Graph Coloring: An Analysis of the Trade-off Between Heuristics and Optimality

Project Number: 10

Team Members:
Aakarsh Mishra (2024111007)
Abhijit Suhas (2024101064)
Rachit Mehta (2024101089)
Jenik Gajera (2024113026)
Divyanshu Giri (2024114009)

December 1, 2025

Abstract

This project presents a comprehensive comparative analysis of algorithms for the **Graph Coloring Problem (GCP)**, a fundamental NP-hard challenge in combinatorial optimization. We implemented and evaluated six distinct algorithms representing the spectrum from efficient heuristics to exact solvers: **Welsh-Powell** and **DSatur** (greedy heuristics), **Simulated Annealing** (SA) and **Tabu Search** (metaheuristics), a **Genetic Algorithm** (evolutionary), and an **Exact Solver** (dynamic programming).

The algorithms were benchmarked against 79 standard **DIMACS** instances and 55 procedurally generated synthetic graphs (including Erdős-Rényi and Barabási-Albert models). Our empirical results demonstrate a stark trade-off between computational tractability and solution quality. While **Welsh-Powell** proved to be the fastest ($O(n^2)$), it consistently overestimated the chromatic number. **DSatur** offered the best baseline balance, while **Simulated Annealing** achieved near-optimal colorings on structured graphs, albeit at 10-100x the runtime of greedy methods. Conversely, the **Genetic Algorithm** demonstrated a critical performance failure, exhibiting a median slowdown of over 700x relative to DSatur with negligible gains in solution quality. **Tabu Search** excelled on structured instances but suffered from stagnation on dense random graphs. This report details the theoretical foundations, implementation challenges, and quantitative performance of these approaches.

Contents

1 Introduction

1.1 Problem Definition

The **Graph Coloring Problem (GCP)** is a classical problem in graph theory and combinatorial optimization. Given an undirected graph $G = (V, E)$, the objective is to assign a color $c(v)$ to each vertex $v \in V$ such that no two adjacent vertices share the same color. Formally, we seek a mapping $c : V \rightarrow \{1, 2, \ldots, k\}$ such that:

$$\forall (u, v) \in E, \quad c(u) \neq c(v) \tag{1}$$

The primary goal is to minimize k, the number of colors used. The smallest integer k for which a valid coloring exists is known as the **chromatic number**, denoted as $\chi(G)$. Determining $\chi(G)$ is **NP-hard** for general graphs. Consequently, finding an optimal solution is computationally intractable for large-scale instances, necessitating the use of approximation algorithms and heuristics that trade optimality for execution speed.

1.2 Real-World Relevance

Graph coloring serves as a powerful abstraction for a wide class of resource allocation and conflict resolution problems in computer science and operations research:

- **Timetabling and Scheduling:** Mapping exams or classes (vertices) to time slots (colors) such that no students have conflicting schedules (edges).

- **Register Allocation:** In compiler optimization, assigning a limited number of CPU registers (colors) to variables (vertices) that are simultaneously active (edges represent liveness interference).

- **Frequency Assignment:** Assigning radio frequencies to transmitters to prevent signal interference between geographically adjacent towers.

- **Map Labeling:** Ensuring adjacent regions on a map are distinct.

1.3 Project Objectives

The primary objective of this project is to explore the "Heuristic-to-Optimal" spectrum by implementing and analyzing diverse algorithmic strategies. Specifically, we aim to:

1. **Implement a diverse algorithmic suite:**

    - *Greedy Heuristics:* Welsh-Powell and DSatur, focusing on speed and scalability.
    - *Metaheuristics:* Simulated Annealing, Tabu Search, and Genetic Algorithms, focusing on escaping local minima to approach $\chi(G)$.
    - *Exact Methods:* A Dynamic Programming solver to establish ground truth for small instances.

2. **Benchmark Performance:** Evaluate these algorithms on widely recognized DIMACS benchmarks and a custom suite of Synthetic Graphs (Planar, Bipartite, Random) to assess robustness across different graph topologies.

3. **Analyze Trade-offs:** Quantify the relationship between runtime (efficiency) and the number of colors used (efficacy).

4. **Critique Failure Modes:** Specifically analyze why certain metaheuristics (e.g., Genetic Algorithms) may fail to scale effectively compared to simpler local search methods.

2 Algorithm Descriptions

This section provides a rigorous, theoretically grounded explanation of each algorithm implemented in the study. For every algorithm, we present:

- A clear, detailed theoretical description,

- A formal correctness statement with justification,

- A complete asymptotic analysis (time and space complexity).

Throughout, let $G = (V, E)$ be a graph with $|V| = n$ vertices and $|E| = m$ edges. A coloring is a function $c : V \to \mathbb{Z}_{\geq 0}$, and it is *proper* if $c(u) \neq c(v)$ for all edges $(u, v) \in E$. The chromatic number $\chi(G)$ is the smallest number of colors required to produce a proper coloring.

2.1 Welsh–Powell Algorithm

2.1.1 Theoretical Description

The Welsh–Powell algorithm is a static greedy heuristic based on ordering vertices by degree. The algorithm proceeds as follows:

1. Sort the vertices $v_1, \ldots, v_n$ in nonincreasing order of degree:

$$d(v_1) \geq d(v_2) \geq \cdots \geq d(v_n).$$

2. Initialize all vertices as uncolored. For color index $t = 1, 2, \ldots$:

   (a) Select the first uncolored vertex in the ordering and assign it color t.

   (b) Scan the remaining vertices and assign color t to any uncolored vertex that has no neighbor already assigned color t.

3. Repeat until every vertex is colored.

2.1.2 Correctness

Claim. The Welsh–Powell algorithm always produces a proper coloring.

   Proof. When assigning color t to a vertex v, the algorithm checks all colored neighbors of v and assigns color t only if none of them already has color t. Thus no edge ever becomes monochromatic during the process. The invariant that all colored vertices form a proper partial coloring is preserved inductively, and therefore the final result is a valid proper coloring. $\square$

2.1.3 Asymptotic Analysis

Time complexity. Sorting requires $O(n \log n)$. For each color class, the algorithm scans remaining vertices and checks adjacency. Using adjacency lists, this yields

$$O(n^2 + nm)$$

in the worst case. In practice, the algorithm is commonly implemented using adjacency matrices, yielding the simpler bound

$$O(n^2).$$

   Space complexity. Storing the graph and color array requires

$$O(n + m).$$

2.2 DSatur Algorithm

2.2.1 Theoretical Description

DSatur is a dynamic greedy algorithm that uses *saturation degree*. For each uncolored vertex v, define:
$$\rho(v) = |\{c(u) : u \in N(v), \ u \text{ colored}\}|.$$

At each step:

1. Choose an uncolored vertex v^* with maximum $\rho(v)$. Break ties by selecting the vertex of highest degree.

2. Assign v^* the smallest color not present in its neighborhood.

3. Update saturation degrees of its neighbors.

2.2.2 Correctness

Claim. DSatur always produces a proper coloring.

Proof. When selecting a color for v^*, the algorithm assigns the lowest index color that does not occur in its neighborhood. Thus by definition, v^* has no neighbor with the assigned color. Since this property holds at each step, the partial coloring remains proper at all times. Hence the final coloring is proper. $\square$

2.2.3 Asymptotic Analysis

Time complexity. Selecting the next vertex takes $O(n)$ time. Updating saturation degrees over all iterations contributes $O(m)$. Thus a standard implementation runs in:
$$O(n^2 + m),$$

often simplified to $O(n^2)$ for dense graphs.

Using a priority queue yields:
$$O((n + m) \log n).$$

Space complexity. Requires storing the graph and all saturation values:
$$O(n + m).$$

2.3 Simulated Annealing (SA)

2.3.1 Theoretical Description

SA is a stochastic metaheuristic inspired by thermodynamic annealing. Let S denote a (possibly improper) coloring. Define the *energy*:
$$E(S) = |\{(u, v) \in E : c(u) = c(v)\}|.$$

A move consists of recoloring a vertex v with a new color c'. A candidate move with energy change ΔE is accepted with probability:

$$P(\text{accept}) = \begin{cases} 1, & \Delta E \leq 0, \\ e^{-\Delta E/T}, & \Delta E > 0, \end{cases}$$

where T is temperature, updated by a cooling schedule (e.g. geometric $T_{k+1} = \alpha T_k$).

2.3.2 Correctness

Guarantee. SA does not guarantee finding the optimal solution in finite time. However, under a logarithmic cooling schedule and infinite time, it converges in probability to a global optimum (Geman & Geman, 1984).

    Justification (sketch). The Markov chain induced by SA is ergodic at fixed temperature. With sufficiently slow cooling, stationary distributions converge to mass concentrated on global minima. Thus SA is asymptotically correct but not in finite time.

2.3.3 Asymptotic Analysis

Time complexity. Let K be the total number of iterations. A move requires examining neighbors of the recolored vertex:

$$O(d(v)) \quad \text{per iteration.}$$

Hence total complexity:

$$O(K \cdot d_{\text{avg}}).$$

    Space complexity. Storing the current coloring and the graph:

$$O(n + m).$$

2.4 Tabu Search (TS)

2.4.1 Theoretical Description

Tabu Search is an aggressive local search metaheuristic using short-term memory to prevent cycles. Given a coloring S, define the neighborhood $N(S)$ as recolorings of a single vertex.
    At each iteration:

1. Evaluate allowed moves (v, c) not forbidden by the Tabu List.

2. Select the best move (or tabu move satisfying an aspiration criterion).

3. Apply the move and push (v, c_{old}) into the Tabu List for a fixed tenure.

2.4.2 Correctness

Guarantee. TS does not guarantee optimality or even correctness (proper coloring) but maintains a well-defined search trajectory.

    Justification. The tabu mechanism prevents immediate reversal of moves, avoiding 2-cycles and small local loops. Aspiration criteria allow overriding tabu when a globally better solution is encountered; therefore TS explores the space without stalling.

2.4.3 Asymptotic Analysis

Time complexity. If the neighborhood evaluates all recolorings of conflict vertices:

$$O(|N(S)| \cdot d_{\text{avg}})$$

per iteration. With I iterations:

$$O(I \cdot |N(S)| \cdot d_{\text{avg}}).$$

Incremental delta evaluation can reduce this to $O(I \cdot n)$ in practice.
    Space complexity. Graph + current solution + Tabu List:

$$O(n + m).$$

2.5 Genetic Algorithm (GA)

2.5.1 Theoretical Description

A GA maintains a population P of candidate solutions (chromosomes). Each chromosome is an array $C[1..n]$ where $C[i]$ is a color assignment.

For each generation:

1. Evaluate fitness of each chromosome (count conflicting edges).

2. Select parents via roulette or tournament selection.

3. Apply crossover to produce offspring.

4. Apply mutation with low probability.

5. Optionally keep elite individuals.

2.5.2 Correctness

Guarantee. GA does not guarantee reaching an optimal or even proper coloring in finite time.

Justification. With positive mutation probability and unbounded time, the GA search process is ergodic and can reach any feasible coloring with nonzero probability. However, no deterministic or finite-time guarantees hold.

2.5.3 Asymptotic Analysis

Time complexity. Fitness evaluation per individual requires scanning all edges:

$$O(m).$$

Thus for population size P and G generations:

$$O(G \cdot P \cdot m).$$

Space complexity. Storing population + graph:

$$O(Pn + m).$$

2.6 Exact Solver: Dynamic Programming Over Subsets

2.6.1 Theoretical Description

Let $f(S)$ denote the minimum number of colors needed to color the induced subgraph $G[S]$ for $S \subseteq V$. The recurrence is:

$$f(\varnothing) = 0, \quad f(S) = 1 + \min_{\substack{I \subseteq S \\ I \text{ independent}}} f(S \setminus I).$$

Dynamic programming computes $f(S)$ for all subsets $S \subseteq V$. Lawler's algorithm reduces the complexity by enumerating maximal independent sets efficiently.

2.6.2 Correctness

Claim. The DP algorithm computes $\chi(G)$ exactly.

Proof. Any optimal coloring partitions S into k independent sets. Selecting the color class used first gives an independent set I, with the remainder requiring $k - 1$ colors. Minimizing over all choices of I establishes the recurrence. DP over all subsets ensures every subproblem is solved exactly once. Thus $f(V) = \chi(G)$. □

2.6.3 Asymptotic Analysis

Time complexity. Lawler's algorithm runs in:

$$O(2.4423^n).$$

This follows from improvements in independent-set enumeration and pruning over the naive $O(3^n)$ DP.

Space complexity. DP table of size 2^n:

$$O(2^n).$$

2.7 Summary of Guarantees

- Welsh–Powell: always proper; not optimal.

- DSatur: always proper; not optimal.

- SA: asymptotically convergent; no finite-time guarantee.

- Tabu: heuristic; no optimality guarantee.

- GA: probabilistic; no finite-time guarantee.

- Exact DP: correct and optimal; exponential time/space.

3 Implementation Details

This section discusses the key design choices, data structures, and implementation challenges encountered during the development of the graph colouring algorithms.

3.1 General Architecture & Graph Representation

The core of the system relies on a unified graph representation designed to support both exact and heuristic solvers efficiently.

- **Data Structure:** The graph is implemented as an Adjacency List (`std::vector<std::vector<int>>`) rather than an Adjacency Matrix.

  - *Reasoning:* Most benchmark graphs (like DIMACS) are sparse. An adjacency matrix would consume $\mathcal{O}(V^2)$ memory, causing allocation failures on large graphs. The adjacency list reduces memory usage to $\mathcal{O}(V+E)$ and allows for faster iteration over neighbours.

- **Parsing:** The `load_graph` function handles the DIMACS format, automatically converting 1-based indices to 0-based internal indices to align with C++ vector indexing.

- **Output:** A centralized `ResultsLogger` outputs CSV data, ensuring consistent formatting for runtime analysis and comparisons.

3.2 Welsh-Powell Algorithm

- **Design Choice:** A static greedy approach. The algorithm pre-processes vertices once rather than dynamically re-evaluating them during colouring.

- **Data Structures:**

  - `std::vector<int> order`: Stores vertex indices.
  - `std::vector<int> colour`: Stores the final assignment.

- **Key Implementation Detail:** The vertices are sorted by **degree in descending order** using a custom lambda comparator. This heuristic assumes that high-degree nodes are the hardest to colour and should be handled first.

- **Challenge:** The main bottleneck is the nested loop structure—for every colour class, we iterate through the remaining uncoloured vertices to check for conflicts. While simple, this is cache-inefficient for very large graphs.

3.3 DSATUR (Degree of Saturation)

- **Design Choice:** Unlike Welsh-Powell, DSATUR is dynamic. It selects the next vertex based on **saturation degree** (number of differently coloured neighbours).

- **Data Structures:**

  - `std::set<NodeInfo, MaxSatCmp>`: This is the most critical structure. It acts as an updateable priority queue. Standard `std::priority_queue` does not support updating keys (saturation) of arbitrary elements efficiently. A `std::set` was used to allow finding, erasing, and re-inserting neighbour nodes when their saturation changed.
  - `std::vector<std::unordered_set<int>> nb_colours`: Used to track exactly which unique colours are adjacent to each node to calculate saturation in $\mathcal{O}(1)$ (amortized) time.

- **Challenge:** The most significant challenge was performance. Every time a node is coloured, *all* its uncoloured neighbours must be removed from the `set`, their saturation updated, and then re-inserted. This overhead makes DSATUR slower than Welsh-Powell per iteration but yields significantly better chromatic numbers.

3.4 Simulated Annealing (SA)

- **Design Choice:** The implementation uses a **fixed-k approach wrapped in a decrement loop**. Instead of optimizing k directly, the algorithm tries to find a valid colouring for a specific k. If successful (0 conflicts), it decrements k and retries.

- **Data Structures:**

  - `std::vector<int> colours`: Represents the current state.

- **Key Mechanisms:**

  - **Cost Function:** Defined strictly as the number of edge conflicts.
  - **Greedy Repair:** The initial state for a new k isn't random; it uses a "Greedy Repair" function to generate a partially valid solution, respecting the new palette size.

- **Cooling Schedule:** Uses geometric cooling ($T \times \alpha$) where α is calculated based on the number of iterations to ensure T reaches T_{min} exactly at the end.

- **Challenge:** Tuning the cooling schedule and deciding when to "give up" on a specific k. If the temperature drops too fast, the algorithm gets stuck in local minima (conflicts > 0).

3.5   Tabu Search

- **Design Choice:** A local search metaheuristic that allows moving to worse solutions to escape local minima, using memory (Tabu list) to prevent cycling. Like SA, it uses the "decrementing k" strategy.

- **Data Structures:**

  - `std::vector<std::vector<int>> tabu`: A 2D matrix where `tabu[v][c]` stores the **iteration number** until which assigning colour `c` to vertex `v` is forbidden. This is $\mathcal{O}(1)$ to check compared to storing a list of moves.

- **Key Mechanisms:**

  - **Incremental Evaluation:** The code calculates `delta` (change in conflicts) for a move. It does *not* recount the total graph conflicts every step, which would be too slow.
  - **Aspiration Criterion:** A tabu move is allowed if it results in a configuration better than the global best found so far.

- **Challenge:** Implementing the "1-move neighborhood" efficiently. For every conflicting vertex, the algorithm must check all possible alternative colours to find the best move, which is computationally expensive for large k.

3.6   Genetic Algorithm

- **Design Choice:** Uses a hybrid "Memetic" approach. It combines evolutionary operators with local search (Greedy Repair) to ensure the population remains viable.

- **Data Structures:**

  - `struct Individual`: Encapsulates the gene (colour vector), fitness score, and conflict count.

- **Key Mechanisms:**

  - **GPX-Lite Crossover:** Instead of random crossover (which destroys graph structures), the implementation uses a lightweight Greedy Partition Crossover (GPX) approach. It tries to inherit valid colour blocks from parents.
  - **Conflict-Focused Mutation:** Mutation doesn't just change random bits; it targets vertices involved in conflicts and attempts to move them to a better colour class.

- **Challenge:** The "Feasibility Problem." In graph colouring, 99% of random mutations result in invalid colourings. The challenge was implementing the `greedy_repair_fixed_k` function to fix broken children immediately after crossover, ensuring the algorithm searches the space of *near-valid* solutions rather than random noise.

3.7   Exact Solver

- **Design Choice:** A **Backtracking** algorithm with Branch and Bound pruning.

- **Data Structures:**

  - **Recursion Stack:** Implicitly maintains the state.
  - `best_solution`: Stores the best complete colouring found so far to update the upper bound.

- **Key Optimization:**

  - **Initial Bound:** It runs `colour_with_dsatur` *before* starting the recursion to establish a tight Upper Bound (UB). If the backtracking current colours exceed this UB, the branch is pruned immediately.
  - **Variable Ordering:** It uses a heuristic `select_vertex` function inside the recursion to pick the most constrained vertex next (similar to DSATUR logic), failing invalid branches as early as possible.

- **Challenge:** Handling the NP-Hard nature of the problem. For large graphs ($V > 100$), the search space is too vast. The implementation includes a timeout/reporting mechanism (`ProgressState`) to monitor progress, as the algorithm may not converge in reasonable time for dense graphs.