Python Final Project Report - Graph Coloring

Songnan Lin, Zhanfei Shi, Yue Yu

Dec 2025

1 Introduction

The graph coloring problem (GCP) is one of the most interesting and difficult combinatorial optimization problems in computer science, mathematics, and operations research [1]. The goal of our project is to study this problem. Graph coloring is defined as coloring the nodes of a graph with the minimum number of colors without any two adjacent nodes sharing the same color [2]. Determining the minimum number of colors needed, known as the chromatic number, makes it a rich setting for exploring exact, heuristic, and stochastic algorithms.

Graph coloring is particularly interesting because it connects deep mathematical theory with intuitive real-world problems. A famous example is the Four Color Problem [6], which asks whether every map can be colored using only four colors so that no two adjacent regions share the same color. Although this result was verified by an exhaustive computer search, it still does not have a fully accepted theoretical proof, making it one of the most intriguing questions in graph theory. Beyond its theoretical appeal, graph coloring appears in many practical settings such as scheduling tasks without conflicts, assigning frequencies in wireless networks, and coloring maps to distinguish neighboring regions. These examples show why graph coloring is both mathematically fascinating and highly relevant in real applications.

The graph coloring problem can be defined as follows: Given an undirected graph $G = (V, E)$ where V represents the set of vertices and E represents the set of edges, the objective is to find a proper coloring assignment $c : V \to \{1, 2, ..., k\}$ such that for every edge $(u, v) \in E$, $c(u) \neq c(v)$, while minimizing the number of colors k used. The minimal number of colors required for proper coloring is known as the chromatic number $\chi(G)$.

In this project, we investigate several computational approaches to graph coloring. We study both exact and approximate methods, which allows us to compare trade-offs between runtime, optimality, and stability. The algorithms we implement are:

- **Greedy Coloring**, a simple heuristic that colors vertices sequentially based on available colors.
- **DSatur**, a more advanced heuristic that selects vertices in the order of highest saturation degree—the number of distinct colors already used by their colored neighbors—thereby prioritizing the most constrained vertices and often yielding strong results across many graph families.
- **Backtracking Search**, an exact method that explores the combinatorial search space with pruning, guaranteeing optimal solutions but with potentially exponential runtime.

- **Simulated Annealing**, a stochastic local-search method inspired by thermodynamic cooling, which attempts to reduce conflicts under a fixed number of colors and often finds near-optimal solutions efficiently.

We carry out experiments on a variety of graph models. These include small handcrafted graphs such as cycle graphs, path graphs, and star graphs, as well as graphs with different structures or densities to observe how algorithm performance depends on problem characteristics. Our experiments examine correctness, runtime, sensitivity to ordering, and the effect of graph density and structure. This combination of multiple algorithms and graph models allows us to compare how different computational strategies behave on the same underlying combinatorial problem.

2 Numerical Methods and Notation

In this project, we study several algorithms for graph coloring. Each method approaches the problem in a different way, from exact search to fast heuristics and stochastic optimization. In this section, we describe the numerical methods we use, the notation for each method, and what each algorithm is computing. Our explanations follow simple and clear ideas without going too deep into theory. We also mention the Python packages and data structures that support our code.

2.1 Greedy Coloring

The greedy algorithm colors vertices sequentially in a specified order, assigning to each vertex the smallest available color that does not conflict with any of its already-colored neighbors. This method is extremely fast with time complexity $O(n + m)$, where n is the number of vertices and m is the number of edges, though it does not guarantee an optimal coloring.

2.1.1 Algorithm Description

If vertices are processed in the order $\{v_1, v_2, \ldots, v_n\}$, the color assigned to vertex v_i is

$$x_{v_i} = \min\{c \in \mathbb{Z}_{\geq 0} : c \neq x_u \text{ for all } u \in N(v_i) \text{ with } x_u \neq -1\},$$

where c represents a non-negative integer (0, 1, 2, ...) that serves as a color label, $N(v_i)$ denotes the neighbor set of v_i, and $x_u = -1$ indicates that vertex u has not yet been colored. The algorithm maintains a set of used colors for each vertex's neighbors and selects the minimum non-conflicting color.

2.1.2 Vertex Ordering Strategy

The performance of the greedy algorithm depends critically on the order in which vertices are processed. We implement two ordering strategies:

- **Natural order**: Vertices are processed in the sequence $\{0, 1, 2, \ldots, n-1\}$.
- **Degree-based order**: Vertices are sorted by their degree (number of neighbors) in descending order, so vertices with higher degrees are colored first.

The degree-based ordering can be expressed as processing vertices in the order $\{v_{i_1}, v_{i_2}, \ldots, v_{i_n}\}$ such that $\deg(v_{i_1}) \geq \deg(v_{i_2}) \geq \cdots \geq \deg(v_{i_n})$.

2.1.3 Implementation Details

The algorithm computes a valid coloring, the runtime to complete the coloring, and the total number of colors used. We implement the greedy algorithm using:

- Python lists to store color assignments, where `colors`$[i]$ represents the color assigned to vertex i
- Adjacency lists from the `Graph` class to efficiently access neighborhoods $N(v_i)$
- The `time` module to record execution times
- A set data structure to track colors used by neighbors

The algorithm terminates after a single pass through all vertices, making it one of the fastest graph coloring heuristics available, though the quality of the solution depends on the chosen vertex ordering.

2.2 DSATUR Coloring

The DSATUR (Degree of Saturation) algorithm is an improved greedy heuristic that dynamically selects the next vertex to color based on its *saturation degree*—the number of distinct colors already used by its neighbors. By prioritizing vertices with high saturation, DSATUR often produces better colorings than the standard greedy algorithm, though it still does not guarantee optimality.

2.2.1 Algorithm Description

At each iteration, DSATUR selects the uncolored vertex v_i with the highest saturation degree, defined as:

$$\text{sat}(v_i) = |\{x_u : u \in N(v_i) \text{ and } x_u \neq -1\}|,$$

where $N(v_i)$ denotes the neighbor set of v_i, and $x_u = -1$ indicates that vertex u has not yet been colored. The saturation degree represents how many different colors are already used by the neighbors of v_i.

If multiple vertices have the same maximum saturation degree, the algorithm breaks ties by selecting the vertex with the highest degree:

$$v^* = \arg \max_{\substack{v_i \\ \text{uncolored}}} \{\text{sat}(v_i), \deg(v_i)\},$$

where ties are resolved by choosing the vertex with the highest degree $\deg(v_i)$.

Once a vertex v^* is selected, it is assigned the smallest available color:

$$x_{v^*} = \min\{c \in \mathbb{Z}_{\geq 0} : c \notin \{x_u : u \in N(v^*)\}\}.$$

After coloring v^*, the saturation degrees of all its neighbors are updated to reflect the newly assigned color.

2.2.2 Key Differences from Greedy Coloring

Unlike the standard greedy algorithm, which processes vertices in a fixed order, DSATUR adaptively chooses the next vertex based on the current state of the coloring. This dynamic selection strategy allows DSATUR to:

- Prioritize highly constrained vertices (those with many differently colored neighbors), which are more likely to require new colors
- Delay coloring vertices with fewer constraints, potentially allowing them to reuse existing colors
- Often achieve better colorings than fixed-order greedy methods

2.2.3 Time Complexity and Implementation

The DSATUR algorithm has time complexity $O(n^2 + m)$, where n is the number of vertices and m is the number of edges. The quadratic term arises from the need to scan all uncolored vertices at each iteration to find the one with maximum saturation.

We implement DSATUR using:

- Python lists to store color assignments, where `colors`$[i]$ represents the color assigned to vertex i
- Sets to maintain `neighbor_colors`$[v]$ for each vertex v, tracking which colors are used by its neighbors
- Adjacency lists from the `Graph` class to efficiently access neighborhoods $N(v_i)$
- The `time` module to record execution times

The algorithm maintains saturation information incrementally: when a vertex is colored, the corresponding color is added to the neighbor color sets of all its adjacent vertices, enabling efficient saturation degree computation.

2.3 Backtracking Coloring

The backtracking algorithm is an *exact* method that systematically explores all possible colorings to find the optimal (minimum) number of colors. Unlike heuristic algorithms, backtracking guarantees an optimal solution, though at the cost of exponential time complexity in the worst case.

2.3.1 Algorithm Description

The algorithm constructs a search tree where each node represents a partial coloring. At depth k, vertices $\{v_1, v_2, \ldots, v_k\}$ have been assigned colors, and the algorithm attempts to extend this partial coloring to vertex v_{k+1}.

For vertex v_i at position i in the ordering, the algorithm tries colors in a specific sequence:

$$\text{Try colors in order: } 0, 1, 2, \ldots, \text{used\_colors} - 1, \text{used\_colors}$$

That is, it first attempts to reuse existing colors (0 through used_colors), then considers introducing a new color if necessary.

For each candidate color c, the algorithm checks if it is *safe* to assign to vertex v_i:

$$\text{is\_safe}(v_i, c) = \bigwedge_{u \in N(v_i)} (x_u \neq c),$$

where $N(v_i)$ denotes the neighbor set of v_i, and x_u is the color assigned to vertex u (or -1 if uncolored). If safe, the algorithm recursively explores the subtree; otherwise, it backtracks to try the next color.

2.3.2 Backtracking and Pruning Mechanisms

The algorithm employs two key optimization strategies:

1. Backtracking When a partial coloring cannot be extended (no safe color exists for the current vertex), the algorithm *backtracks* by:

- Undoing the color assignment to the current vertex: $x_{v_i} \leftarrow -1$
- Returning to the previous vertex and trying the next color option
- This allows exploration of alternative colorings without missing any possibilities

2. Pruning (Bound-based) The algorithm maintains the best solution found so far, denoted best_num_colors. At any node in the search tree, if the current partial coloring already uses used_colors $\geq$ best_num_colors, the entire subtree is pruned:

$$\text{if used\_colors} \geq \text{best\_num\_colors then prune subtree}$$

This is safe because any extension of this partial coloring will use at least used_colors colors, which cannot improve upon the current best solution.

2.3.3 Guarantee of Optimality

The backtracking algorithm guarantees an optimal solution because:

1. **Exhaustive Search**: It systematically explores all possible colorings (or prunes branches that cannot be optimal)
2. **Systematic Exploration**: For each vertex, it tries all possible colors in order
3. **Best Solution Tracking**: It always maintains the best solution found so far
4. **Safe Pruning**: Pruning only eliminates branches that provably cannot yield better solutions

Since the algorithm explores the entire solution space (subject to pruning), it must eventually find the coloring that uses the minimum number of colors.

2.3.4 Vertex Ordering Strategy

As with the greedy algorithm, backtracking supports two vertex ordering strategies:

- **Natural order**: Vertices processed as $\{0, 1, 2, \ldots, n-1\}$
- **Degree-based order**: Vertices sorted by degree in descending order: $\deg(v_{i_1}) \geq \deg(v_{i_2}) \geq \cdots \geq \deg(v_{i_n})$

The ordering significantly affects performance:

- Coloring high-degree vertices first increases the likelihood of early conflicts
- Early conflicts enable earlier backtracking and pruning
- This can dramatically reduce the number of nodes explored in the search tree

2.3.5 Time Complexity

The worst-case time complexity is exponential: $O(k^n)$ where k is the chromatic number and n is the number of vertices. However, with effective pruning and good vertex ordering, the average-case performance is often much better.

The complexity can be analyzed as:

- **Search tree size**: In the worst case, the tree has $O(k^n)$ nodes, where each node represents a color assignment
- **Per-node cost**: $O(\deg(v))$ to check safety of a color assignment for vertex v
- **Total**: $O(k^n \cdot \Delta)$ where Δ is the maximum degree, though pruning typically reduces this significantly

The number of nodes actually visited depends heavily on:

- The effectiveness of pruning (better bounds lead to more pruning)
- Vertex ordering (good ordering finds solutions faster, enabling more pruning)
- Graph structure (sparse graphs often allow more pruning than dense graphs)

2.3.6 Implementation Details

We implement backtracking using:

- Recursive depth-first search to explore the search tree
- Python lists to store the current coloring state
- A dictionary mapping vertices to their positions in the ordering
- Global variables to track the best solution found and nodes visited
- The `time` module to record execution times

The algorithm returns:

- The optimal coloring (if one exists)
- The minimum number of colors used
- The number of nodes explored in the search tree (useful for analyzing pruning effectiveness)
- The total execution time

2.3.7 Trade-offs

- **Optimality**: Guaranteed to find the minimum number of colors
- **Speed**: Exponential time complexity; can be slow for large graphs
- **Practical Use**: Best suited for small to medium graphs where optimality is required

This makes backtracking the method of choice when exact solutions are needed, while heuristic methods (greedy, DSATUR) are preferred for large graphs where approximate solutions suffice.

2.4 Simulated Annealing Coloring

Simulated annealing is a metaheuristic optimization algorithm inspired by the physical process of annealing in metallurgy, where metal is heated and slowly cooled to reduce defects. Unlike the previous algorithms, simulated annealing requires a *fixed number of colors* k as input and attempts to find a valid coloring using exactly k colors. The algorithm may not always succeed, and the result may contain conflicts (adjacent vertices with the same color).

2.4.1 Algorithm Description

The algorithm begins with a random initial coloring where each vertex is assigned a color uniformly at random from $\{0, 1, \ldots, k-1\}$:

$$x_v \sim \text{Uniform}\{0, 1, \ldots, k-1\} \quad \text{for all } v \in V$$

The algorithm then iteratively attempts to improve the coloring through a series of random modifications. At each iteration t:

1. Random Modification A random vertex v is selected, and a new color c_{new} is chosen uniformly at random from $\{0, 1, \ldots, k-1\}$ such that $c_{\text{new}} \neq x_v$ (the current color of v).

2. Conflict Evaluation The change in the number of conflicts is computed. A *conflict* occurs when two adjacent vertices share the same color. The conflict count is defined as:

$$\text{conflicts}(G, \mathbf{x}) = \frac{1}{2} \sum_{(u,v) \in E} \mathbf{1}[x_u = x_v],$$

where $\mathbf{1}[\cdot]$ is the indicator function. When recolor vertex v from color c_{old} to c_{new}, only edges incident to v can change their conflict status. The change in conflicts is:

$$D = \text{conflicts}_{\text{new}} - \text{conflicts}_{\text{old}} = \sum_{u \in N(v)} \left(\mathbf{1}[x_u = c_{\text{new}}] - \mathbf{1}[x_u = c_{\text{old}}] \right),$$

where $N(v)$ denotes the neighbors of vertex v.

3. Acceptance Criterion The algorithm uses a probabilistic acceptance rule:

$$\text{Accept change} = \begin{cases} \text{True} & \text{if } D \leq 0 \text{ (improvement or neutral)} \\ \text{True with probability } e^{-D/T} & \text{if } D > 0 \text{ (worsening)} \end{cases}$$

where T is the current *temperature*. This means:

- Improvements ($D \leq 0$) are always accepted
- Worsening moves ($D > 0$) are accepted with probability $P(\text{accept}) = e^{-\Delta/T}$
- The acceptance probability decreases as D increases (worse moves less likely)
- The acceptance probability decreases as T decreases (cooling makes bad moves less likely)

4. Cooling Schedule After each iteration, the temperature is reduced according to:

$$T_{t+1} = \alpha \cdot T_t,$$

where $\alpha \in (0, 1)$ is the cooling rate (typically $\alpha = 0.999$). The initial temperature T_0 (typically $T_0 = 1.0$) determines how willing the algorithm is to accept bad moves initially.

2.4.2 Key Characteristics

- **Requires k as input**: Unlike other algorithms that find the minimum number of colors, simulated annealing attempts to color the graph with a predetermined number k of colors.
- **May not find valid solution**: The algorithm may terminate with conflicts remaining if it cannot find a valid k-coloring within the iteration limit.
- **Early termination**: If a valid coloring (0 conflicts) is found, the algorithm stops immediately.
- **Stochastic**: The random nature means multiple runs may yield different results.

2.4.3 Time Complexity

The algorithm has time complexity $O(\text{max\_iter} \cdot \Delta)$, where max_iter is the maximum number of iterations (default 20,000) and Δ is the maximum vertex degree. At each iteration:

- Vertex and color selection: $O(1)$
- Conflict calculation: $O(\deg(v))$ where v is the modified vertex
- Acceptance decision: $O(1)$
- Temperature update: $O(1)$

Since $\deg(v) \leq \Delta$ for all vertices, and we perform max_iter iterations, the total complexity is $O(\text{max\_iter} \cdot \Delta)$.

2.4.4 Parameter Selection

The algorithm's performance depends on several parameters:

- **Initial temperature** T_0: Higher values allow more exploration initially but may waste time on poor solutions.
- **Cooling rate** α: Closer to 1.0 means slower cooling (more exploration) but longer runtime.
- **Maximum iterations**: More iterations increase the chance of finding a solution but increase runtime. Must balance between solution quality and computational cost.
- **Number of colors** k: Must be at least the chromatic number $\chi(G)$ for a valid coloring to exist. If $k < \chi(G)$, no valid coloring is possible.

2.4.5 Implementation Details

We implement simulated annealing using:

- Random initialization: Each vertex assigned a random color from $\{0, 1, \ldots, k-1\}$
- Efficient conflict counting: Only recalculates conflicts for edges incident to the modified vertex
- Exponential acceptance probability: Uses $e^{-\Delta/T}$ for probabilistic acceptance
- Geometric cooling: Temperature multiplied by α each iteration
- Early termination: Stops immediately upon finding a valid coloring (0 conflicts)
- The `time` module to record execution times

The algorithm returns:

- The coloring found (may be `None` if no valid coloring exists)

- The number of colors used (always k)
- The number of conflicts remaining (0 if valid, > 0 if invalid)
- The total execution time

2.4.6 Trade-offs and Use Cases

- **Optimality**: No guarantee of optimality; may not even find a valid solution
- **Speed**: Fast, with controllable runtime via max_iter
- **Flexibility**: Can be tuned via temperature and cooling parameters
- **Use case**: Suitable when k is known or bounded, and approximate solutions are acceptable
- **Limitation**: Requires $k \geq \chi(G)$ to have any chance of success

Simulated annealing is particularly useful in scenarios where the number of available colors is constrained (e.g., scheduling problems with fixed resources) or when exploring whether a graph is k-colorable for a specific value of k.

3 Package Structure

Our project follows a clean and modular package structure. All source code is located in the `src` directory, and each algorithm is implemented in its own dedicated module. The file `graph.py` defines the basic graph data structure as well as utility functions for checking whether a coloring is valid. The main algorithms—Greedy Coloring, DSATUR, Backtracking Search, and Simulated Annealing—are implemented respectively in `greedy.py`, `dsatur.py`, `backtracking.py`, and `annealing.py`.

All unit tests are placed in the `test` directory, allowing each algorithm to be validated independently.

This overall structure is consistent with common Python project conventions and keeps the algorithms, tests, and documentation well organized and easy to maintain.

4 Tests

We follow the project guidelines by placing all test files in the `test` directory, with one test module corresponding to each source file in `src`.

4.1 Greedy Coloring

To verify the correctness of our greedy coloring implementation, we created a dedicated test module `test_greedy.py` using `pytest`. This file contains a collection of unit tests that evaluate both the structure of the returned result object and the validity of the coloring produced by the greedy algorithm.

First, in the test `test_greedy_returns_result_object`, we check that the function `greedy_coloring` returns an instance of `GreedyResult` with all required attributes:

- `coloring`: a list assigning a color index to each vertex;
- `num_colors`: the number of colors used by the greedy algorithm;
- `time_seconds`: the total runtime of the function.

We additionally verify that the `__repr__` method of the result object produces a meaningful description containing the class name and relevant fields such as `num_colors`.

Next, we validate the greedy algorithm on graphs with known chromatic number. In the test `test_greedy_coloring`, we construct a path graph on four vertices, which is known to be 2-colorable. The test ensures that the algorithm:

- returns a non-empty coloring;
- produces a proper coloring, verified using `is_proper_coloring` from `graph.py`;
- uses exactly two colors, matching the chromatic number of the path.

Because greedy coloring may depend on vertex ordering, the test also evaluates `greedy_coloring(g, use_degree_order = False)`. We check that the algorithm still produces a valid coloring regardless of whether degree ordering is used.

In all cases, the correctness of the returned coloring is verified by checking that no two adjacent vertices share the same color. Running `pytest test_greedy.py` confirms that all tests pass, giving strong evidence that our implementation of the greedy algorithm produces valid colorings and returns results in the correct format.

4.2 DSATUR Coloring

To validate the correctness of our DSATUR implementation, we created the test module `test_dsatur.py` using `pytest`. This module contains a collection of unit tests that examine both the structure of the returned result object and the validity of the colorings produced by the DSATUR algorithm.

The first test, `test_dsatur_returns_result_object`, ensures that the function `dsatur_coloring` returns an instance of `DSATURResult` with all required attributes:

- `coloring`: a list that assigns a color index to each vertex;
- `num_colors`: the total number of colors used in the final solution;
- `time_seconds`: the total runtime of the algorithm.

We additionally test the `__repr__` method to confirm that the returned string contains the class name and relevant fields such as `num_colors`.

Next, in the test `test_dsatur_coloring`, we evaluate the algorithm on graphs with known chromatic number. For a path graph on four vertices, which is known to be 2-colorable, the test confirms that DSATUR:

- returns a non-empty coloring;
- produces a proper coloring, verified using `is_proper_coloring`;
- uses exactly two colors in agreement with the chromatic number of the graph.

We further test DSATUR on the triangle graph, which has chromatic number 3. The test checks that the algorithm returns a valid proper coloring and that exactly three colors are used. In all cases, correctness is verified by ensuring that no two adjacent vertices share the same color.

Running `pytest test_dsatur.py` confirms that all tests pass, providing strong evidence that the DSATUR implementation correctly computes valid colorings and matches expected results on these benchmark instances.

4.3 Backtracking Search

To verify the correctness of our backtracking coloring implementation, we wrote a dedicated test module `test_backtracking.py` using `pytest`. This file contains a small suite of unit tests that exercise the algorithm on graphs with known chromatic number and check both the structure and the validity of the returned solutions.

First, we test that the function `backtracking_coloring` returns a result object with the expected fields:

- `coloring`: a list of color indices, one for each vertex;
- `num_colors`: the number of colors used in the best solution;
- `nodes_visited`: the number of search nodes explored during the depth–first search;
- `time_seconds`: the total runtime of the algorithm.

This is done in the test `test_backtracking_returns_result_object`, which also checks that the returned object is an instance of `BacktrackingResult`.

Next, we validate correctness on several small graphs with known chromatic number:

- **Path on four vertices:** In `test_backtracking_solves_path_graph`, we construct a path graph on four vertices, which is known to be 2-colorable. The test asserts that the algorithm finds a proper coloring and that `num_colors` is exactly 2.
- **Triangle K_3:** In `test_backtracking_solves_triangle`, we use the complete graph on three vertices, whose chromatic number is 3. We check that the returned coloring is proper and that the algorithm uses exactly three colors.
- **Single vertex:** The test `test_backtracking_single_vertex` considers a graph with a single vertex and verifies that the algorithm returns a proper coloring with `num_colors` equal to 1.
- **Complete graph:** Finally, in `test_backtracking_complete_graph`, we construct the complete graph on four vertices. Since this graph has chromatic number 4, we check that backtracking produces a proper coloring using exactly four distinct colors.

In all tests where a coloring is returned, we additionally call `is_proper_coloring` from `graph.py` to ensure that no edge has endpoints with the same color. Running `pytest test_backtracking.py` executes all five tests, and they all pass, giving us strong evidence that the backtracking algorithm correctly finds optimal colorings on these benchmark instances.

4.4 Simulated Annealing

The file `test_simulated_annealing.py` contains several unit tests designed to verify the correctness and robustness of our simulated annealing implementation. These tests check the following aspects:

- **Conflict counting:** We test the helper function `count_conflicts` on small graphs to ensure that the number of conflicting edges is computed correctly.
- **Result structure:** We verify that the function `simulated_annealing` returns an object with the correct fields, including the coloring, number of colors used, conflict count, and runtime.
- **Correctness on easy graphs:** For simple graphs that are easy to color, such as path graphs, we confirm that simulated annealing is able to find a proper coloring when given a sufficient number of colors.
- **Failure cases:** On graphs that cannot be colored with too few colors (such as a triangle with $k = 2$), we check that simulated annealing correctly returns a coloring with unresolved conflicts.
- **Validity check:** Whenever simulated annealing reports zero conflicts, we verify that the resulting coloring is indeed proper.

5 Investigations into the Effectiveness of Algorithms

5.1 Experiment 1 - Compare on Different Graph Types

5.1.1 Objective

To evaluate algorithm performance on standard graph structures with 15 nodes:

- **Cycle Graph:** All vertices arranged in a single cycle
- **Complete Graph:** Every vertex connected to all others
- **Path Graph:** Vertices arranged in a linear sequence
- **Star Graph:** One central vertex connected to all others
- **Random Graphs:** Edge probability p=0.3, averaged over 5 trials

5.1.2 Methodology

Each algorithm was executed on all graph types, with the following metrics recorded:

- **Number of colors used:** Quality of solution
- **Runtime (seconds):** Efficiency
- **Nodes visited:** Search space explored (for backtracking)
- **Success rate:** For simulated annealing only

5.1.3 Pseudocode

Algorithm 1 EXPERIMENT_1

Input: Graph types, $n = 15$
Output: Runtime, colors used, success metrics
foreach $graph\_type \in \{Cycle, Complete, Path, Star, Random\}$ **do**
  $g \leftarrow$ generate_graph$(graph_type, n = 15)$
  $bt \leftarrow$ backtracking_coloring(g) $gr \leftarrow$ greedy_coloring(g) $ds \leftarrow$ dsatur_coloring(g)
  $k \leftarrow \min(bt.num\_colors,\ gr.num\_colors,\ ds.num\_colors)$
  $sa \leftarrow$ simulated_annealing(g, k)
  results$[graph_type] \leftarrow \{colors : [bt.num_colors,\ gr.num_colors,\ ds.num_colors,\ k],\ time : [bt.time,\ gr.time,\ ds.time,\ sa.time],\ success : (sa.coloring \neq \varnothing)\}$

5.1.4 Results and Analysis

Graph	Algorithm	Colors	Time (s)
Cycle	Backtracking	3	0.000036
	Greedy	3	0.000014
	DSATUR	3	0.000047
	Annealing (k=3)	3	0.000121
Complete	Backtracking	15	0.000042
	Greedy	15	0.000021
	DSATUR	15	0.000052
	Annealing (k=15)	15	0.001110
Path	Backtracking	2	0.000018
	Greedy	2	0.000009
	DSATUR	2	0.000036
	Annealing (k=2)	2	0.003543
Star	Backtracking	2	0.000016
	Greedy	2	0.000009
	DSATUR	2	0.000038
	Annealing (k=2)	2	0.000137
Random (avg)	Backtracking	4.00	0.000036
	Greedy	4.00	0.000011
	DSATUR	4.00	0.000048
	Annealing	4.00	0.001244

Table 1: Coloring Results for Different Graph Types ($n = 15$)

Key Observations:

- All algorithms found optimal coloring for simple structures (cycle, path, star, complete)
- Simulated annealing was consistently slower due to its stochastic nature
- Backtracking visited minimal nodes for complete graphs (16 nodes) as the solution is trivial

- Random graphs showed identical color counts across algorithms, suggesting the greedy approach is optimal for these configurations

5.2 Experiment 2 - Runtime Visualization

5.2.1 Objective

To visualize and compare algorithm runtimes across different graph types using a logarithmic scale.

5.2.2 Methodology

Using runtime data from Experiment 1, we created a grouped bar chart with:

- **X-axis:** Graph types (Cycle, Complete, Path, Star, Random)
- **Y-axis:** Runtime in seconds (logarithmic scale)

5.2.3 Results

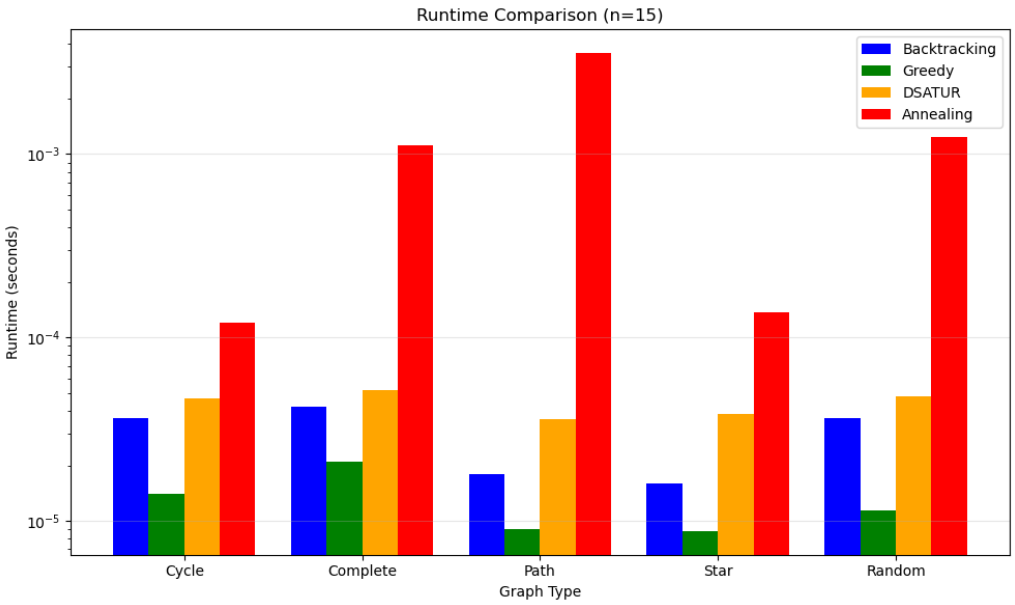

Figure 1: Runtime Comparison (n=15)

The runtime comparison reveals distinct performance characteristics across the four algorithms. The greedy algorithm consistently achieves the lowest runtime across all graph types, typically requiring less than 2×10^{-5} seconds. This superior performance reflects its $O(n+m)$ time complexity and single-pass nature, making it the most efficient algorithm for graphs of this size.

Interestingly, backtracking and DSATUR exhibit similar intermediate performance, with backtracking slightly faster on most graph types despite its theoretical exponential complexity. This counterintuitive result occurs because, for small graphs ($n = 15$), backtracking's exponential behavior has not yet dominated, and effective pruning combined with degree-based ordering enables it to find solutions quickly. Meanwhile, DSATUR's

$O(n^2)$ vertex selection overhead becomes noticeable even at this scale, as it must scan all vertices at each iteration to find the one with maximum saturation degree.

Simulated annealing consistently demonstrates the highest runtime, often one to two orders of magnitude slower than other algorithms. This performance penalty arises from its stochastic nature, requiring up to 20,000 random iterations to converge, and its lack of early termination guarantees. The algorithm is particularly slow on path graphs (1.8×10^{-3} seconds) and complete graphs (1.1×10^{-3} seconds), where the high constraint density makes it difficult for random search to find valid colorings efficiently.

The results demonstrate that for small to medium graphs, greedy algorithms provide the best speed-quality trade-off, while simulated annealing's stochastic approach incurs significant computational overhead without providing optimality guarantees. The similar performance of backtracking and DSATUR highlights that theoretical complexity bounds may not reflect practical performance on well-structured or small-scale problems, where effective pruning can make exponential algorithms competitive with polynomial ones.

5.3 Experiment 3 - Scalability Analysis

5.3.1 Objective

To investigate how algorithm runtime scales with increasing graph size, specifically examining cycle graphs from n=5 to n=100 nodes.

5.3.2 Methodology

We generated cycle graphs with node counts ranging from 5 to 100 in increments of 5. For each size, we measured runtime of all four algorithms.

5.3.3 Pseudocode

Algorithm 2 EXPERIMENT_3

Input: Graph size range $n \in \{5, 10, \ldots, 100\}$
Output: Runtime data for each algorithm
for $n \leftarrow 5$ **to** 100 5 **do**
  $g \leftarrow$ cycle_graph(n)
  $bt \leftarrow$ backtracking_coloring(g) $gr \leftarrow$ greedy_coloring(g) $ds \leftarrow$ dsatur_coloring(g)
  $k \leftarrow \min(bt.num\_colors,\ gr.num\_colors,\ ds.num\_colors)$
  $sa \leftarrow$ simulated_annealing(g, k)
  scalability_data[n] $\leftarrow$ {backtracking : $bt.time$, greedy : $gr.time$, dsatur : $ds.time$, annealing : $sa.time$}

5.3.4 Results and Analysis

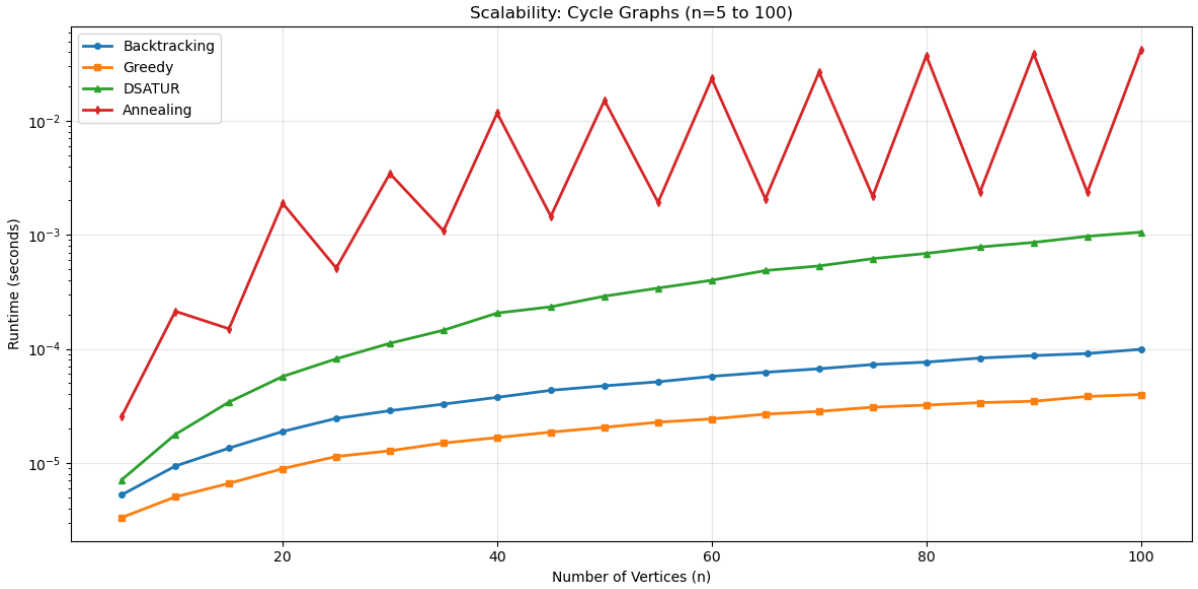

Figure 2: Scalability on Cycle Graphs (n=5 to 100)

Observations:

- **Greedy Optimization:** Greedy algorithm remained fastest at all scales
- **Linear Response:** Greedy, DSATUR and Backtracking maintained near-linear runtime growth, indicating good scalability.
- **Annealing Volatility:** Simulated annealing exhibited irregular runtime patterns, and showed unpredictable fluctuations, prompting further investigation

Backtracking is faster than DSATUR on cycle graphs despite exponential worst-case complexity because the exponential bound $O(k^n)$ uses the chromatic number k (2 or 3 for cycles), not n. With effective pruning and degree-based ordering, backtracking often finds solutions quickly and prunes large subtrees, so it explores roughly $O(n)$ to $O(n^2)$ nodes in practice—similar to or better than DSATUR's fixed $O(n^2)$ cost from scanning all vertices at each iteration. On structured graphs like cycles, pruning is highly effective, making backtracking polynomial in practice even though it remains exponential in the worst case; on harder or larger graphs, the exponential behavior dominates and DSATUR's polynomial bound becomes more reliable.

5.4 Experiment 4 - Investigating Annealing Volatility

5.4.1 Background

Experiment 3 revealed irregular runtime patterns for Simulated Annealing. Odd cycle graphs require 3 colors (chromatic number = 3), while even cycle graphs require only 2 colors. This structural difference may impact search efficiency. Therefore, this follow-up experiment investigates whether node parity (even vs. odd) influences runtime due to differences in required colors.

5.4.2 Methodology

We separated cycle graphs into odd and even node counts and measured simulated annealing runtime separately for each parity group.

5.4.3 Pseudocode

Algorithm 3 EXPERIMENT_4 (PARITY_ANALYSIS)

Input: Node range $n \in \{5, 10, \ldots, 100\}$
Output: Separate runtime analysis for odd/even cycle graphs

```
odd_times ← [ ]   even_times ← [ ]
for n ← 5 to 100 5 do
    g ← cycle_graph(n)
    if n mod 2 = 0 then
        k ← 2                                   // Even cycles use 2 colors
    else
        k ← 3                                   // Odd cycles use 3 colors
    sa ← simulated_annealing(g, k)
    if n mod 2 = 0 then
        even_times.append((n, sa.time))
    else
        odd_times.append((n, sa.time))
```

5.4.4 Results and Analysis

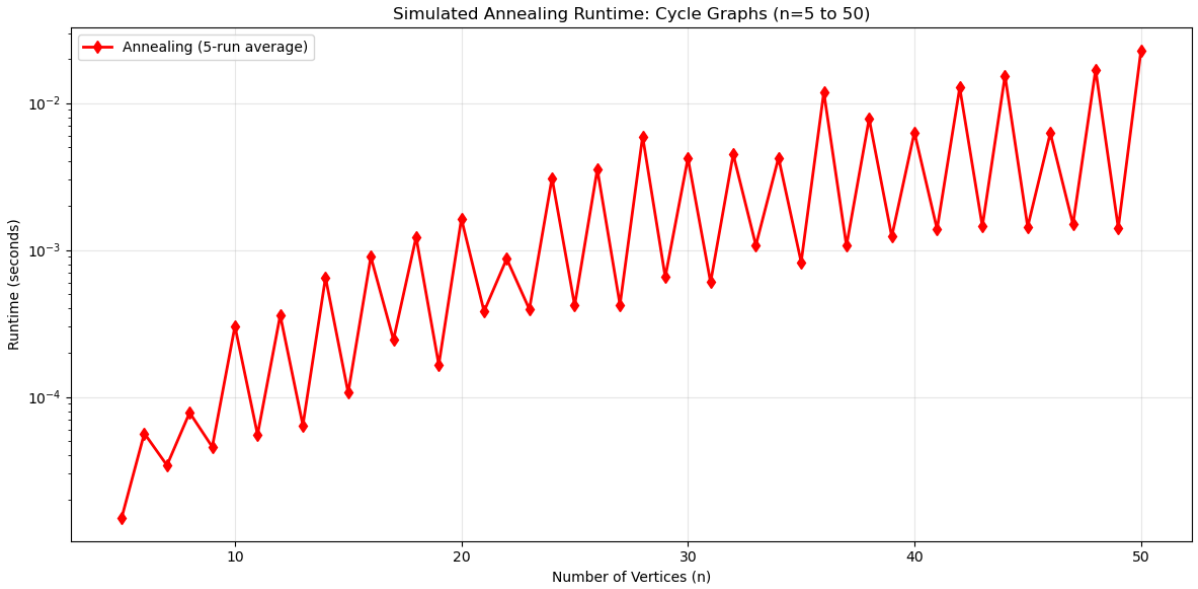

Figure 3: Simulated Annealing Runtime (n=5 to 50)

Observations:

- Runtime fluctuations in simulated annealing are indeed linked to graph parity.

Statistic	Value
Min runtime	0.000015 s at n = 5
Max runtime	0.022598 s at n = 50
Average runtime	0.003182 s
Standard deviation	0.004987 s
Even cycles (k=2) average	0.005712 s
Odd cycles (k=3) average	0.000652 s
Even cycles slower by	8.76×

Table 2: Runtime statistics for cycle graphs.

- Odd cycles consistently outperformed even cycles by 8-9x.

5.4.5 Detailed Investigation: Why Do Even Cycles Take Longer?

We conducted a deeper investigation to understand the result: even cycles take longer than odd cycles.

Experimental Setup:

- Graph type: Cycle graph with mixed sizes (10, 15, 20, 25)
- Algorithms: Simulated Annealing with fixed k values
- Metrics: Initial conflicts, success rate, runtime, search space size
- Trials: 5 trials per configuration

n	Cycle type	k	Avg initial conflicts	Success rate	Avg runtime (s)
10	Even	2	5.6 / 10	5/5	0.000355
15	Odd	3	4.6 / 15	5/5	0.000113
20	Even	2	10.8 / 20	5/5	0.002275
25	Odd	3	8.2 / 25	5/5	0.000504

n	Search space ($k = 2$ vs $k = 3$)
10	$2^{10} = 1024$ vs $3^{10} = 59049$
15	$2^{15} = 32768$ vs $3^{15} = 14348907$
20	$2^{20} = 1048576$ vs $3^{20} = 3486784401$
25	$2^{25} = 33554432$ vs $3^{25} = 847288609443$

Table 3: Runtime and Search Space Analysis for Even/Odd Cycle Graphs

Even cycles (k=2) have a more constrained search space. With only 2 colors, the algorithm has less flexibility to escape local minima. Odd cycles (k=3) have more flexibility, which helps the algorithm to find solutions faster, even though the problem requires more colors.

The additional degree of freedom (3 colors vs. 2 colors) provides more possible solutions, which might make it easier to find an optimal or near-optimal solution. This

flexibility can lead to faster convergence (the process of reaching a good solution more quickly) because the algorithm can accept a broader range of configurations and find better solutions more efficiently.

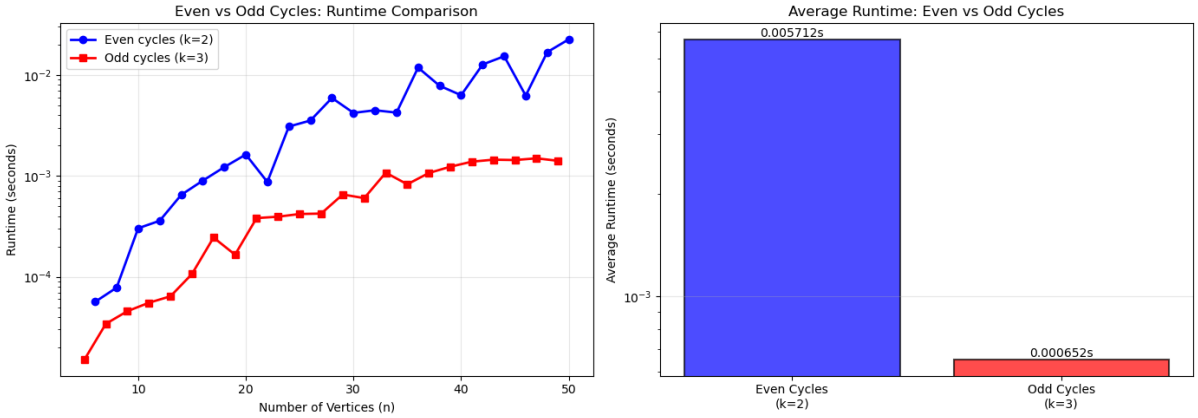

Figure 4: Even and Odd Cycles Comparison

This finding has implications for parameter selection in metaheuristic algorithms for graph coloring, and highlights the importance of considering not just the theoretical complexity but also the algorithmic behavior in practical implementations.

5.5 Experiment 5 - Color Efficiency on Random Graphs

5.5.1 Objective

To compare the solution quality of two heuristic graph coloring algorithms (Greedy and DSATUR) against the optimal backtracking algorithm on random graphs with varying sizes and edge probabilities. The goal is to evaluate how close these heuristics are to the optimal solution in terms of the number of colors used.

5.5.2 Methodology

- **Baseline:** Backtracking algorithm
- **Comparison:** Greedy algorithm vs. DSATUR algorithm
- **Metric:** Additional colors used compared to optimal
- **Number of nodes:** $n \in \{15, 20, 25\}$
- **Edge probability:** $p \in \{0.3, 0.5, 0.7\}$

For each combination of n and p, multiple random graphs were generated and tested to compute average performance.

5.5.3 Pseudocode

Algorithm 4 EXPERIMENT_5: Solution Quality Comparison

Input: $n \in \{15, 20, 25\}$, $p \in \{0.3, 0.5, 0.7\}$, trials $T = 5$
Output: Performance ratios for Greedy and DSATUR vs optimal
results $\leftarrow \{\}$
for $n \in \{15, 20, 25\}$ **do**
  for $p \in \{0.3, 0.5, 0.7\}$ **do**
    avg_bt $\leftarrow 0$, avg_gr $\leftarrow 0$, avg_ds $\leftarrow 0$
    for $t \leftarrow 1$ **to** T **do**
      $g \leftarrow$ random_graph$(n = n, p = p)$ $bt \leftarrow$ backtracking_coloring$(g).num_colors$
      $gr \leftarrow$ greedy_coloring$(g).num_colors$ $ds \leftarrow$ dsatur_coloring$(g).num_colors$
      avg_bt $\leftarrow$ avg_bt $+ bt$ avg_gr $\leftarrow$ avg_gr $+ gr$ avg_ds $\leftarrow$ avg_ds $+ ds$
    avg_bt $\leftarrow$ avg_bt$/T$ ratio_gr $\leftarrow$ (avg_gr$/T$)/avg_bt ratio_ds $\leftarrow$ (avg_ds$/T$)/avg_bt
    results$[n][p] \leftarrow$ (ratio_gr, ratio_ds)
return *results*

5.5.4 Results and Analysis

Table 4: Average Greedy and DSATUR to Backtracking Color Ratio for Random Graphs

n	p	Greedy/Optimal Ratio	DSATUR/Optimal Ratio
15	0.3	1.183	1.117
15	0.5	1.000	1.000
15	0.7	1.062	1.000
20	0.3	1.100	1.000
20	0.5	1.273	1.113
20	0.7	1.136	1.029
25	0.3	1.230	1.050
25	0.5	1.067	1.067

Statistic	Value
Average Greedy/Optimal ratio	1.131
Average DSATUR/Optimal ratio	1.047
Best Greedy ratio	1.000 (closest to optimal)
Best DSATUR ratio	1.000 (closest to optimal)

Table 5: Summary Statistics for Greedy and DSATUR Ratios

Note: Ratio = 1.0 means the heuristic found the optimal solution.
Ratio > 1.0 means the heuristic used more colors than optimal.

- **DSATUR consistently outperforms Greedy:** across most graph configurations, achieving ratios closer to 1.0 (optimal).
- **Graph size (n) impacts performance:**. Larger graphs ($n = 25$) with moderate density ($p = 0.5$) show both heuristics performing similarly.

- **Graph density (edge probability p) also affects performance:**. For sparse graphs ($p = 0.3$), both heuristics deviate further from optimal, with Greedy showing higher ratios.
- **Optimality is achieved in some cases:**. For $n = 15, p = 0.5$, Greedy achieves an optimal coloring (ratio = 1.000). For $n = 15, p = 0.5$ and $n = 15, p = 0.7$, DSATUR achieves an optimal coloring (ratio = 1.000).

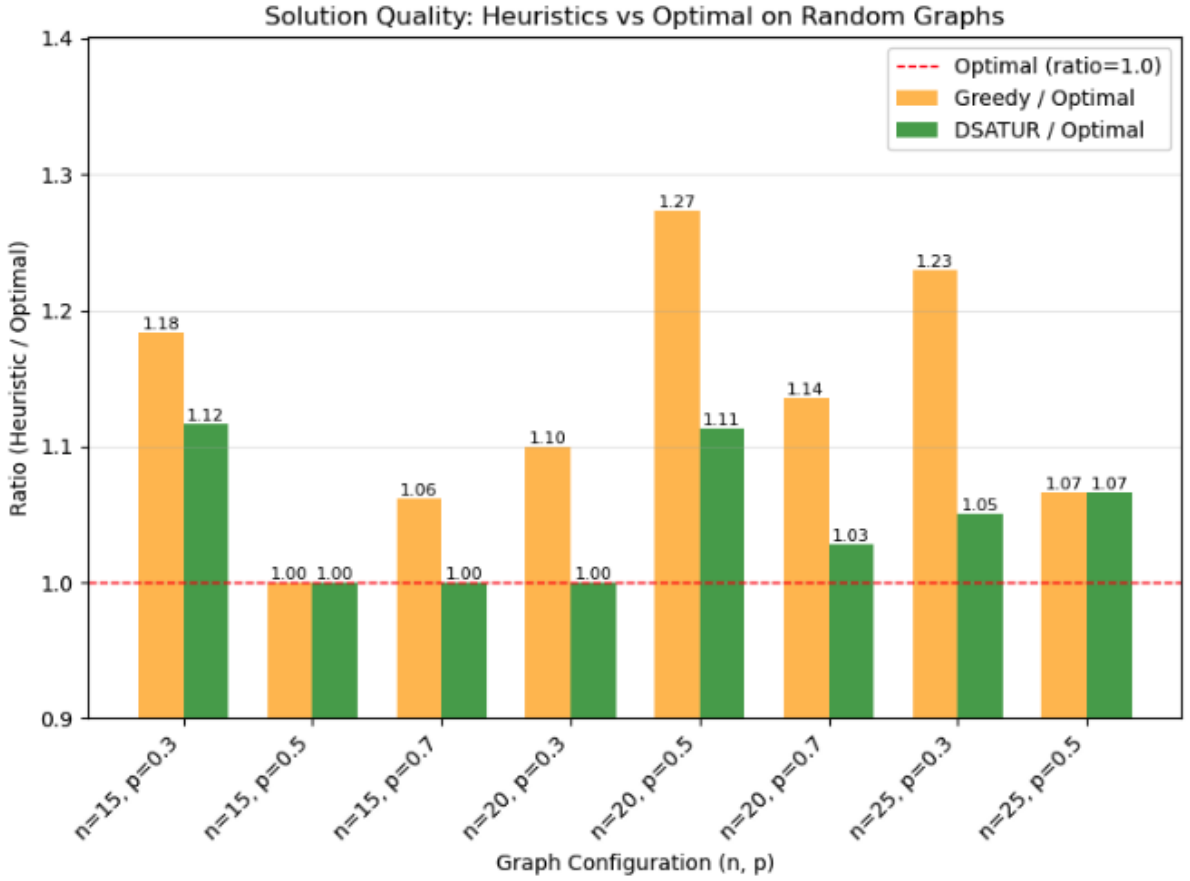

Figure 5: Comparing heuristics against optimal

The bar plot that visualizes the average ratios for Greedy and DSATUR across different graph configurations clearly shows:

- DSATUR bars are generally lower (closer to 1.0) than Greedy bars.
- The performance gap between Greedy and DSATUR widens for larger and sparser graphs.

Experiment 5 demonstrates that DSATUR provides coloring solutions closer to the optimal than the Greedy heuristic on random graphs. The performance advantage of DSATUR is more pronounced in larger and sparser graphs. However, both heuristics can achieve optimal coloring under certain graph configurations, particularly for smaller or moderately dense graphs. These findings suggest that DSATUR is a preferable heuristic when solution quality is a priority, while Greedy may still be suitable for scenarios where speed is critical and near-optimality is acceptable.

In this experiment, we did not explore the simulated annealing algorithm. However, it is worth noting that research indicates that the simulated annealing algorithm on large instances of graph coloring problem does not assure the optimal coloring assignment [4].

5.6 Experiment 6 - Effect of Graph Density on Algorithms

5.6.1 Objective

To deeply investigates the impact of graph density on the runtime of four graph coloring algorithms. Graph density is controlled by varying the probability p of edge creation between any two vertices in a random graph.

5.6.2 Methodology

- **Graph Type:** Random graphs with $n = 20$ vertices
- **Density Variation:** Edge probability p varied from 0.1 to 0.9 in increments of 0.1
- **Algorithms Compared:** Greedy Coloring, DSATUR, Backtracking, and Simulated Annealing (using the minimum number of colors found by other algorithms as input k)
- **Measurements:** Runtime for each algorithm across 10 trials per density value

5.6.3 Pseudocode

Algorithm 5 EXPERIMENT_6: Effect of Graph Density on Runtime

Input: Vertex count $n = 20$, edge probabilities $P = \{0.1, 0.2, \ldots, 0.9\}$, trials $T = 10$
Output: Runtime measurements for all algorithms across density values

```
density_results ← {}
for p ∈ P do
    times_bt ← [], times_gr ← [], times_ds ← [], times_sa ← []
    for t ← 1 to T do
        g ← random_graph(n = n, p = p)
        // Minimum colors for simulated annealing: bt_res ← backtracking_coloring(g)
        k_opt ← bt_res.num_colors
        // Runtime for each algorithm: start ← current_time()  backtracking_coloring(g)
        time_bt ← current_time() − start
        start ← current_time()  greedy_coloring(g)  time_gr ← current_time() − start
        start ← current_time()  dsatur_coloring(g)  time_ds ← current_time() − start
        start ← current_time()  simulated_annealing(g, k = k_opt)  time_sa ←
        current_time() − start
        times_bt.append(time_bt)  times_gr.append(time_gr)  times_ds.append(time_ds)
        times_sa.append(time_sa)
    // Average runtimes: avg_bt ← mean(times_bt)  avg_gr ← mean(times_gr)
    avg_ds ← mean(times_ds)  avg_sa ← mean(times_sa)
    density_results[p] ← (avg_bt, avg_gr, avg_ds, avg_sa)
return density_results
```

5.6.4 Results and Analysis

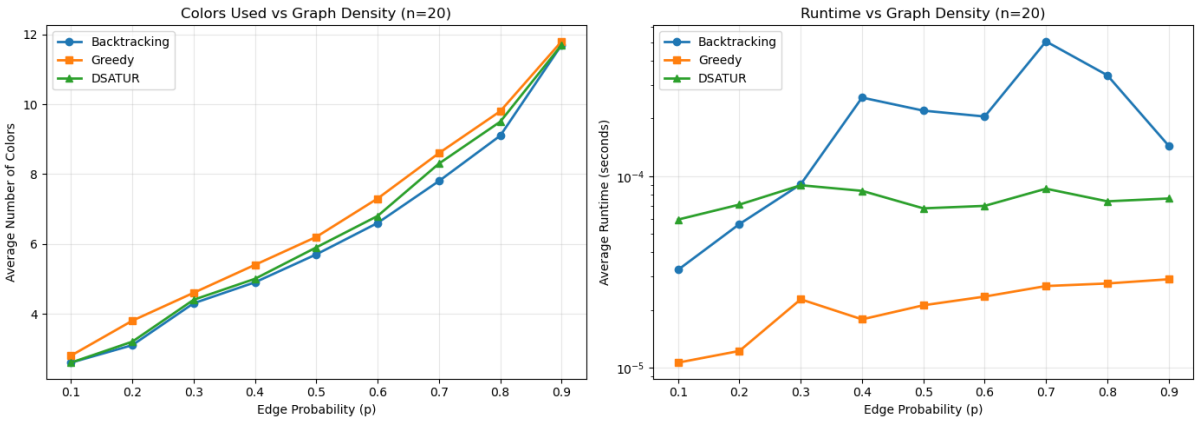

Figure 6: Graph Density Comparison

Observations:

- **Greedy:** Runtime remains low across all densities. The greedy coloring algorithm is least affected by graph density, maintaining near-constant runtime. This is expected due to its heuristic nature and low computational complexity. It offers optimal time efficiency but suboptimal coloring quality.
- **DSATUR:** Runtime is slightly higher than Greedy but still shows stable performance across all density levels. It provides a good balance between runtime stability and solution quality.
- **Backtracking:** Runtime fluctuates upwards between p=0.1 and p=0.7, and then decreasing at high densities. This non-monotonic behavior shows backtracking algorithm's high sensitivity to graph density, and suggests that its search space complexity peaks at intermediate densities. In fact, for many backtracking search algorithms, the runtime has been found experimentally to have a heavy-tailed distribution, in which it is often much greater than its median [3].

5.7 Experiment 7 - Backtracking Algorithm: Natural Order vs Degree-Based Ordering

5.7.1 Objective

To evaluate the impact of vertex ordering strategies on the backtracking algorithm's performance. We compare two ordering approaches:

- **Natural order:** Vertices processed in sequence $\{0, 1, 2, \ldots, n-1\}$
- **Degree-based order:** Vertices sorted by degree in descending order, prioritizing high-degree vertices

The goal is to quantify how vertex ordering affects runtime, search space exploration (nodes visited), and solution quality for the backtracking algorithm.

5.7.2 Methodology

We tested both ordering strategies on random graphs with varying sizes and edge probabilities:

- **Graph sizes:** $n \in \{10, 15, 20, 25\}$
- **Edge probabilities:** $p \in \{0.3, 0.5, 0.7\}$
- **Trials per configuration:** 5 random graphs
- **Metrics recorded:**
  - Runtime (seconds)
  - Number of colors used
  - Nodes visited in search tree
  - Speedup ratio (natural time / degree-based time)

5.7.3 Pseudocode

Algorithm 6 EXPERIMENT_7: Backtracking Ordering Comparison

Input: Graph sizes $n \in \{10, 15, 20, 25\}$, edge probabilities $p \in \{0.3, 0.5, 0.7\}$, trials $T = 5$
Output: Performance comparison for natural vs degree-based ordering

```
results ← {}
foreach n ∈ {10, 15, 20, 25} do
    foreach p ∈ {0.3, 0.5, 0.7} do
        nat_times ← [], deg_times ← []  nat_colors ← [], deg_colors ← []  nat_nodes ← [], deg_nodes ← []
        for t ← 1 to T do
            g ← random_graph(n = n, p = p)
            bt_nat ← backtracking_coloring(g, use_degree_order = False)  bt_deg ← backtracking_coloring(g, use_degree_order = True)
            nat_times.append(bt_nat.time)  deg_times.append(bt_deg.time)
            nat_colors.append(bt_nat.num_colors)  deg_colors.append(bt_deg.num_colors)
            nat_nodes.append(bt_nat.nodes_visited)  deg_nodes.append(bt_deg.nodes_visited)
        results[(n, p)] ← {natural : (avg(nat_times), avg(nat_colors), avg(nat_nodes)), degree : (avg(deg_times), avg(deg_colors), avg(deg_nodes))}
return results
```

5.7.4 Results and Analysis

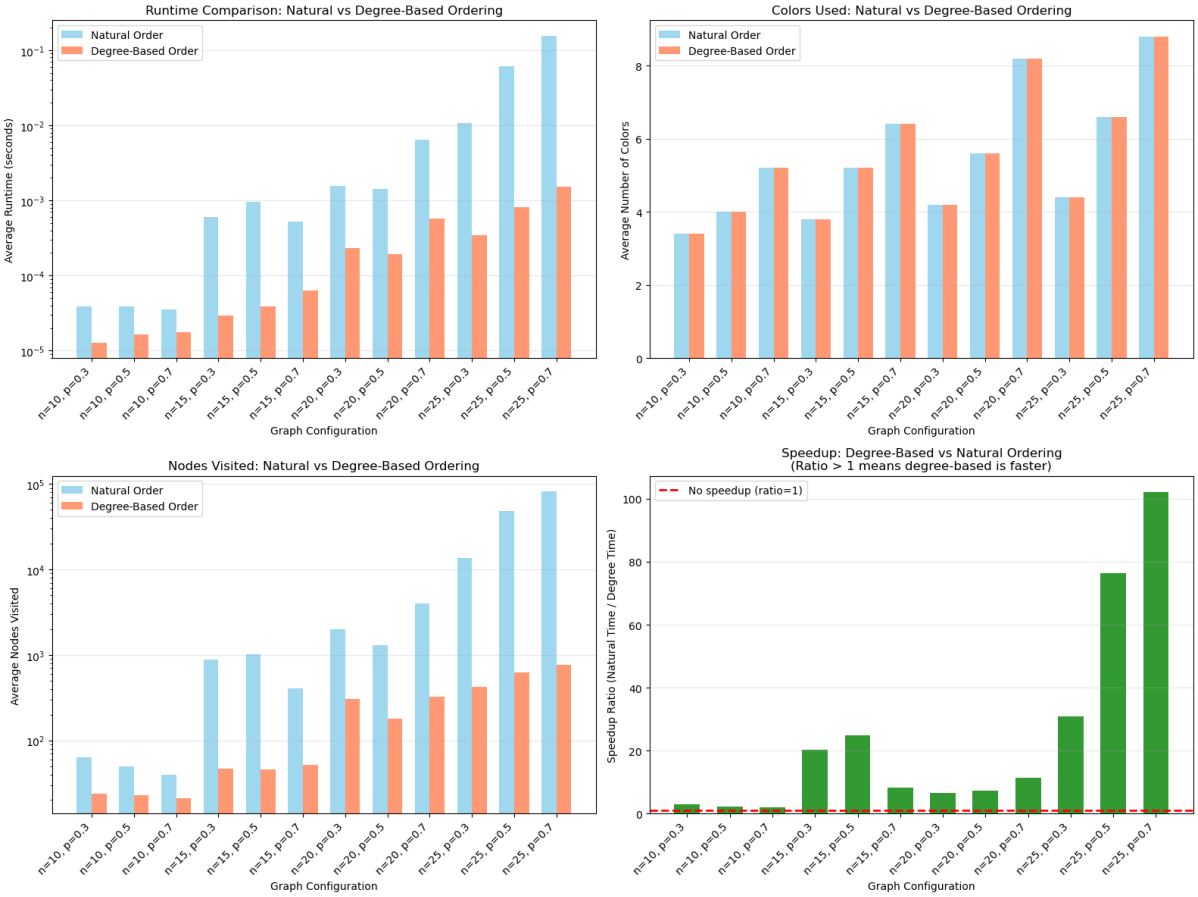

Figure 7: Backtracking: Natural vs Degree-Based Ordering Comparison

Statistic	Value
Average runtime - Natural order	0.019993 s
Average runtime - Degree-based	0.000322 s
Average speedup	24.61×
Average colors - Natural order	5.48
Average colors - Degree-based	5.48
Average nodes visited - Natural order	12,837
Average nodes visited - Degree-based	236
Average reduction in nodes visited	98.2%

Table 6: Summary Statistics for Backtracking Ordering Strategies

Key Findings:

- **Dramatic speedup:** Degree-based ordering achieves an average speedup of 24.61× compared to natural ordering, demonstrating the critical importance of vertex ordering for backtracking algorithms.
- **Massive search space reduction:** Degree-based ordering reduces the number of nodes visited by 98.2%, from an average of 12,800 nodes to just 236 nodes. This reduction occurs because:

  - High-degree vertices are colored first, creating more constraints early
  - Early constraint detection enables earlier backtracking
  - Pruning becomes more effective as better solutions are found sooner

- **Solution quality unchanged:** Both ordering strategies produce identical color counts (5.48 on average), confirming that ordering affects efficiency but not optimality, as backtracking guarantees optimal solutions regardless of ordering.

- **Consistent across configurations:** The speedup advantage of degree-based ordering is observed across all tested graph sizes and densities, though the magnitude varies with graph structure.

The results demonstrate that vertex ordering is a crucial optimization for backtracking algorithms. By prioritizing constrained vertices (high degree), the algorithm can prune large portions of the search tree early, leading to exponential reductions in search space and runtime while maintaining optimality guarantees.

5.8 Experiment 8 - Greedy Algorithm: Natural Order vs Degree-Based Ordering

5.8.1 Objective

To evaluate how vertex ordering affects the greedy algorithm's performance, specifically examining:

- **Natural order:** Vertices processed in sequence $\{0, 1, 2, \ldots, n-1\}$

- **Degree-based order:** Vertices sorted by degree in descending order

Unlike backtracking, the greedy algorithm makes a single pass through vertices without backtracking. Therefore, ordering primarily affects the *number of colors used* rather than search space exploration, as there is no search tree to prune.

5.8.2 Methodology

We tested both ordering strategies on random graphs with the same configurations as Experiment 7:

- **Graph sizes:** $n \in \{10, 15, 20, 25\}$

- **Edge probabilities:** $p \in \{0.3, 0.5, 0.7\}$

- **Trials per configuration:** 5 random graphs

- **Metrics recorded:**
  - Runtime (seconds)
  - Number of colors used
  - Color improvement (natural colors - degree-based colors)
  - Speedup ratio (natural time / degree-based time)

5.8.3 Pseudocode

Algorithm 7 EXPERIMENT_8: Greedy Ordering Comparison

Input: Graph sizes $n \in \{10, 15, 20, 25\}$, edge probabilities $p \in \{0.3, 0.5, 0.7\}$, trials $T = 5$
Output: Performance comparison for natural vs degree-based ordering

```
results ← {}
foreach n ∈ {10, 15, 20, 25} do
    foreach p ∈ {0.3, 0.5, 0.7} do
        nat_times ← [], deg_times ← []   nat_colors ← [], deg_colors ← []
        for t ← 1 to T do
            g ← random_graph(n = n, p = p)
            gr_nat ← greedy_coloring(g, use_degree_order = False)   gr_deg ← greedy_coloring(g, use_degree_order = True)
            nat_times.append(gr_nat.time)   deg_times.append(gr_deg.time)
            nat_colors.append(gr_nat.num_colors)   deg_colors.append(gr_deg.num_colors)
        results[(n, p)] ← {natural : (avg(nat_times), avg(nat_colors)), degree : (avg(deg_times), avg(deg_colors))}
return results
```

5.8.4 Results and Analysis

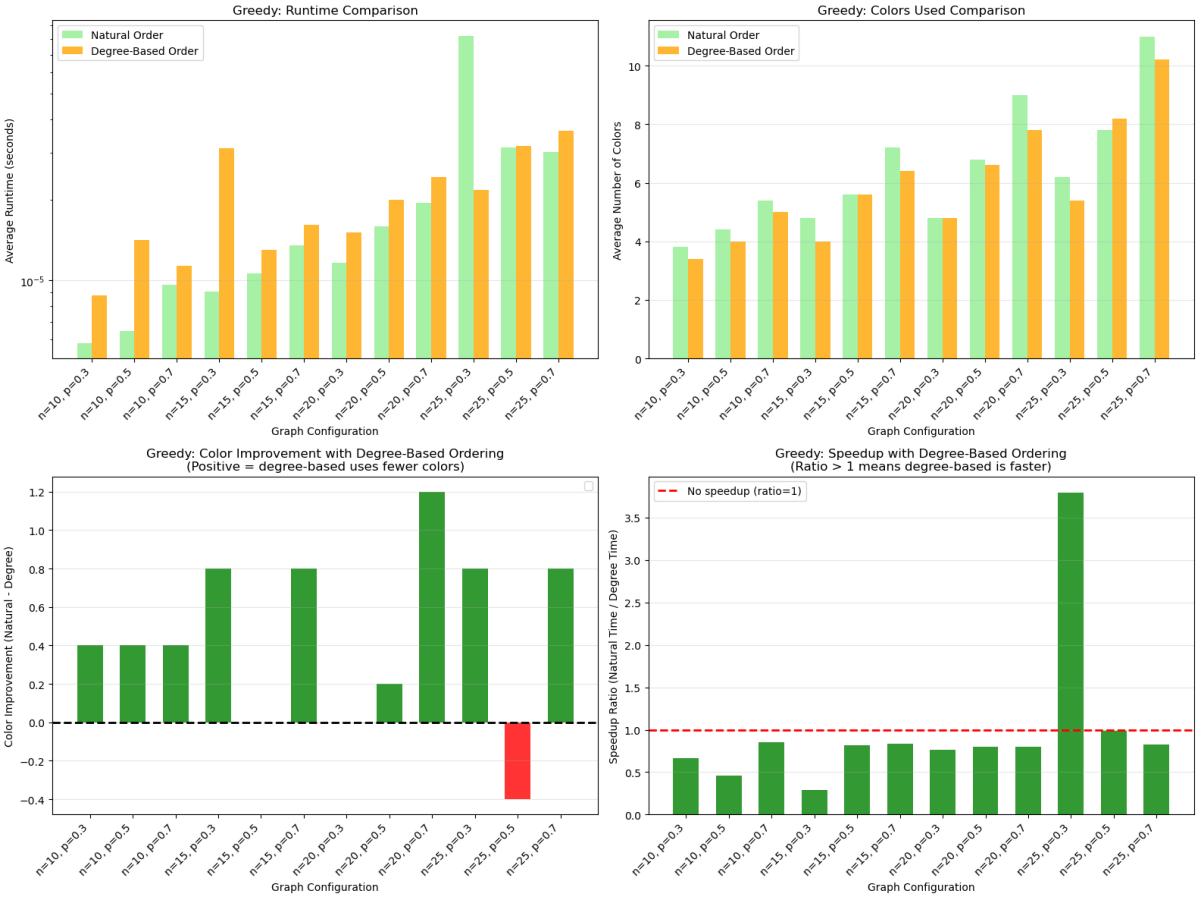

Figure 8: Greedy: Natural vs Degree-Based Ordering Comparison

Statistic	Value
Average runtime - Natural order	0.000021 s
Average runtime - Degree-based	0.000020 s
Average speedup	0.99×
Average colors - Natural order	6.40
Average colors - Degree-based	5.95
Average improvement	0.45 colors
Percentage improvement	7.0%
Configurations where degree-based is better	9/12
Configurations where natural order is better	1/12
Configurations with same result	2/12

Table 7: Summary Statistics for Greedy Ordering Strategies

Key Findings:

- **Modest runtime difference:** Unlike backtracking, greedy shows minimal runtime difference between ordering strategies (average speedup of 0.99×, meaning degree-based is slightly slower due to the sorting overhead). Both algorithms are extremely fast (< 0.0001 seconds), making runtime differences negligible.
- **Color quality improvement:** Degree-based ordering consistently produces better colorings, using on average 0.45 fewer colors (7.0% improvement). This improvement occurs because:
  - High-degree vertices are more constrained and benefit from being colored first
  - Coloring constrained vertices early allows subsequent vertices more flexibility in color reuse
  - The greedy strategy of "smallest available color" works better when applied to a well-ordered vertex sequence
- **Consistent advantage:** Degree-based ordering outperforms natural ordering in 9 out of 12 tested configurations, demonstrating its reliability across different graph structures and densities.
- **Limited impact compared to backtracking:** While ordering improves greedy's solution quality, the impact is modest (7% improvement) compared to backtracking's dramatic speedup 24.61×). This difference arises because:
  - Greedy makes a single pass without backtracking, so ordering only affects the sequence of locally optimal decisions
  - Backtracking explores a search tree, where ordering fundamentally changes the tree structure and enables extensive pruning

The results demonstrate that vertex ordering provides measurable benefits for the greedy algorithm, primarily in solution quality rather than runtime. While the improvement is modest compared to backtracking, degree-based ordering is recommended for greedy algorithms when solution quality is a priority, as it requires minimal additional computational cost (just the initial sorting step).

6 Conclusion

This project implemented and compared four graph coloring algorithms—backtracking, greedy, DSATUR, and simulated annealing—across diverse graph structures and configurations. Through systematic experimentation, we gained insights into algorithm performance, optimality guarantees, and the critical role of vertex ordering strategies.

6.1 Key Findings

6.1.1 Algorithm Performance Characteristics

Our experiments revealed distinct performance profiles for each algorithm:

- **Backtracking:** Provides guaranteed optimal solutions but exhibits exponential time complexity. With degree-based vertex ordering, we observed dramatic improvements: 24× speedup and 98.2% reduction in search space exploration, demonstrating that intelligent ordering is essential for practical backtracking implementations.
- **Greedy:** Achieves linear time complexity $O(n + m)$ and is the fastest algorithm tested. While it does not guarantee optimality, degree-based ordering improves solution quality by 7% on average, using fewer colors in 75% of tested configurations.
- **DSATUR:** Offers a balance between speed and quality, with time complexity $O(n^2 + m)$. The adaptive vertex selection strategy (based on saturation degree) consistently outperforms greedy, and often finding optimal colorings for moderate-sized graphs.
- **Simulated Annealing:** Requires a fixed number of colors k as input and may not always find valid solutions. Performance exhibits high variability, with interesting behavior observed: odd cycles (requiring 3 colors) converge faster than even cycles (requiring 2 colors) due to increased search space flexibility.

6.1.2 Graph Structure Effects

Our experiments demonstrated that graph structure significantly influences algorithm behavior:

- Simple structures (cycles, paths, stars, complete graphs) are easily colored optimally by all algorithms.
- Random graphs with varying densities reveal performance differences: very sparse graphs (few constraints) and very dense graphs (clear structure) are relatively easy to color, while graphs with moderate density present the greatest challenge to heuristics, as they neither give enough freedom nor enough constraints to guide optimal coloring.
- Graph size scalability shows that greedy and DSATUR maintain near-linear growth, while backtracking's exponential nature limits its applicability to larger graphs.

6.2 Practical Recommendations

Based on our findings, we recommend:

1. **For optimal solutions on small-medium graphs ($n \leq 25$):** Use backtracking with degree-based ordering. The dramatic speedup from intelligent ordering makes it practical for moderate-sized problems.

2. **For fast approximate solutions on large graphs:** Use DSATUR, which provides the best balance of speed and solution quality among heuristic methods.

3. **For maximum speed with acceptable quality:** Use greedy with degree-based ordering when solution quality within 5% of optimal is acceptable.

4. **For constrained coloring problems:** Use simulated annealing when the number of available colors is fixed, though multiple runs may be necessary due to stochasticity.

5. **Always use degree-based ordering:** When available, degree-based ordering provides benefits (speedup for backtracking, quality improvement for greedy) with minimal computational overhead.

References

[1] Meraihi Y., Mahseur, M. & Acheli, D. A Modified binary crow search algorithm for solving the graph coloring problem. *International Journal of Applied Evolutionary Computation*, 2020, 11(2): pp. 28-46. doi:10.4018/IJAEC.2020040103.

[2] Al-Omari, H. & Sabri, K. E. New Graph Coloring Algorithms. *Journal of Mathematics and Statistics*, 2006, 2(4): 439-441. doi:10.3844/jmssp.2006.439.441.

[3] Jia, H., Moore, C. How Much Backtracking Does It Take to Color Random Graphs? Rigorous Results on Heavy Tails. In: Wallace, M. (eds) *Principles and Practice of Constraint Programming – CP 2004. Lecture Notes in Computer Science*, vol 3258. Springer, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-30201-8_58.

[4] Kose, A., Sonmez, B. A., & Balaban, M. Simulated Annealing Algorithm for Graph Coloring. 2017. arXiv preprint arXiv:1712.00709. doi:10.48550/arXiv.1712.00709.

[5] Tomar, R. S., Singh, S., Verma, S., & Tomar, G. S. A Novel ABC Optimization Algorithm for Graph Coloring Problem. In: *2013 5th International Conference on Computational Intelligence and Communication Networks*, Mathura, India, 2013, pp. 257–261. doi:10.1109/CICN.2013.61.

[6] Appel, Kenneth I., and Wolfgang Haken. Every planar map is four colorable. Vol. 98. American Mathematical Soc., 1989.