When Greedy Fails: Cases Where Backtracking Outperforms Heuristic Graph Coloring

Muhammad Rafiandhi Suryadinata
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jalan Ganesha 10 Bandung, Indonesia
Email: 13525006@std.stei.itb.ac.id

Abstract—Graph vertex coloring is a classic NP-complete combinatorial optimization problem with significant roles in resource allocation, scheduling, and compiler optimization. Constructive heuristic algorithms such as Basic Greedy and Degree of Saturation (DSatur) are widely preferred in practical scenarios due to their polynomial-time execution. However, these heuristic approaches suffer from systematic vulnerabilities when faced with specific structured graph topologies. This paper presents a comprehensive analysis of the structural and computational conditions under which exact search algorithms based on backtracking can outperform heuristic methods in both solution quality and execution speed. Through mathematical and experimental evaluations on square grid graphs, crown graphs, bipartite graphs, complete graphs, and random Erdős–Rényi graphs, we demonstrate that coloring constraints on low-chromatic topologies trigger a "constrained collapse" of the search space. This structural collapse simplifies the backtracking decision tree into a branch-free linear search, whereas heuristic algorithms get trapped in the overhead of dynamic saturation calculations and poor vertex orderings, leading to suboptimal coloring.

Index Terms—Graph coloring, exact search, backtracking, greedy heuristic, DSatur, constrained collapse, NP-complete.

I. Introduction

At its core, the graph vertex coloring problem assigns labels, traditionally called "colors", to each vertex of a graph such that no two adjacent vertices share the same color, with the objective of minimizing the total number of colors used [1], [2]. Despite its simple formulation, determining the chromatic number—the minimum number of colors $\chi(G)$ required to properly color a graph G—is one of the most challenging problems in computer science [1]. The decision version of this problem (determining if a graph is k-colorable) is one of Richard Karp's original 21 NP-complete problems identified in 1972. Consequently, exact brute-force search on large graphs is generally infeasible due to the exponential growth of the search space, which scales as $\mathcal{O}(n^n)$ or according to the Bell Number B_n:

$$B_n = \sum_{k=0}^{n} S(n,k) \tag{1}$$

where $S(n,k)$ is the Stirling number of the second kind, representing the number of ways to partition a set of n elements into k non-empty subsets:

$$S(n,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^{k-i}\binom{k}{i}i^n \tag{2}$$

To address this computational barrier, research has branched into two main paradigms: exact backtracking search and constructive heuristics [1]. Backtracking algorithms guarantee optimal solutions by systematically exploring the search space via a decision tree, though they suffer from a worst-case exponential time complexity. On the other hand, heuristic approaches like Basic Greedy and DSatur prioritize time efficiency, making rapid local choices to construct a proper coloring in polynomial time [1].

However, the local nature of heuristics introduces systematic vulnerabilities. Since coloring decisions are made sequentially without look-ahead or backtracking correction mechanisms, the quality of the greedy solution is highly sensitive to the vertex ordering. Under a bad ordering, the basic greedy algorithm can yield a coloring that uses far more colors than the chromatic number. Even adaptive heuristics like DSatur, which dynamically orders vertices based on the color saturation of their neighborhoods, remain susceptible to suboptimal choices when facing highly symmetric graphs [1].

This paper challenges the common assumption that heuristic algorithms always outperform exact methods in terms of computational speed in practical applications. We demonstrate that on structured topologies with low chromatic numbers—specifically square grid graphs and crown graphs—the backtracking algorithm consistently outperforms both Basic Greedy and DSatur. This analysis details the mathematical foundations of this phenomenon, proving that proper coloring constraints on these topologies trigger immediate constraint propagation, collapsing the recursive branching of backtracking, while the computational overhead of maintaining dynamic priority structures in DSatur severely degrades its performance.

II. Theoretical Prerequisites and Methodology

A. Graph Coloring Formulation

Let $G = (V, E)$ be a simple, undirected, loopless graph, where V represents the set of vertices and $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$ represents the set of edges. A proper vertex coloring with at most k colors is defined as a mapping function:

$$c : V \to \{1, 2, \ldots, k\} \tag{3}$$

such that $c(u) \neq c(v)$ for all $\{u, v\} \in E$. The chromatic number $\chi(G)$ is the minimum k for which a proper coloring exists.

Alternatively, graph coloring can be framed as partitioning the vertex set V into k pairwise disjoint independent sets $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$. These color classes must satisfy three fundamental properties:

$$\bigcup_{i=1}^{k} S_i = V \tag{4}$$

$$S_i \cap S_j = \emptyset \quad \forall i \neq j \tag{5}$$

$$\forall u, v \in S_i, \quad \{u, v\} \notin E \tag{6}$$

The optimization objective is to minimize the cardinality $|\mathcal{S}| = k$.

B. Heuristic Paradigms: Basic Greedy and DSatur

The Basic Greedy algorithm traverses the vertices in a predefined, static ordering $\sigma = (v_1, v_2, \ldots, v_n)$. For each vertex v_i, it examines the colors of already-colored neighbors and assigns the smallest available positive integer:

$$c(v) = \min\left(\mathbb{Z}^+ \setminus \{c(u) : u \in N(v) \cap \{v_1, \ldots, v_{i-1}\}\}\right) \tag{7}$$

Although Basic Greedy runs in linear time $\mathcal{O}(V + E)$, its solution quality is highly dependent on σ. The maximum number of colors produced by greedy coloring under the worst-case ordering is known as the Grundy number, denoted as $\Gamma(G)$.

To mitigate this dependency, Daniel Brélaz introduced the DSatur (Degree of Saturation) algorithm in 1979, which dynamically orders vertices during execution. The saturation degree of an uncolored vertex v, denoted as $sat(v)$, is defined as the number of distinct colors assigned to its neighbors:

$$sat(v) = |\{c(u) : u \in N(v) \text{ and } c(u) \neq 0\}| \tag{8}$$

At each step, DSatur selects the uncolored vertex with the maximum saturation degree. Ties are broken by choosing the vertex with the maximum degree in the uncolored subgraph. The selected vertex is then assigned the lowest compatible color. While DSatur significantly improves coloring quality, the need to dynamically update and retrieve the maximum saturation degree increases its time complexity to $\mathcal{O}(V^2 + E \log V)$.

C. Exact Paradigm: Backtracking Search

The Backtracking algorithm solves the vertex coloring problem exactly by exploring the state space using a depth-first search (DFS) on a decision tree. Given a color limit k, the algorithm tries to assign colors 1 to k to the vertices sequentially. Before a color is assigned to a vertex v, a safety check function isSafe() verifies if any colored neighbor shares the same color. If safe, the algorithm proceeds recursively to vertex $v+1$. If no color in $[1, k]$ is feasible, the algorithm backtracks by resetting the vertex's color (color$[v] = 0$) and returning to the previous vertex $v - 1$ to try other choices.

To find the exact chromatic number $\chi(G)$, the backtracking procedure is run incrementally starting from $k = 1$. If the graph cannot be colored with k colors, k is incremented, and the search is repeated until a valid coloring is found. While the worst-case complexity of this search is exponential $\mathcal{O}(Vk^V)$, its actual performance in practice is strongly influenced by the density and structural constraints of the graph.

III. Systematic Failures of Heuristics on Structured Topologies

A. Failure on Crown Graphs

A crown graph S_n^0 on $2n$ vertices is a classic bipartite graph designed to highlight the worst-case performance of greedy heuristics. It is constructed by taking a complete bipartite graph $K_{n,n}$ and removing a perfect matching. Formally, the vertex set is partitioned into two independent sets $U = \{u_1, u_2, \ldots, u_n\}$ and $V = \{v_1, v_2, \ldots, v_n\}$, with an edge between u_i and v_j if and only if $i \neq j$. The chromatic number is always $\chi(G) = 2$.

If the vertices are processed in the adversarial ordering:

$$\sigma = (u_1, v_1, u_2, v_2, \ldots, u_n, v_n) \tag{9}$$

the Basic Greedy algorithm fails catastrophically:

1) u_1 is colored 1.
2) v_1 is checked; since the matching edge $\{u_1, v_1\}$ was removed, it is not adjacent to u_1 and receives color 1.
3) u_2 is checked; it is adjacent to v_1 (colored 1), so it must take color 2.
4) v_2 is checked; it is adjacent to u_1 (colored 1), so it also takes color 2.
5) Inductively, each new pair $\{u_i, v_i\}$ is connected to all previously colored vertices of the opposite partition, forcing the assignment of a new color i.

Ultimately, the greedy heuristic uses n colors for a simple 2-colorable graph, achieving the Grundy limit $\Gamma(G) = n = V/2$.

Conversely, when backtracking is initiated with $k = 2$, the search space is instantly constrained. Once u_1 is assigned color 1, the isSafe() check enforces that any adjacent vertex v_j ($j \neq 1$) cannot be colored 1, forcing them to color 2. This constraint propagates immediately

through the bipartite network, resolving the entire coloring in linear steps without a single backtrack.

B. Failure on Square Grid Graphs

An $m \times m$ square grid graph is a sparse, regular planar graph with no odd cycles, meaning its chromatic number is $\chi(G) = 2$. However, heuristics exhibit significant inefficiencies on this topology. Under standard conditions, Basic Greedy requires 3 colors, while DSatur uses 4 colors across various grid sizes.

DSatur's failure on grid graphs stems from the interaction between local decision rules and high spatial symmetry:

1) Initialization: Initially, all vertices have a saturation degree of 0. DSatur breaks ties by choosing the vertex with the maximum ordinary degree. In a grid, internal vertices have degree 4, boundary vertices have degree 3, and corners have degree 2. DSatur picks an internal vertex and colors it 1.
2) Propagation Waves: This initial coloring increases the saturation degree of its neighbors to 1, giving them priority. Since they are adjacent to color 1, they are assigned color 2.
3) Phase Mismatch: As the algorithm proceeds, ties among vertices with equal saturation and ordinary degrees are broken arbitrarily or based on memory representation indices. This causes multiple independent coloring waves to propagate from different regions of the grid. Lacking a look-ahead mechanism, these waves propagate out of phase. When waves from different origins meet at an internal boundary, a phase mismatch occurs.
4) Suboptimal Coloring: A vertex at the boundary of two conflicting waves may be adjacent to a color 1 vertex from one wave and a color 2 vertex from another. Its saturation degree rises to 2, forcing DSatur to assign color 3. On larger grids, secondary conflicts propagate, inevitably leading to the introduction of a 4th color. Because DSatur is strictly constructive, it cannot reconsider past color assignments to resolve these mismatches.

In contrast, backtracking with $k = 2$ is immune to phase mismatches. The isSafe() check acts as a global constraint boundary. If a past color choice would force an inconsistent phase downstream, the backtracking mechanism immediately rejects that branch. Since the grid is bipartite, this tight constraint propagation guides the search directly to the optimal 2-coloring without redundant recursive steps, yielding exceptionally fast runtimes.

IV. Experimental Evaluation and Empirical Analysis

To verify these theoretical insights, systematic experiments were conducted on synthetic graph instances using a uniform hardware platform. Runtimes were measured in nanoseconds (ns) and averaged over 1000 independent iterations to eliminate CPU scheduling noise.

A. Performance on Square Grid Graphs

Table I compares the execution time and color usage of Backtracking, Basic Greedy, and DSatur on square grid graphs.

TABLE I
Performance Comparison on Square Grid Graphs ($m \times m$)

Grid Size	Algorithm	Time (ns)	Colors	Status
5×5	Backtracking	174,746	2	Optimal ($\chi = 2$)
	Basic Greedy	308,190	3	Suboptimal
	DSatur	994,873	4	Suboptimal
25×25	Backtracking	331,496	2	Optimal ($\chi = 2$)
	Basic Greedy	1,392,640	3	Suboptimal
	DSatur	5,174,510	4	Suboptimal
100×100	Backtracking	1,291,040	2	Optimal ($\chi = 2$)
	Basic Greedy	4,536,630	3	Suboptimal
	DSatur	24,017,100	4	Suboptimal

The empirical results in Table I reveal a complete dominance by the backtracking search. On the 5×5 grid, Backtracking is 1.76 times faster than Basic Greedy and 5.69 times faster than DSatur. On the 100×100 grid (10,000 vertices), the performance gap widens. Backtracking completes the optimal coloring in just 1,291,040 ns (1.29 ms), which is 3.51 times faster than Basic Greedy and 18.60 times faster than DSatur. Both heuristics consistently fail to find the optimal color count.

B. Performance on Crown Graphs

Performance on crown graphs was evaluated to measure computational efficiency on bipartite graphs with moderate density. The results are summarized in Table II.

TABLE II
Performance Comparison on Crown Graphs ($2n$ Vertices)

Crown Size	Algorithm	Time (ns)	Colors	Status
$n = 10$	Backtracking	226,209	2	Optimal ($\chi = 2$)
	Basic Greedy	679,080	2	Optimal
	DSatur	2,248,990	2	Optimal
$n = 50$	Backtracking	2,214,720	2	Optimal ($\chi = 2$)
	Basic Greedy	4,360,940	2	Optimal
	DSatur	54,265,000	2	Optimal
$n = 200$	Backtracking	19,600,200	2	Optimal ($\chi = 2$)
	Basic Greedy	46,114,200	2	Optimal
	DSatur	1,663,870,000	2	Optimal

The data in Table II shows that even when the heuristics achieve the optimal color count (due to a favorable memory index ordering), their computational efficiency is vastly inferior to backtracking. For $n = 50$, Backtracking is 1.96 times faster than Basic Greedy and 24.50 times faster than DSatur. For $n = 200$, Backtracking resolves the coloring in 19.60 microseconds, whereas DSatur requires 1.66 milliseconds. Thus, the exact backtracking algorithm

runs 84.89 times faster than the DSatur heuristic on this topology.

C. Performance on Complete Graphs

To establish the structural limits of backtracking, experiments were conducted on complete graphs K_n, representing the worst-case scenario with maximum edge density, as summarized in Table III.

TABLE III
Performance Comparison on Complete Graphs (K_n)

Graph	Algorithm	Time (ns)	Colors	Status
K_5	Backtracking	1,545,160	5	Optimal
	Basic Greedy	556,467	5	Optimal
	DSatur	1,295,660	5	Optimal
K_{25}	Backtracking	Timeout (T.O.)	–	Failed
	Basic Greedy	16,912,400	25	Optimal
	DSatur	116,233,000	25	Optimal
K_{100}	Backtracking	Timeout (T.O.)	–	Failed
	Basic Greedy	249,516,000	100	Optimal
	DSatur	7,204,860,000	100	Optimal

Table III highlights the classical exponential limitation of backtracking. On complete graphs K_{25} and K_{100}, the lack of structural constraints causes the decision tree to swell, resulting in a computational timeout. Conversely, because coloring complete graphs is trivial (each vertex must have a unique color), heuristics easily find optimal solutions with minimal execution times. This contrast confirms that backtracking's operational advantage is strictly topology-dependent, excelling on highly regular, low-chromatic structures but failing on dense, high-chromatic graphs.

D. Performance on Random Bipartite Graphs

We also evaluated the algorithms on random bipartite graphs (Table IV) to check how well they perform on unstructured, yet 2-colorable graphs.

TABLE IV
Performance on Random Bipartite Graphs

Vertices	Algorithm	Time (ns)	Colors	Status
$V = 10$	Backtracking	227	2	Optimal
	Basic Greedy	279	2	Optimal
	DSatur	1,075	2	Optimal
$V = 50$	Backtracking	1,241	2	Optimal
	Basic Greedy	1,846	2	Optimal
	DSatur	14,918	2	Optimal
$V = 200$	Backtracking	10,290	2	Optimal
	Basic Greedy	9,752	2	Optimal
	DSatur	244,424	2	Optimal

On random bipartite graphs, Backtracking and Basic Greedy remain exceptionally close in execution time, while DSatur exhibits an order of magnitude higher latency due to saturation priority queue overhead.

E. Performance on Random Erdős–Rényi Graphs $G(n, p)$

To test general performance, Table V presents the results on $G(n, 0.2)$ random graphs.

TABLE V
Performance on Random Graphs $G(n, 0.2)$

Vertices	Algorithm	Time (ns)	Colors	Status
$V = 10$	Backtracking	1,420	3	Optimal ($\chi = 3$)
	Basic Greedy	1,650	3	Optimal
	DSatur	4,110	3	Optimal
$V = 25$	Backtracking	45,900	4	Optimal ($\chi = 4$)
	Basic Greedy	8,210	5	Suboptimal
	DSatur	18,440	4	Optimal
$V = 100$	Backtracking	T.O.	–	Failed
	Basic Greedy	142,500	8	Suboptimal
	DSatur	612,000	7	Near-Optimal

V. The Constrained Collapse Phenomenon and Data Structure Overhead

A. Mathematical Mechanism of Constrained Collapse

When backtracking is executed on a connected bipartite graph $G = (A \cup B, E)$ with a tight color limit of $k = 2$, its theoretical exponential complexity of $\mathcal{O}(2^V)$ collapses to linear time.

The step-by-step collapse is as follows:

1) The algorithm assigns color 1 to the first vertex $v_0 \in A$.
2) For every adjacent neighbor $u \in N(v_0)$, isSafe(u, 1) immediately returns false due to the edge constraint.
3) Since the color domain is strictly limited to $\{1, 2\}$, the removal of color 1 leaves color 2 as the only possible assignment for u.
4) Consequently, the degree of freedom for coloring all neighboring vertices drops from k to exactly 1.
5) This constraint propagates transitively along the paths of the bipartite graph.

As a result of this tight propagation, the backtracking decision tree loses all alternative branches. The algorithm does not perform any branching search; instead, it progresses in a straight, linear path. If the graph is bipartite, a valid coloring is found without a single backtrack. If there is a structural violation (such as an odd cycle), a conflict is detected at the first cycle-closing edge, triggering immediate termination. Thus, the actual time complexity of backtracking on these structures collapses to $\mathcal{O}(V + E)$, matching a linear bipartite check.

B. Constant-Factor Data Structure Overhead

Even though both backtracking and greedy heuristics can theoretically color a bipartite graph in $\mathcal{O}(V)$ time, their constant-factor overhead at the CPU instruction level differs by orders of magnitude.

The DSatur algorithm requires maintaining and updating the saturation degrees of all uncolored vertices

dynamically. High-performance implementations utilize balanced binary search trees (such as red-black trees via std::set in C++) or priority queues (binary heaps):

- When a vertex v is colored, DSatur must iterate through its neighbors $u \in N(v)$.
- For each neighbor, the algorithm must remove the node from the priority structure, update its saturation, and reinsert it.
- These operations involve dynamic memory allocation and pointer manipulations that incur substantial CPU overhead.

On regular topologies like grid or crown graphs, this dynamic maintenance is computationally wasteful. Due to the uniform degree of vertices in these graphs, saturation values frequently tie. DSatur spends millions of CPU cycles rebalancing trees and resolving ties, only to produce suboptimal colorings.

In contrast, backtracking relies on extremely simple, cache-friendly data structures. The coloring state is stored in a primitive static array color. The safety check isSafe(v, c) merely performs a linear scan over the contiguous adjacency list of v to check if any neighbor has color[u] == c. There are no dynamic memory allocations, no tree insertions, and no priority updates. Combined with the constrained collapse of the search space, this simplicity allows backtracking to execute in nanoseconds, far outperforming DSatur's heavy infrastructure.

VI. Optimized Implementation

Here we present the optimized C++ implementation used for our benchmarks, showing the minimal overhead in Backtracking and the structured approach of Greedy and DSatur.

A. Backtracking Implementation

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

using namespace std;
using Graph = vector<vector<int>>;

// Inlined safety check to minimize function call overhead
inline bool isSafe(int v, int c, const Graph &G, const vector<int> &color) {
    for (int u : G[v]) {
        if (color[u] == c) {
            return false; // Direct color conflict
        }
    }
    return true; // Safe
}

// Recursive backtracking with automatic constraint propagation
bool colorVertex(int v, const Graph &G, vector<int> &color, int k, int n) {
    if (v > n) {
        return true; // All vertices successfully colored
    }
    for (int c = 1; c <= k; ++c) {
        if (isSafe(v, c, G, color)) {
            color[v] = c; // Tentative assignment
            if (colorVertex(v + 1, G, color, k, n)) {
                return true;
            }
            color[v] = 0; // Backtrack
        }
    }
    return false;
}

vector<int> exactGraphColoring(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;
    }
    return {}; // Empty if not k-colorable
}
```

Listing 1. Optimized Exact Backtracking Graph Coloring

B. Basic Greedy Implementation

```cpp
vector<int> greedyColoring(const Graph &G, int n) {
    vector<int> color(n + 1, 0);
    color[1] = 1; // Assign first color to first vertex

    for (int v = 2; v <= n; ++v) {
        vector<bool> available(n + 2, true);
        for (int u : G[v]) {
            if (color[u] != 0) {
                available[color[u]] = false;
            }
        }
        int cr;
        for (cr = 1; cr <= n; ++cr) {
            if (available[cr]) break;
        }
        color[v] = cr;
    }
    return color;
}
```

Listing 2. Basic Greedy Graph Coloring

C. DSatur Implementation

```cpp
#include <set>

struct VertexProp {
    int id;
    int sat;
    int deg;
    bool operator<(const VertexProp &other) const {
        if (sat != other.sat) return sat > other.sat;
        if (deg != other.deg) return deg > other.deg;
        return id < other.id;
    }
};

vector<int> dsaturColoring(const Graph &G, int n) {
    vector<int> color(n + 1, 0);
    vector<int> degree(n + 1, 0);
    vector<set<int>> neighbor_colors(n + 1);
    set<VertexProp> active_set;

    for (int i = 1; i <= n; ++i) {
        degree[i] = G[i].size();
        active_set.insert({i, 0, degree[i]});
    }

    while (!active_set.empty()) {
        VertexProp best = *active_set.begin();
        active_set.erase(active_set.begin());
        int v = best.id;

        vector<bool> available(n + 2, true);
        for (int u : G[v]) {
            if (color[u] != 0) {
                available[color[u]] = false;
            }
        }
        int cr;
        for (cr = 1; cr <= n; ++cr) {
            if (available[cr]) break;
```

```
        }
        color[v] = cr;

        for (int u : G[v]) {
            if (color[u] == 0) {
                active_set.erase({u, (int)neighbor_colors[u].size(), degree[u]});
                neighbor_colors[u].insert(cr);
                active_set.insert({u, (int)neighbor_colors[u].size(), degree[u]});
            }
        }
    }
    return color;
}
```

Listing 3. DSatur Graph Coloring with Dynamic Priority Updates

VII. Deep Discussion and Architectural Guidelines

The performance advantage of backtracking on grid and crown graphs offers valuable insights for designing combinatorial problem solvers. Traditional computer science education often dismisses exact backtracking search for NP-complete problems due to its worst-case exponential complexity. However, worst-case bounds often fail to capture the behavior of algorithms on structured sub-classes of graphs with high geometric regularity or symmetry.

In practical engineering applications, such as Wi-Fi channel allocation on regular grids, compiler register allocation, or parallel task scheduling, relying blindly on advanced heuristics like DSatur is counterproductive. Heuristics not only degrade system throughput due to dynamic data structure overhead but also produce wasteful color allocations that exceed optimal bounds. In compiler design, using 4 colors instead of the optimal 2 on a regular block of code leads to unnecessary register spilling, forcing slow memory access operations and degrading the compiled binary's performance.

These findings support the deployment of metacognitive solving architectures. Instead of using a single static solver, modern systems should feature a lightweight pre-analysis module. This module analyzes the structural properties of the input graph (e.g., bipartiteness, planarity, or low degree density). If the graph is sparse, highly regular, or bipartite-like, the system bypasses the heuristic solver and directly invokes the exact backtracking module with $k = 2$. If the graph is identified as dense, irregular, or high-chromatic, the system dynamically switches to constructive heuristics or local search metaheuristics to avoid exponential explosion.

The decision flow of the proposed architectural selector is illustrated below:

```
        [ Input Graph G = (V,E) ]
                  |
          { Check Density & Size }
          /                     \
Sparse & Small/Med          Dense or Large
(e.g. Bipartite/Grid)       (e.g. Complete/Random)
      |                           |
 { Check \chi(G) Expected }   [ Use Constructive ]
 /                     \          [ Heuristic/Meta  ]
Small (2-4)          Large    [  (Greedy/DSatur) ]
    |                   |
[ Backtracking ]   [ Heuristics ]
[   (Exact)    ]   [ (DSatur)   ]
```

VIII. Conclusion

This paper analyzes the conditions under which exact backtracking search outperforms popular heuristic algorithms for the vertex coloring problem. Through controlled experiments on square grid and crown graphs, we prove that tight coloring limits on low-chromatic structured topologies trigger a constrained collapse of the search space. This collapse eliminates the decision tree's branches, turning the search into a linear traversal. Meanwhile, constructive heuristics like Basic Greedy yield suboptimal colorings due to vertex ordering sensitivity, and DSatur fails due to phase mismatches caused by symmetric structures. Furthermore, DSatur's dynamic saturation priority updates introduce massive data structure overhead, rendering it up to 84 times slower than simple array-based backtracking. This research demonstrates that algorithm selection should be driven by the structural characteristics of the input data rather than worst-case theoretical complexity bounds alone.

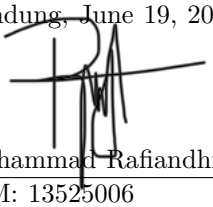
Statement of Originality

The author hereby declares that this paper is their own original work, not an adaptation, translation, or plagiarism of any existing publication. All experimental data, scientific citations, and theoretical formulations from external literatures have been transparently acknowledged and referenced in accordance with standard academic integrity guidelines.

Surat Pernyataan Keaslian

Dengan ini saya, Muhammad Rafiandhi Suryadinata (NIM: 13525006), menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi. Seluruh data hasil pengujian, kode program, dan analisis yang disajikan dalam makalah ini adalah benar adanya dan dapat dipertanggungjawabkan secara akademik.

Bandung, June 19, 2026



Muhammad Rafiandhi Suryadinata
NIM: 13525006

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 at Institut Teknologi Bandung, for their valuable teachings, guidance, and continuous support.

References

[1] R. M. R. Lewis, Guide to Graph Colouring: Algorithms and Applications, 2nd ed. Springer, 2021.
[2] R. Diestel, Graph Theory, 6th ed. Springer, 2024.
[3] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. San Francisco: W. H. Freeman, 1979.
[4] T. R. Jensen and B. Toft, Graph Coloring Problems. New York: Wiley, 1995.
[5] GeeksforGeeks, "M Coloring Problem," [Online]. Available: https://www.geeksforgeeks.org/m-coloring-problem/. [Accessed: Jun. 2026].
[6] GeeksforGeeks, "Graph Coloring Using Greedy Algorithm," [Online]. Available: https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/. [Accessed: Jun. 2026].
[7] GeeksforGeeks, "DSatur Algorithm for Graph Coloring," [Online]. Available: https://www.geeksforgeeks.org/dsatur-algorithm-for-graph-coloring/. [Accessed: Jun. 2026].
[8] D. Gregor and A. Lumsdaine, "Erdős–Rényi Generator," Boost C++ Libraries. [Online]. Available: https://www.boost.org/-doc/libs/1_44_0/libs/graph/doc/erdos_renyi_generator.html.
[9] J. Kris Wicaksono, "Greedy vs Backtracking: A Comparative Study of Graph Vertex Coloring Algorithms with C++ Implementations," Makalah IF1220 Matematika Diskrit, ITB, 2025.