performance. While the number of colors remains constant, execution time increases across the board—most dramatically for Backtracking, which is known to exhibit exponential growth due to its exhaustive nature.

For the largest graph size ($n = 100$), Backtracking fails to terminate within a reasonable time, confirming its impracticality for large instances. In contrast, the Basic Greedy algorithm offers superior speed, though it tends to use more colors than DSatur. DSatur, while slower, consistently produces more color-efficient solutions. This highlights the classic trade-off between speed and coloring optimality in heuristic approaches.

The results on random graphs reinforce theoretical expectations: greedy-based methods scale better than exact algorithms like Backtracking, and DSatur provides a middle ground with improved coloring quality at a moderate computational cost.

B. *Performance on Square Grid Graphs*

Square grid graphs represent a class of sparse and highly structured graphs, where each internal vertex has a degree of at most four. These graphs resemble two-dimensional lattices and are commonly used in applications such as finite element meshes and image processing. Due to their predictable topology and regularity, they offer an interesting contrast to the randomness of Erdős–Rényi graphs.

Table II summarizes the performance of each algorithm on square grids of size 5×5, 25×25, and 100×100. A notable theoretical property of such grids is their low chromatic number, which is known to be 2 for sufficiently large grids with even dimensions. This property significantly affects how the algorithms behave in practice.

TABLE II
PERFORMANCE ON SQUARE GRID GRAPHS

Algorithm	**Time (ns)**	**Colors Used**	$n \times n$
Backtracking	174.746	2	5x5
Basic Greedy	308.19	3	5x5
DSatur	994.873	4	5x5
Backtracking	331.496	2	25x25
Basic Greedy	1392.64	3	25x25
DSatur	5174.51	4	25x25
Backtracking	1291.04	2	100x100
Basic Greedy	4536.63	3	100x100
DSatur	24017.1	4	100x100

From the results, we observe that Backtracking is remarkably efficient on grid graphs compared to its performance on random graphs. This is largely due to the low chromatic number of grid graphs, which narrows the search space for valid colorings. In fact, Backtracking consistently finds optimal 2-colorings, even for large grids, with execution time increasing only modestly as n grows.

Basic Greedy, while significantly faster than DSatur, tends to overcolor the grid, consistently using 3 colors. This indicates that despite the regularity of the input, greedy coloring based purely on vertex order may still yield suboptimal results. DSatur, on the other hand, surprisingly uses 4 colors across all grid sizes—likely a consequence of its degree-saturation heuristic reacting unnecessarily to symmetric structures.

Overall, this experiment highlights how structural properties of the graph, such as planarity and low-degree regularity, can mitigate the exponential weakness of Backtracking while exposing limitations in heuristic methods that rely on local decisions without global context.

C. *Performance on Bipartite Graphs*

Bipartite graphs form an important class of graphs in graph theory and computer science due to their guaranteed 2-colorability. By definition, a graph is bipartite if its vertices can be divided into two disjoint sets such that no two vertices within the same set are adjacent. This structural property ensures that the chromatic number is exactly 2, regardless of the graph's size or density.

Table III shows the performance of all three algorithms on randomly generated bipartite graphs of increasing size. As expected, all algorithms correctly assigned only 2 colors, thus confirming their correctness on this specific class of graphs.

TABLE III
PERFORMANCE ON BIPARTITE GRAPHS

Algorithm	Time (ns)	Colors Used	n
Backtracking	227.135	2	10
Basic Greedy	278.801	2	10
DSatur	1074.8	2	10
Backtracking	1241.31	2	50
Basic Greedy	1845.71	2	50
DSatur	14918.2	2	50
Backtracking	10289.6	2	200
Basic Greedy	9751.56	2	200
DSatur	244424	2	200

TABLE IV
PERFORMANCE ON COMPLETE GRAPHS

Algorithm	Time (ns)	Colors Used	n
Backtracking	1545.16	5	5
Basic Greedy	556.467	5	5
DSatur	1295.66	5	5
Backtracking	∞	-	25
Basic Greedy	16912.4	25	25
DSatur	116233	25	25
Backtracking	∞	-	100
Basic Greedy	249516	100	100
DSatur	7204860	100	100

From the data, we observe that while all algorithms produce the correct number of colors, their execution time trends vary significantly. The Backtracking algorithm, although guaranteed to find an optimal solution in this case, still incurs growing computational cost as the number of vertices increases. This is because Backtracking explores the solution space through recursive depth-first search, even when the problem has a trivially small chromatic number.

Basic Greedy performs efficiently and remains close in speed to Backtracking for larger graphs. Interestingly, DSatur becomes the slowest of the three for larger instances, taking nearly 25 times longer than Basic Greedy at $n = 200$. This overhead stems from the need to maintain and update degree saturation information dynamically, which becomes increasingly expensive on large graphs—even when the coloring decision is straightforward.

This experiment highlights that even when all algorithms yield equally optimal solutions, their internal mechanisms can lead to vastly different performance profiles. It further reinforces the idea that algorithm selection should consider not only output quality but also computational efficiency under structural constraints.

D. Performance on Complete Graphs

Complete graphs K_n represent the worst-case scenario for coloring algorithms in terms of chromatic number. Since every vertex is adjacent to every other vertex, the chromatic number of a complete graph is exactly n. Consequently, no coloring algorithm can avoid using n colors in this setting, making optimality trivial but emphasizing raw computational performance instead.

Table IV summarizes the performance results of the three algorithms on complete graphs of varying sizes. As shown, both Greedy and DSatur color these graphs correctly by assigning a unique color to each vertex. However, the real challenge lies in the scalability of the algorithms when dealing with dense edge connectivity.

The Backtracking algorithm completely breaks down for $n \geq 25$, as the factorial growth of the solution space renders brute-force exploration infeasible. Even for smaller values, its performance is significantly slower compared to the greedy methods due to its exhaustive nature.

On the other hand, both Greedy and DSatur successfully complete the coloring process across all tested sizes. Basic Greedy remains consistently faster, owing to its minimal overhead and simple implementation. DSatur, while more computationally intensive due to its saturation degree tracking, does not gain any advantage in terms of color reduction for complete graphs—since there is no room for optimization in terms of color count.

This experiment clearly illustrates that for dense graphs with high chromatic number, the overhead of sophisticated heuristics like DSatur may not be justified. Simpler greedy strategies perform just as well in terms of solution quality, but with significantly lower execution times.

E. Performance on Crown Graphs

Crown graphs form a unique class of bipartite graphs derived from complete bipartite graphs by removing a perfect matching. While they are 2-colorable, their symmetrical and non-trivial structure makes them notoriously difficult for simple greedy heuristics, particularly those that rely on vertex ordering.

Table V presents the performance results for all three algorithms. Despite the underlying simplicity in terms of chromatic number, we observe that greedy algorithms—especially Basic Greedy—unexpectedly perform well even on larger instances. All methods consistently achieve the optimal coloring, suggesting that the vertex ordering used during experimentation happened to align favorably with the structure of the crown graphs.

TABLE V
PERFORMANCE ON CROWN GRAPHS

Algorithm	**Time (ns)**	**Colors Used**	n
Backtracking	226.209	2	10
Basic Greedy	679.08	2	10
DSatur	2248.99	2	10
Backtracking	2214.72	2	50
Basic Greedy	4360.94	2	50
DSatur	54265	2	50
Backtracking	19600.2	2	200
Basic Greedy	46114.2	2	200
DSatur	1663870	2	200

Interestingly, all three algorithms successfully achieve the optimal color count across all tested sizes. This suggests that although crown graphs are typically challenging for greedy strategies, the specific vertex ordering used in this experiment was favorable. Nevertheless, the execution time tells a different story.

DSatur, while consistent in producing optimal colorings, incurs substantial computational cost—especially evident as n increases. At $n = 200$, DSatur is nearly 40 times slower than Basic Greedy, despite producing the same result. This highlights the trade-off between robustness and efficiency: DSatur is reliable but computationally expensive, whereas Basic Greedy is fast but sensitive to vertex ordering, although this is not shown in the table.

Backtracking also demonstrates surprisingly good performance on crown graphs, likely due to the constrained solution space and low chromatic number. However, its scalability remains limited compared to greedy heuristics.

These results reinforce the notion that graph structure—not just size—plays a crucial role in determining algorithm performance. While crown graphs are theoretically simple, their deceptive topology can sometimes mislead greedy strategies unless vertex ordering is carefully considered.

F. Summary of Observations

DSatur consistently offers a good balance between color quality and runtime, especially on non-trivial graphs like random structures, though it incurs higher costs on large dense graphs. Backtracking provides optimal results but scales poorly beyond small or sparse instances. Basic Greedy is the fastest, but its color efficiency varies greatly with vertex ordering and performs poorly on complex graphs. Overall, the best algorithm depends on the graph's structure and the trade-off between speed and optimality.

VII. CONCLUSION

This paper presents a comparative study of three graph coloring algorithms across various graph structures. Backtracking delivers optimal results but suffers from poor scalability. Basic Greedy is fast but often yields suboptimal colorings, especially on complex graphs. DSatur offers a balanced trade-off, producing near-optimal solutions with acceptable runtime. The choice of algorithm should depend on the graph's structure and the application's priorities between speed and coloring quality.

APPENDIX

The complete source code used for the experiments in this paper is available on GitHub: https://github.com/Joji0/graph-coloring-comparison. The repository contains the C++ implementations of the Backtracking, Basic Greedy, and DSatur algorithms, as well as the benchmarking framework and graph generators used in the evaluation process. Furthermore, the link to the video explaining this paper is provided here: https://youtu.be/Xrqsx7Mgddg.

ACKNOWLEDGMENT

The author would like to express sincere gratitude to God Almighty for His guidance and blessings throughout the preparation of this paper. Special thanks are extended to Dr. Ir. Rinaldi Munir, M.T. and Mr. Arrival Dwi Sentosa, S.Kom., M.T., lecturers of the IF1220 Discrete Mathematics course, for their valuable teachings and support. The author also wishes to thank their family for their unwavering support during the course of these studies.

REFERENCES

[1] R. M. R. Lewis, *Guide to Graph Colouring: Algorithms and Applications*, 2nd ed., Springer, 2021.

[2] P. E. Black and P. J. Tanenbaum, "Graph," *NIST Dictionary of Algorithms and Data Structures*. [Online]. Available: https://www.nist.gov/dads/HTML/graph.html. [Accessed: May 26, 2025].

[3] R. Diestel, *Graph Theory*, 6th ed., Springer, 2024.

[4] Department of Mathematics, Princeton University, "Lecture Notes 2: Graph Coloring." [Online]. Available: https://web.math.princeton.edu/math_alive/5/Notes2.pdf. [Accessed: May 27, 2025].

[5] GCol Developers, "GCol: Graph Coloring Algorithms Library Documentation." [Online]. Available: https://gcol.readthedocs.io/en/latest/. [Accessed: May 27, 2025].

[6] GeeksforGeeks, "M Coloring Problem." [Online]. Available: https://www.geeksforgeeks.org/m-coloring-problem/. [Accessed: Jun. 9, 2025].

[7] GeeksforGeeks, "Graph Coloring Using Greedy Algorithm." [Online]. Available: https://www.geeksforgeeks.org/dsa/graph-coloring-set-2-greedy-algorithm/. [Accessed: Jun. 14, 2025].

[8] GeeksforGeeks, "DSatur Algorithm for Graph Coloring." [Online]. Available: https://www.geeksforgeeks.org/dsatur-algorithm-for-graph-coloring/. [Accessed: Jun. 17, 2025].

[9] D. Gregor and A. Lumsdaine, "Erdős–Rényi Generator," *Boost C++ Libraries Documentation*. [Online]. Available: https://www.boost.org/doc/libs/1_44_0/libs/graph/doc/erdos_renyi_generator.html. [Accessed: Jun. 19, 2025].

STATEMENT

Hereby, I declare that this paper I have written is my own work, not an adaptation or translation of someone else's paper, and not a product of plagiarism.

Bandung, 20 June 2025

[illegible]

Jonathan Kris Wicaksono
13524023