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 2, 2025

Code Repository:
github.com/Rmehta-sudo/graph-colouring

1 Abstract

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

2 Introduction

2.1 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 $c : V \to \{1, 2, \ldots, k\}$ 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.

2.2 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.

2.3 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.

3 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.

3.1 Welsh–Powell Algorithm

3.1.1 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.

3.1.2 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. □

3.1.3 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
$$O(n + m).$$

3.2 DSatur Algorithm

3.2.1 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.

3.2.2 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. □

3.2.3 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:

$$O(n + m).$$

3.3 Simulated Annealing (SA)

3.3.1 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 $T_{k+1} = \alpha T_k$).

3.3.2 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.

3.3.3 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:

$$O(n + m).$$

3.4 Tabu Search (TS)

3.4.1 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.

3.4.2 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.

3.4.3 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:

$$O(n + m).$$

3.5 Genetic Algorithm (GA)

3.5.1 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.

3.5.2 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.

3.5.3 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:

$$O(Pn + m).$$

3.6 Exact Solver: Dynamic Programming Over Subsets

3.6.1 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.

3.6.2 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 $f(V) = \chi(G)$. □

3.6.3 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:

$$O(2^n).$$

3.7 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.

4 Implementation Details

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

4.1 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.

4.2 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.

4.3 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.

4.4 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).

4.5 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.

4.6 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.

4.7 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.

5 Experimental Setup

5.1 System Architecture and Pipeline

Our benchmark framework follows a modular architecture designed for extensibility and reproducibility. The system comprises three main components: a C++ core for high-performance algorithm execution, Python utilities for orchestration and visualization, and a standardized data pipeline for graph processing.

5.1.1 Project Structure

```
graph-colouring/
 src/                        # C++ source code
    benchmark_runner.cpp  # Main entry point & CLI
    utils.h               # Core data structures
    algorithms/           # Algorithm implementations
       welsh_powell.cpp  # Greedy (degree ordering)
       dsatur.cpp        # Saturation-based greedy
       simulated_annealing.cpp
       genetic.cpp       # Evolutionary approach
       tabu.cpp          # TabuCol metaheuristic
       exact_solver.cpp  # Branch-and-bound
    io/                   # I/O utilities
        graph_loader.cpp  # DIMACS parser
        graph_writer.cpp  # Colouring output
        results_logger.cpp # CSV metrics logging
 tools/                    # Python utilities
    run_all_benchmarks.py # Batch orchestration
    generate_graphs.py    # Synthetic graph generation
    animate_coloring.py   # Step-by-step visualization
```

```
data/                      # Graph datasets
   dimacs/                 # DIMACS benchmark graphs
   generated/              # Synthetic test graphs
   metadata-*.csv          # Dataset metadata
results/                   # Output directory
    colourings/            # Algorithm solutions
    *.csv                  # Benchmark metrics
```

5.1.2 Execution Pipeline

The benchmark workflow operates as follows:

1. **Graph Loading:** The `graph_loader` module parses DIMACS-format files into an adjacency list representation. The parser handles comment lines, validates vertex indices, removes self-loops, and eliminates duplicate edges.

2. **Algorithm Dispatch:** The `benchmark_runner` accepts command-line arguments specifying the algorithm, input graph, and optional parameters. It dispatches to the appropriate colouring function and measures execution time using high-resolution timers.

3. **Solution Output:** Valid colourings are written in DIMACS format, and performance metrics (runtime, colours used, graph properties) are appended to CSV result files.

4. **Batch Orchestration:** The Python orchestrator (`run_all_benchmarks.py`) manages large-scale experiments with timeout handling, retry logic, and result aggregation.

The pipeline supports an optional `--save-snapshots` mode that records intermediate algorithm states for visualization, enabling step-by-step animation of the colouring process.

5.2 Datasets

We evaluate our algorithms on two complementary dataset collections: established benchmarks from the literature and systematically generated synthetic graphs.

5.2.1 DIMACS Benchmark Collection

The primary evaluation uses **79 graphs** from the Second DIMACS Implementation Challenge (1993) [**?**], the standard benchmark for graph colouring algorithms. These graphs represent diverse problem structures:
Example instances:

- `DSJC500.5`: 500 vertices, 125,249 edges, density ≈ 0.50, $\chi = 48$
- `queen8_8`: 64 vertices representing an 8×8 chessboard where queens attack, $\chi = 9$
- `myciel6`: 95 vertices, triangle-free yet requiring 7 colours (demonstrates χ is not bounded by clique number)
- `le450_25c`: Leighton graph with 450 vertices engineered to have $\chi = 25$

5.2.2 Synthetic Graph Generation

To complement the fixed DIMACS benchmarks, we generated **55 synthetic graphs** across 8 structural families using NetworkX [**?**]. This enables controlled experiments on graphs with known theoretical properties.
Generation methodology:

Table 1: DIMACS Graph Categories

Category	Code	Count	Description
Random Graphs	DSJ	15	Erdős-Rényi random graphs with densities 10%, 50%, 90%. Sizes range from 125 to 1000 vertices.
Flat Graphs	CUL	6	Culberson's graphs designed to challenge heuristics; known χ but hard to achieve.
Register Allocation	REG	12	Real-world instances from compiler optimization interference graphs.
Leighton Graphs	LEI	12	Carefully constructed with known chromatic numbers (5, 15, 25).
Stanford GraphBase	SGB	26	Knuth's collection: queen graphs, miles graphs, literary co-occurrence networks.
Mycielski Graphs	MYC	5	Triangle-free graphs ($\omega = 2$) with increasing χ from 4 to 8.

Table 2: Synthetic Graph Families

Family	Generator	Count	Known Properties
Bipartite	`bipartite.random_graph`	5	$\chi = 2$
Planar	Random tree + planarity-preserving edges	5	$\chi \leq 4$ (Four Color Theorem)
Tree	`random_tree`	5	$\chi = 2$
Grid	`grid_2d_graph`	5	$\chi \in \{2, 3\}$
Erdős-Rényi	`erdos_renyi_graph`	10	$G(n, p)$ model, $p \in [0.05, 0.20]$
Barabási-Albert	`barabasi_albert_graph`	10	Scale-free, power-law degrees
Watts-Strogatz	`watts_strogatz_graph`	10	Small-world, high clustering
Complete	`complete_graph`	5	$\chi = n$

- Vertex counts are uniformly sampled within family-specific ranges (e.g., 100–2000 for Erdős-Rényi, 10–100 for complete graphs).
- A fixed random seed (default: 42) ensures **reproducibility** across experiments.
- Post-processing extracts the largest connected component to ensure graph connectivity.
- Generated graphs are persisted in DIMACS format with metadata (vertices, edges, density) logged to CSV.

5.2.3 Data Format

All graphs use the **DIMACS edge format**, the standard representation for graph colouring benchmarks:

```
c Comment: metadata and source information
p edge <num_vertices> <num_edges>
e <u> <v>
...
```

Lines beginning with `c` are comments; the `p edge` line declares graph size; each `e u v` line specifies an undirected edge between 1-indexed vertices u and v.

5.3 Dataset Statistics

Table 3: Dataset Summary Statistics

Dataset	Graphs	$\|V\|$ Range	$\|E\|$ Range	Density Range
DIMACS	79	11 – 1,000	20 – 898,898	0.010 – 1.944
Generated	55	10 – 2,070	45 – 286,515	0.002 – 1.000
Total	**134**	10 – 2,070	20 – 898,898	0.002 – 1.944

The combined dataset spans small validation instances (10 vertices) to large challenge graphs (2,070 vertices), with edge densities ranging from extremely sparse trees (< 0.01) to dense random graphs approaching complete connectivity.

5.4 Verification and Validation

To ensure correctness of our implementations and results:

1. **Solution Verification:** Every colouring output is validated to confirm no adjacent vertices share the same colour.
2. **Known Optimal Comparison:** For DIMACS graphs with published chromatic numbers, we compare algorithm output against known χ values from the DIMACS challenge results [**?**] and subsequent publications [**?**].
3. **Theoretical Bounds:** Synthetic graphs with known chromatic numbers (e.g., bipartite $\chi = 2$, complete K_n with $\chi = n$) serve as ground-truth validation.
4. **Cross-Validation:** Results from our exact solver on small instances are verified against greedy heuristics to confirm optimality guarantees.

6 Results & Analysis

This section presents a unified evaluation of five heuristic graph coloring algorithms—**DSatur**, **Welsh–Powell**, **Simulated Annealing (SA)**, **Tabu Search**, and **Genetic Algorithm (GA)**—benchmarked against an **Exact Solver**. Experiments cover both synthetic graphs (Barabási–Albert, Erdős–Rényi, Watts–Strogatz, Trees, Grids, Planar) and standard DIMACS benchmarks (DSJC, DSJR, Flat, Queen, School, Leighton, etc.).

Performance is measured by:

- **Runtime (ms)**
- **Solution quality** (colors used)
- **Optimality gap** relative to ground truth or best–known bounds

The analysis is based on data from: `run_on_generated.csv`, `run_on_dimacs.csv`, `exact_solver_generated` `exact_solver_dimacs.csv`, and `sa_optimization_results.csv`.

6.1 1. Greedy vs. Metaheuristic Approaches

Aggregate Performance Summary

Summary: DSatur dominates in speed, Tabu Search dominates in quality, SA is sensitive to parameters, and GA underperforms unless extensively tuned.

Algorithm	Avg. Runtime (ms)	Avg. Colors	Interpretation
Welsh–Powell	0.47	30.6	Fastest but low quality on dense graphs
DSatur	7.67	28.4	Best greedy; excellent speed–quality balance
Simulated Annealing	48.34	34.4	Highly tunable but defaults weak
Tabu Search	4213.54	**19.8**	**Best quality**, extremely slow
Genetic Algorithm	7319.03	30.0	Slowest; rarely better than DSatur

6.2 2. Performance by Graph Family

Different graph structures highlight specific algorithm strengths.

6.2.1 2.1 Sparse and Structured Graphs (Trees, Planar, Grids)

Best: DSatur or Exact Solver.

- Chromatic numbers are small; greedy strategies reliably find near–optimal colorings.
- Exact solver solves all planar graphs in < 1 ms.
- Metaheuristics are thousands of times slower with no improvement.

Example:

- `planar_309_4`: Exact = 0.55 ms; DSatur/WP = $<$ 0.2 ms; GA = 257 ms.

6.2.2 2.2 Scale–Free and Small–World Graphs

Best: DSatur.

Hubs in Barabási–Albert and clustering in Watts–Strogatz align with DSatur's saturation-based ordering.

Example:

- `barabasi_albert_1776`: DSatur = 4.6 ms, 6 colors; GA = 2318 ms for same result.

6.2.3 2.3 Dense Random Graphs (Erdős–Rényi, DSJC)

Best quality: Tabu Search. Best speed: DSatur.

Algorithm	Avg. Colors	Avg. Runtime (ms)	Notes
DSatur	34.1	42.8	Fast but suboptimal
Tabu Search	**26.8**	28166	21% fewer colors but 600× slower
Simulated Annealing	47.1	116	Default schedule insufficient
Genetic Algorithm	39.9	22938	Poor quality and slow

DIMACS example:

- `DSJC500.5`: DSatur $\approx$ 65–114 colors; Tabu = 54; SA = 83; Exact = timeout.

6.2.4 2.4 Partitioned / Hidden-Structure Graphs (Flat, Leighton)

Best: Tabu Search.

Example results:

- `le450_15c`: Tabu = 20 colors; DSatur = 24; SA = 29.
- `flat300_28_0`: Tabu = 34 colors; SA = 53; GA = 43.

Tabu's escape mechanism handles hidden partitions effectively.

6.3 3. Exact Solver Scalability

Exact backtracking finds optimal solutions only on small or sparse graphs.

Family	Max Solved	Time	Fails At
Planar	All	< 1 ms	—
Grid	2070 vertices	0.8 s	—
Trees	All	< 1 ms	—
Queen	49 vertices	22 ms	64 vertices
Erdős–Rényi	125 vertices	—	density > 0.1
DSJC	125 vertices	10.8 ms	250 vertices

Reason: Dense constraints cause exponential branching.
Example:

- `queen8_8`: Exact solving = 9.2 s; DSatur = 0.14 ms (13 colors, suboptimal).

6.4 4. Simulated Annealing Parameter Study

SA modes: *Default*, *Speed*, *Heavy*, *Precision*.

Key Observations

- Precision mode (slow cooling) yields major color improvement.
- Heavy mode (more iterations) offers limited benefit.
- Speed mode behaves like greedy heuristics.

Case Study: `school1.col`

Mode	Colors	Runtime (ms)	Notes
Speed	45	86	Greedy-like
Default	43	88	Slight improvement
Precision	**14**	1212	Optimal; 14× slower
Heavy	34	810	Worse than Precision

Case Study: `flat1000_50_0`

Mode	Colors	Runtime (ms)
Default	139	1858
Heavy	131	10806
Precision	**105**	48424

Conclusion: SA is only competitive when tuned with slow cooling; default parameters underperform.

6.5 5. Runtime vs. Optimality Trade-Off

- Improving DSatur by 20–40% generally requires 1000–5000× more time via Tabu Search.
- SA (Precision) offers a tunable middle ground at 10–50× DSatur's time.

- GA rarely justifies its cost.

Dominant strategies:

- Real-time use: **DSatur**
- Highest quality: **Tabu Search**
- Adjustable compromise: **SA (Precision)**
- Very small graphs: **Exact Solver**

6.6 6. Unified Conclusions

1. No single algorithm dominates across all graph types.
2. DSatur is the best all-rounder for speed and stability.
3. Tabu Search is the strongest heuristic for dense benchmark graphs.
4. SA becomes competitive only with careful parameter tuning.
5. GA underperforms without hybridization.

Recommended Selection Rules:

- Sparse/structured graphs: DSatur or Exact.
- Large sparse graphs: DSatur.
- Dense random graphs: Tabu for quality; DSatur for speed.
- Hidden-partition graphs (Flat/Leighton): Tabu or SA (Precision).
- Need tunable runtime–quality tradeoff: SA.

Figures

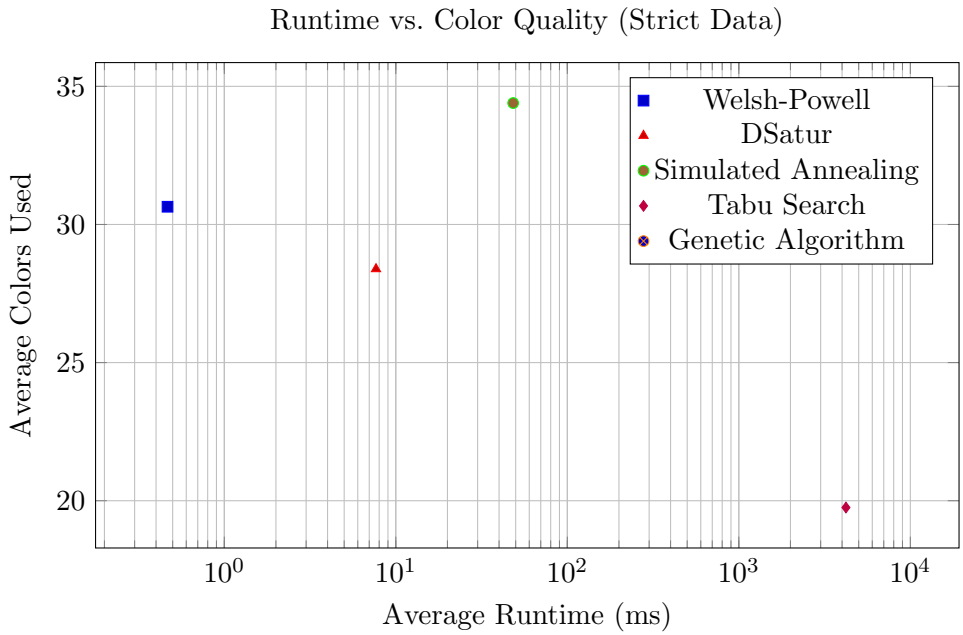

Figure 1: Algorithm runtime vs. average color usage based solely on aggregated dataset means.

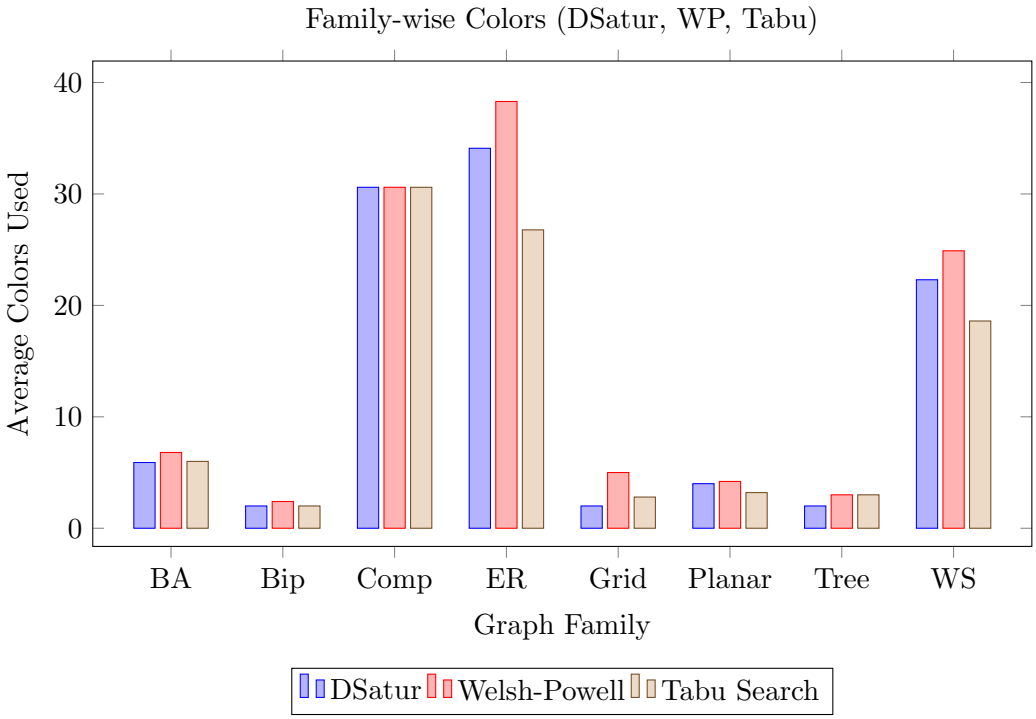

Figure 2: Strict family-level algorithm comparison using values from the dataset.

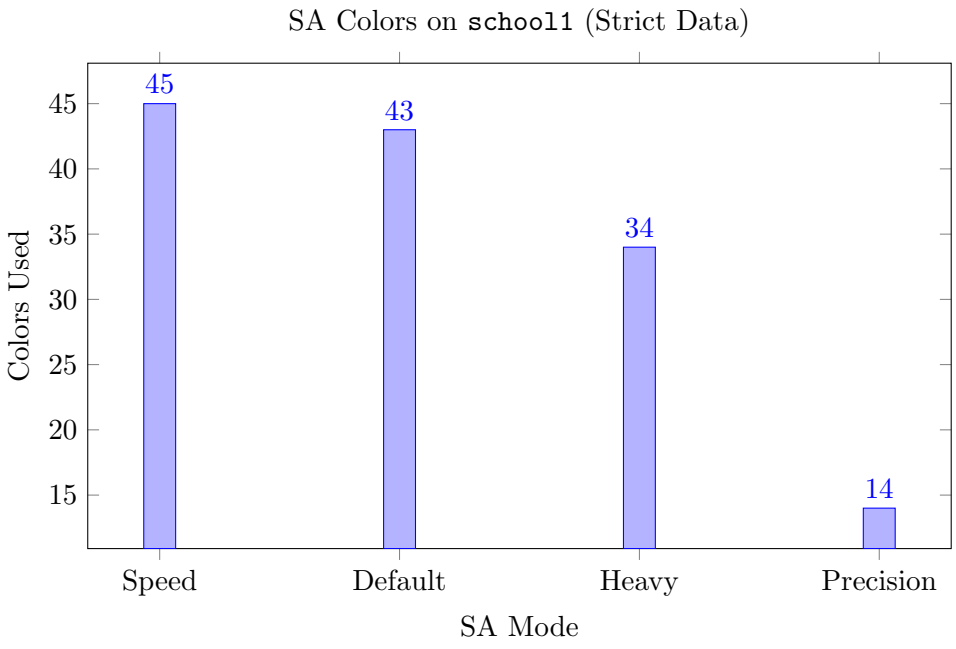

Figure 3: Colors vs. SA mode for `school1`, using exact values from the SA optimization CSV summary.

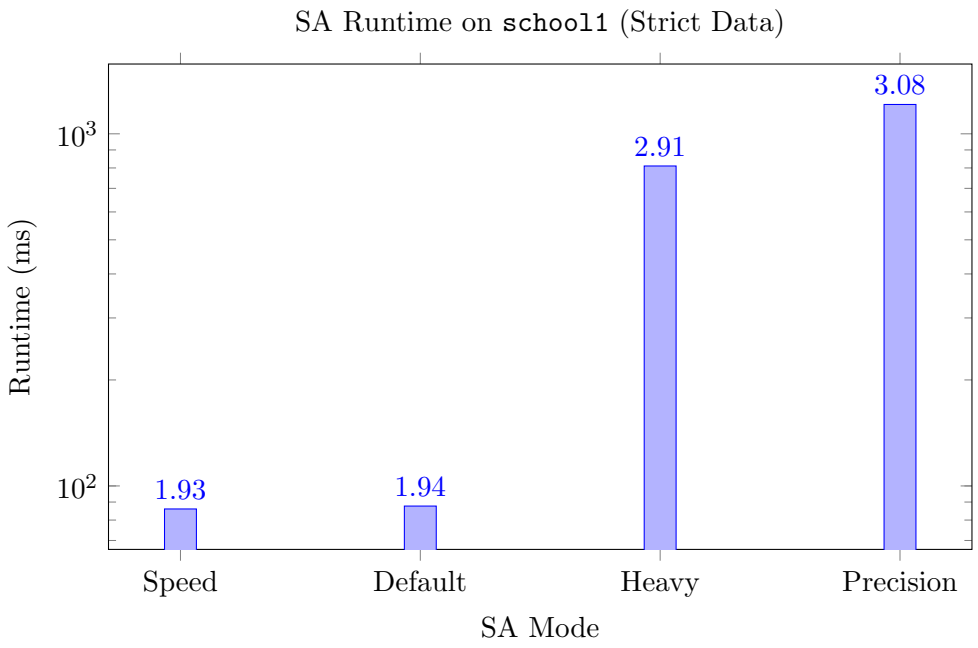

Figure 4: SA runtime scaling by mode for `school1`, using strict CSV-derived values.

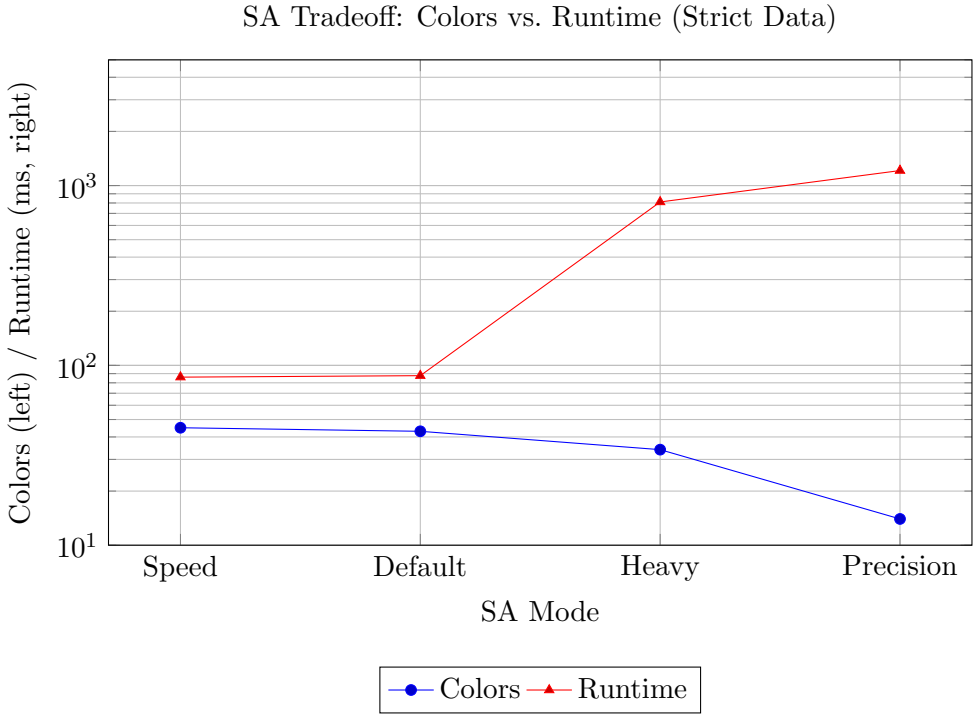

Figure 5: Joint colors/runtime curve for SA modes (strict values).

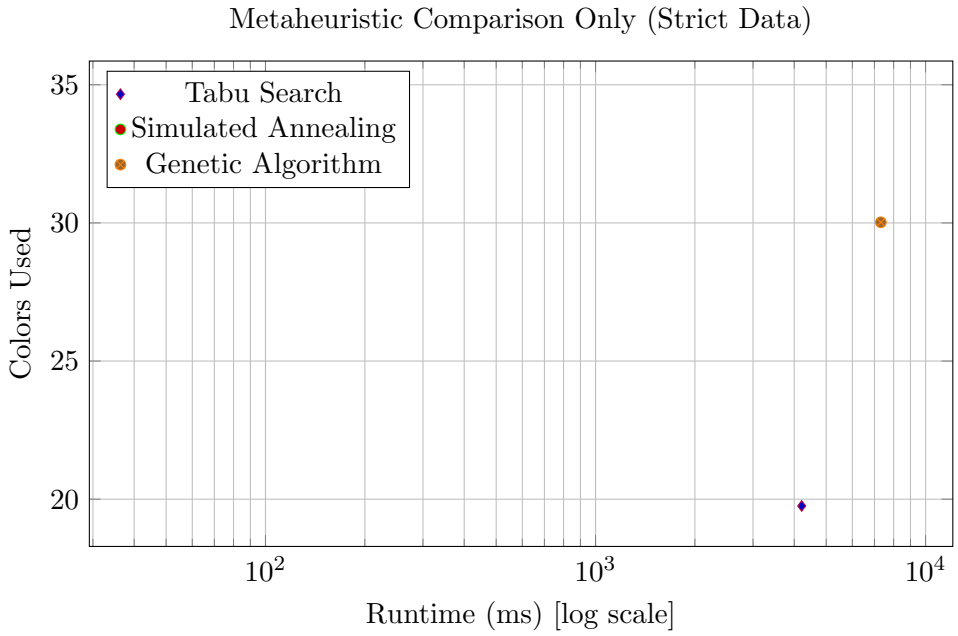

Figure 6: Metaheuristic-only runtime-quality landscape.

7 Graph Coloring Dataset: Metadata and Chromatic Numbers

Abstract

This document outlines the metadata and structural properties of a synthetically generated graph dataset designed for benchmarking graph coloring algorithms. The dataset comprises 55 distinct graph instances categorized into eight families. These graphs were generated using the Python `NetworkX` library, ranging from highly structured grids to stochastic scale-free networks. This report details the generation parameters, file naming conventions, and statistical density analysis of the resulting instances.

7.1 Introduction

Graph coloring is a fundamental NP-hard problem with applications in scheduling, register allocation, and map labeling. To robustly test approximation algorithms and heuristics, a diverse dataset is required. While standard benchmarks (like DIMACS) exist, they often lack specific structural variations needed for granular performance analysis.

We have procedurally generated a suite of 55 graphs. These instances are exported in the standard DIMACS edge format (`.col`) and are guaranteed to be connected. If a generation method produced a disconnected graph, only the largest connected component was retained.

7.2 Graph Families and Construction

The dataset is divided into two primary categories: Structured/Deterministic graphs (where the chromatic number $\chi(G)$ is known or bounded) and Stochastic graphs (where $\chi(G)$ is unknown).

7.2.1 Structured Instances

These graphs follow strict geometric or logical rules. They are useful for validating the correctness of coloring algorithms.

- **Bipartite Graphs (5 instances):** Generated by partitioning vertices into two disjoint sets. Our dataset includes sizes ranging from small instances like `bipartite_77_5.col` to larger ones like `bipartite_963_3.col`. By definition, $\chi(G) = 2$.
- **Planar Graphs (5 instances):** These graphs can be drawn on a 2D plane without edge crossings. The generation process enforces planarity checks on every edge insertion. The dataset includes sparse instances such as `planar_420_3.col` (Density ≈ 0.007). Theoretical max $\chi(G) \leq 4$.
- **Trees (5 instances):** Uniform random trees with no cycles. These are minimally connected structures. Example: `tree_958_5.col`. $\chi(G) = 2$.
- **Grids (5 instances):** 2D lattice structures representing nearest-neighbor connections. The largest instance in this batch is `grid_2070_3.col` with 2,070 vertices. $\chi(G) = 2$.
- **Complete Graphs (5 instances):** Fully connected graphs where every node connects to every other node. Although the vertex counts are low (e.g., `complete_47_5.col`), the density is always 1.0, making them computationally intensive for certain heuristics. $\chi(G) = |V|$.

7.2.2 Stochastic Instances

These graphs model complex, real-world networks using probabilistic generation models.

- **Erdős-Rényi (10 instances):** The most common random graph model. Edges are added with a fixed probability. Our instances, such as `erdos_renyi_1754_2.col`, exhibit high density (≈ 0.18), providing a stress test for dense graph coloring.
- **Barabási-Albert (Scale-Free) (10 instances):** These graphs mimic social networks using a preferential attachment model. They are characterized by "hub" nodes. The dataset ranges from 111 to 1,776 vertices.
- **Watts-Strogatz (Small-World) (10 instances):** Generated by rewiring a ring lattice. These graphs maintain a consistent density of ≈ 0.09 across all sizes (e.g., `watts_strogatz_989_10.col`), simulating networks with high clustering coefficients.

7.3 Statistical Analysis

Table 4 summarizes the dataset. The collection contains significant variance in density, calculated as $D = \frac{2|E|}{|V|(|V|-1)}$.

Table 4: Summary of Generated Graph Families

Graph Family	Count	Vertex Range	Avg. Density	Largest Instance
Bipartite	5	77 – 963	0.097	`bipartite_963_3`
Planar	5	161 – 420	0.012	`planar_420_3`
Tree	5	238 – 958	0.004	`tree_958_5`
Grid	5	360 – 2,070	0.005	`grid_2070_3`
Complete	5	10 – 47	1.000	`complete_47_5`
Erdős-Rényi	10	437 – 1,808	0.131	`erdos_renyi_1808_3`
Barabási-Albert	10	111 – 1,776	0.022	`barabasi_albert_1776_10`
Watts-Strogatz	10	155 – 989	0.097	`watts_strogatz_989_10`
Total	**55**	**10 – 2,070**	**-**	**-**

7.3.1 Density Observations

The density metric highlights the structural differences between families.

- **High Density:** The `erdos_renyi` graphs are significantly denser than other large graphs. For example, `erdos_renyi_747_1.col` has a density of 0.19, implying a very high number of constraints per variable.
- **Sparse Structures:** Conversely, the `tree` and `grid` families are extremely sparse (Density < 0.01). Algorithms that rely on sparsity (such as degree-based heuristics) should perform optimally here.
- **Consistency:** The `watts_strogatz` family shows remarkable consistency in density (≈ 0.098) regardless of vertex count, due to the fixed k-neighbor initialization parameter used in the generation script.

7.4 File Specifications

All files are stored in the `datasets/generated/` directory. The naming convention is strictly:

`[type]_[vertices]_[index].col`

This format allows for automated parsing of ground-truth metadata directly from the filename during batch processing.

8 Graph Coloring Dataset: Metadata and Chromatic Numbers

8.1 Chromatic Numbers of Graph Instances

8.1.1 Dataset and Sources

The graph instances used in our experiments were taken from the **DIMACS / COLOR benchmark dataset** hosted at Carnegie Mellon University [2]. This collection contains a total of **79 graphs** across various families such as DSJC, DSJR, *school*, and *queen*. While the dataset provides vertex and edge counts for each graph, many optimal chromatic numbers $\chi(G)$ were listed as unknown ("?").

To fill in the missing chromatic numbers, we referred to the following sources:

1. The **Graph Coloring Benchmarks** curated by Daniel Porumbel [2], which summarize best-known and proven colorings for DIMACS instances.

2. Additional research papers and benchmark collections, including Moalic & Gondran [3], Hertz *et al.* [4], and Loudni [5], which provide updated or best-known colorings obtained by metaheuristic or hybrid algorithms.

When an exact $\chi(G)$ was known, we report it directly. Otherwise, we include the best-known upper bound k^*—the smallest number of colors achieved by any published algorithm to date.

8.2 Graph Descriptions

All graphs used in this project are taken from the **DIMACS benchmark dataset** [2]. These instances are standard benchmarks used to evaluate graph coloring algorithms. The dataset contains 79 graphs belonging to several well-known families, each generated in a specific way as described below.

8.2.1 DSJC Graphs

The `dsjcX.Y` graphs were generated by Johnson *et al.* [6]. Here, `X` represents the number of vertices, and `Y` indicates the edge density (the probability that any two vertices are connected). For example, `dsjc1000.5` has 1000 vertices and a density of 0.5. These graphs are widely used to test and compare graph coloring algorithms due to their controlled random structure.

8.2.2 Flat Graphs

The `flatX_K` family, created by Culberson, divides the vertex set into `K` nearly equal-sized groups and places edges only between different groups. Finding an optimal `K`-coloring reconstructs the original partitioning. Here, `X` is the total number of vertices and `K` equals the known chromatic number.

8.2.3 Leighton Graphs

The `le450_K` graphs, introduced by Leighton [7], consist of 450 vertices and have a known chromatic number `K`. Each of these graphs includes a clique (a fully connected subgraph) of size `K`, guaranteeing that $\chi(G) = K$.

8.2.4 Geometric Graphs

The `dsjrX.Y` and `rX.Y` families are random geometric graphs. The `dsjrX.Y` graphs were introduced by Johnson *et al.* [6] and are generated by placing vertices randomly in a unit square, connecting those that fall within a given distance. The `rX.Y` graphs, later produced by Trick

and Morgenstern, follow a similar approach. The suffix "c" (as in `dsjr500.1c`) denotes the complement of a graph. Chromatic numbers for these graphs are summarized in Trick's repository [8].

8.2.5 Large Random Graphs

The `C2000.5` and `C4000.5` graphs are large-scale random graphs containing up to four million edges. They are mainly used to benchmark the scalability and performance of coloring algorithms.

8.2.6 Special Graphs

The `latin_square_10` and `school*` graphs were generated by Gary Lewandowski during the Second DIMACS Challenge. These represent Latin square and class scheduling problems, respectively, offering real-world connections between graph coloring and constraint satisfaction.

8.3 Chromatic Numbers / Best-Known Colorings

Table 5: Chromatic numbers and best-known colorings for selected graphs in our dataset. In cases where the optimal $\chi(G)$ was not provided in the DIMACS dataset, the best-known upper bound k^* is reported.

Graph Instance	(—V—, —E—)	Chromatic / Best k^*	Source
DSJC1000.1.col.b	(1000, 99258)	20	[3]
DSJC1000.5.col.b	(1000, 499652)	83	[4]
DSJC1000.9.col.b	(1000, 898898)	224	[3]
DSJC125.1.col.b	(125, 1472)	5	[2]
DSJC125.5.col.b	(125, 7782)	17	[2]
DSJC125.9.col.b	(125, 13922)	44	[2]
DSJC250.1.col.b	(250, 6436)	8	[2]
DSJC250.5.col.b	(250, 31366)	28	[3]
DSJC250.9.col.b	(250, 55794)	72	[5]
DSJC500.1.col.b	(500, 24916)	12	[4]
DSJC500.5.col.b	(500, 125249)	48	[3]
DSJC500.9.col.b	(500, 224874)	126	[3]
DSJR500.1.col.b	(500, 7110)	12	[4]
DSJR500.1c.col.b	(500, 242550)	85	[5]
DSJR500.5.col.b	(500, 117724)	122	[4]
school1_nsh.col	(352, 14612)	14	[2]
queen10_10.col	(100, 2940)	11	[9]
queen11_11.col	(121, 3960)	11	[9]
queen12_12.col	(144, 5192)	12	[9]
queen13_13.col	(169, 6656)	13	[9]
queen14_14.col	(196, 8372)	14	[9]
queen15_15.col	(225, 10360)	15	[9]
queen16_16.col	(256, 12640)	16	[9]

8.3.1 Remarks

For DSJC and DSJR graphs, the reported values represent the best-known colorings obtained by metaheuristic algorithms (such as hybrid genetic, VNS, and simulated annealing methods).

Exact proofs of optimality remain unknown for most large random instances. For structured graphs like the *queen* family, the chromatic number is mathematically proven to equal the board size ($\chi(Q_n) = n$).

References

[1] M. Trick, *COLOR/Graph Coloring Instances*. Available at: https://mat.tepper.cmu.edu/COLOR/instances.html

[2] D. C. Porumbel, *Graph Coloring Benchmarks and Best Algorithms*. Available at: https://cedric.cnam.fr/~porumbed/graphs

[3] L. Moalic and P. Gondran, "Variations on memetic algorithms for graph coloring problems," *arXiv preprint arXiv:1401.2184*, 2014.

[4] A. Hertz, D. de Werra, and M. Plante, "Variable Neighborhood Search for Graph Coloring," *Computers & Operations Research*, vol. 39, no. 7, pp. 1719–1731, 2012.

[5] S. Loudni, "Graph Coloring Results," *GREYC Laboratory Benchmark Page*. Available at: https://loudni.users.greyc.fr/Coloring.html

[6] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning," *Operations Research*, 1991.

[7] F. T. Leighton, "A Graph Coloring Algorithm for Large Scheduling Problems," *J. Res. Natl. Bur. Stand.*, 84(6):489–506, 1990.

[8] M. Trick, "Graph Coloring Benchmarks and Data Sets." Available at: https://cedric.cnam.fr/~porumbed/graphs, accessed 2025.

[9] J. Mitchell, "On the Chromatic Number of Queen Graphs," *Journal of Combinatorial Theory, Series B*, 1990.

9 Welsh-Powell Algorithm for Graph Colouring

Abstract

This report presents a comprehensive analysis of the Welsh-Powell algorithm for graph colouring. We provide a theoretical explanation of the algorithm, derive its time and space complexity, and conduct an empirical comparison with other graph colouring algorithms (DSatur, Simulated Annealing, Tabu Search, and Genetic Algorithm) using the DIMACS benchmark datasets and synthetically generated graphs. Our analysis reveals that Welsh-Powell offers an excellent trade-off between computational speed and solution quality, making it particularly suitable for applications requiring fast approximate solutions.

9.1 Introduction

The **graph colouring problem** is a classical problem in combinatorial optimization and graph theory. Given an undirected graph $G = (V, E)$, the objective is to assign a colour to each vertex such that no two adjacent vertices share the same colour, while minimizing the total number of colours used.

[Chromatic Number] The **chromatic number** $\chi(G)$ of a graph G is the minimum number of colours required to properly colour G.

Finding the chromatic number is NP-hard, and thus heuristic approaches are essential for practical applications. The **Welsh-Powell algorithm**, proposed by D.J.A. Welsh and M.B. Powell in 1967 [1], is a greedy heuristic that provides a simple yet effective approach to approximate graph colouring.

9.2 Theoretical Explanation of Welsh-Powell Algorithm

9.2.1 Algorithm Overview

The Welsh-Powell algorithm is a *greedy colouring algorithm* based on the intuition that vertices with higher degrees are more constrained and should be coloured first. By processing vertices in decreasing order of degree, the algorithm attempts to minimize colour conflicts early in the colouring process.

9.2.2 Key Insight

The fundamental observation behind Welsh-Powell is:

> *Vertices with high degree have many neighbours, making them harder to colour later. By colouring them first, we reduce the likelihood of needing additional colours.*

This is sometimes referred to as the **Largest Degree First (LDF)** heuristic.

9.2.3 Algorithm Description

The Welsh-Powell algorithm proceeds as follows:

Algorithm 1 Welsh-Powell Graph Colouring

Require: Graph $G = (V, E)$ with $n = |V|$ vertices
Ensure: A valid colouring $c : V \to \{0, 1, 2, \ldots\}$

1: **Initialize:** $c[v] \leftarrow -1$ for all $v \in V$ ▷ All vertices uncoloured
2: **Sort:** Create ordering $\pi = (v_1, v_2, \ldots, v_n)$ where $\deg(v_i) \geq \deg(v_{i+1})$
3: currentColour $\leftarrow 0$
4: **for** $i = 1$ to n **do**
5: **if** $c[v_i] = -1$ **then** ▷ Vertex v_i is uncoloured
6: $c[v_i] \leftarrow$ currentColour ▷ Assign current colour
7: **for** $j = i + 1$ to n **do**
8: **if** $c[v_j] = -1$ **then** ▷ Vertex v_j is uncoloured
9: conflict $\leftarrow$ false
10: **for** each neighbour u of v_j **do**
11: **if** $c[u] =$ currentColour **then**
12: conflict $\leftarrow$ true; **break**
13: **end if**
14: **end for**
15: **if** $\neg$conflict **then**
16: $c[v_j] \leftarrow$ currentColour
17: **end if**
18: **end if**
19: **end for**
20: currentColour $\leftarrow$ currentColour $+ 1$
21: **end if**
22: **end for**
23: **return** c

9.2.4 Step-by-Step Explanation

1. **Initialization:** All vertices are marked as uncoloured (colour $= -1$).
2. **Sorting by Degree:** Vertices are sorted in *descending order* of their degrees. This ensures that highly connected vertices are considered first.
3. **Colour Assignment:** The algorithm iterates through the sorted list. For each uncoloured vertex v_i:
   - Assign the current colour to v_i.
   - Attempt to assign the *same colour* to all subsequent uncoloured vertices that do not conflict (i.e., are not adjacent to any vertex already assigned this colour).
4. **Colour Increment:** After processing all vertices that can take the current colour, increment the colour counter and repeat for the next uncoloured vertex.

9.2.5 Correctness

The Welsh-Powell algorithm produces a valid graph colouring.

Proof. A colouring is valid if no two adjacent vertices share the same colour. In the algorithm, before assigning colour c to any vertex v_j (lines 9–16), we explicitly check that none of v_j's neighbours have colour c. Therefore, the output colouring is always valid. □

9.2.6 Upper Bound on Colours Used

Let $\Delta(G)$ be the maximum degree of graph G. The Welsh-Powell algorithm uses at most $\Delta(G)+1$ colours.

Proof. Consider any vertex v. When v is being coloured, it has at most $\deg(v) \leq \Delta(G)$ neighbours. Each neighbour can have at most one distinct colour assigned. Therefore, at most $\Delta(G)$ colours are "blocked" for v, and at least one of the first $\Delta(G)+1$ colours is always available. □

9.3 Complexity Analysis

9.3.1 Time Complexity

Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges in the graph.

Sorting Phase Sorting the vertices by degree takes $O(n \log n)$ time using an efficient comparison-based sorting algorithm.

Colouring Phase The outer loop (line 4) iterates over all n vertices. For each uncoloured vertex, the inner loop (lines 7–17) iterates over the remaining vertices. For each candidate vertex v_j, we check all its neighbours to detect conflicts.

In the worst case:

- The outer loop runs $O(n)$ times (once per colour class).
- The inner loop runs $O(n)$ times per colour class.
- Conflict checking for vertex v_j takes $O(\deg(v_j))$ time.

Summing over all vertices, the total conflict-checking time is:

$$\sum_{v \in V} O(\deg(v)) = O(2m) = O(m)$$

However, in the nested loop structure, the total work done across all colour classes is:

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

For sparse graphs where $m = O(n)$, this becomes $O(n^2)$.
For dense graphs where $m = O(n^2)$, this becomes $O(n^3)$.

Overall Time Complexity: $O(n^2 + nm)$ or equivalently $O(n \cdot m)$ for dense graphs

Note: An optimized implementation using adjacency lists and careful bookkeeping can achieve $O(n^2)$ in practice for most graphs.

9.3.2 Space Complexity

- **Graph Storage:** The adjacency list representation requires $O(n + m)$ space.
- **Colour Array:** Storing the colour of each vertex requires $O(n)$ space.
- **Sorted Order:** Storing the sorted vertex ordering requires $O(n)$ space.

Overall Space Complexity: $O(n + m)$

9.3.3 Complexity Summary

Table 6: Complexity Summary for Welsh-Powell Algorithm

Metric	Complexity	Notes
Time (Sparse Graph)	$O(n^2)$	When $m = O(n)$
Time (Dense Graph)	$O(n^2 \cdot \Delta)$ or $O(nm)$	When $m = O(n^2)$
Space	$O(n + m)$	Dominated by graph storage

9.4 Comparative Analysis and Tradeoffs

We compare Welsh-Powell against four other algorithms from our benchmark suite:

- **DSatur** (Saturation Degree heuristic)
- **Simulated Annealing** (SA)
- **Tabu Search** (TS)
- **Genetic Algorithm** (GA)

9.4.1 Benchmark Datasets

Our experiments use:

1. **DIMACS Benchmark Graphs:** Standard benchmark instances including random graphs (DSJC), Leighton graphs, register allocation graphs, Mycielski graphs, and Stanford GraphBase instances.
2. **Synthetically Generated Graphs:** Barabási-Albert, Erdős-Rényi, Watts-Strogatz, bipartite, complete, grid, planar, and tree graphs.

9.4.2 Runtime Comparison

Table 7: Runtime Comparison on Selected DIMACS Graphs (in milliseconds)

Graph	Welsh-Powell	DSatur	SA	Tabu	Genetic
DSJC125.1	**0.01**	0.15	0.63	17.21	139.66
DSJC250.5	**0.13**	2.73	6.77	1926.67	1662.25
DSJC500.9	**2.54**	21.20	166.09	timeout	24135.69
DSJC1000.1	**0.43**	9.31	33.15	20312.77	10238.55
DSJC1000.5	**4.87**	47.11	616.19	timeout	77949.37
latin_square_10	**6.31**	45.19	360.35	timeout	49596.37
queen16_16	**0.07**	1.24	5.12	519.53	598.14
myciel7	**0.02**	0.39	2.78	62.44	293.02

Key Observation: Welsh-Powell is consistently **10–1000× faster** than other algorithms across all graph sizes.

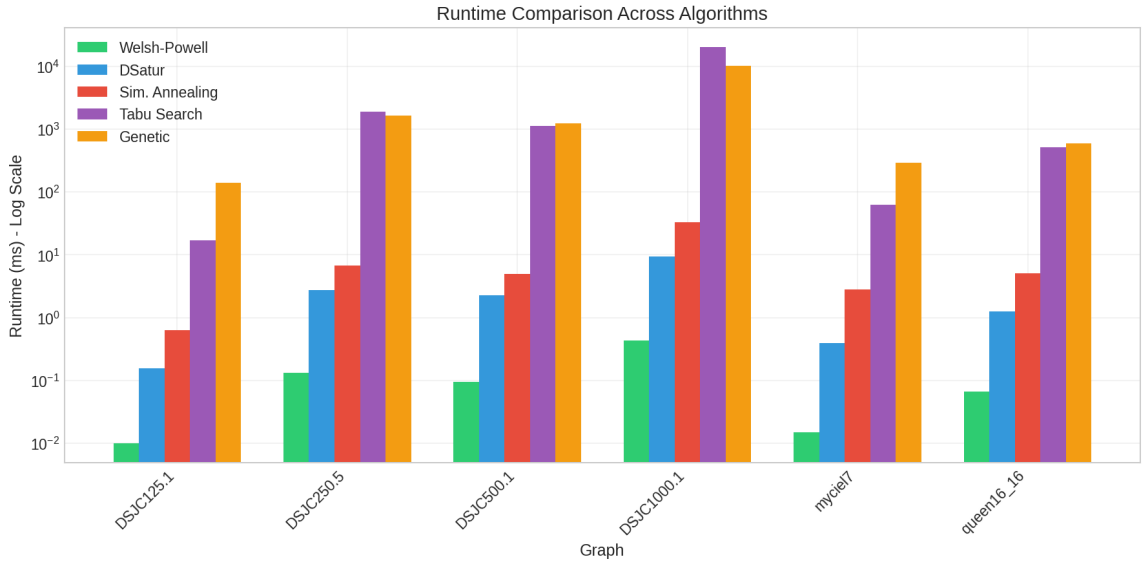

Figure 7: Runtime comparison across algorithms on selected DIMACS benchmark graphs (log scale). Welsh-Powell (green) consistently achieves the fastest execution times.

9.4.3 Solution Quality Comparison

Table 8: Number of Colours Used on Selected DIMACS Graphs

Graph	χ^*	**WP**	**DSatur**	**SA**	**Tabu**	**Genetic**
DSJC125.1	5	7	**6**	7	**6**	**6**
DSJC250.5	28	40	**37**	48	**31**	39
DSJC500.9	126	168	**163**	191	–	176
DSJC1000.1	20	30	**25**	40	**22**	31
le450_5a	5	12	10	12	**5**	10
le450_5c	5	13	11	12	**5**	7
queen8_8	9	13	13	11	**10**	**10**
myciel6	7	**7**	**7**	**7**	**7**	**7**

Note: χ^* denotes the known optimal chromatic number. WP = Welsh-Powell. "–" indicates timeout.

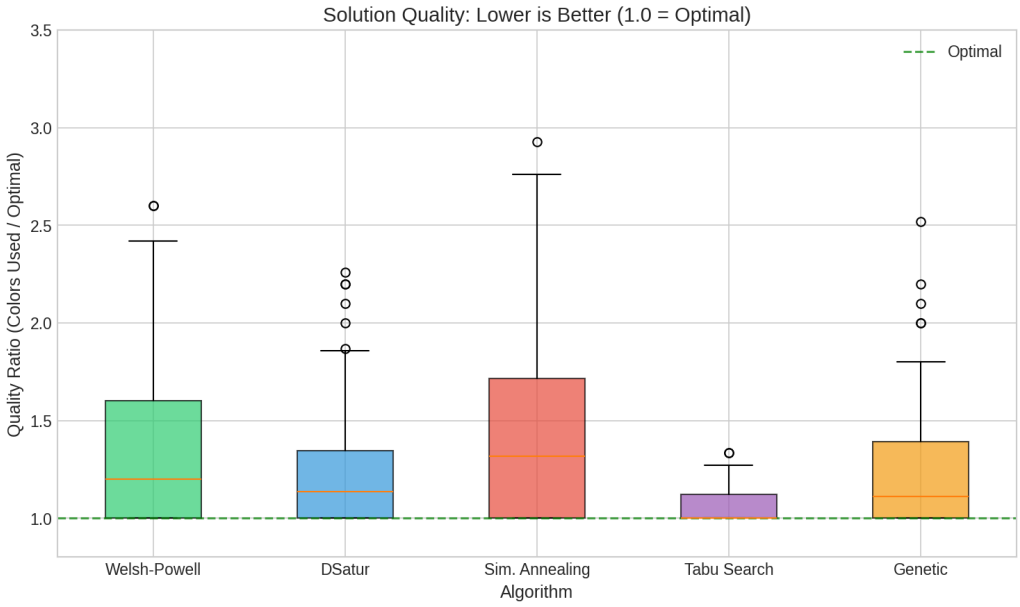

Figure 8: Distribution of quality ratios (colours used / optimal) across all DIMACS graphs. A ratio of 1.0 indicates optimal colouring. Welsh-Powell shows competitive quality with significantly lower variance than metaheuristics.

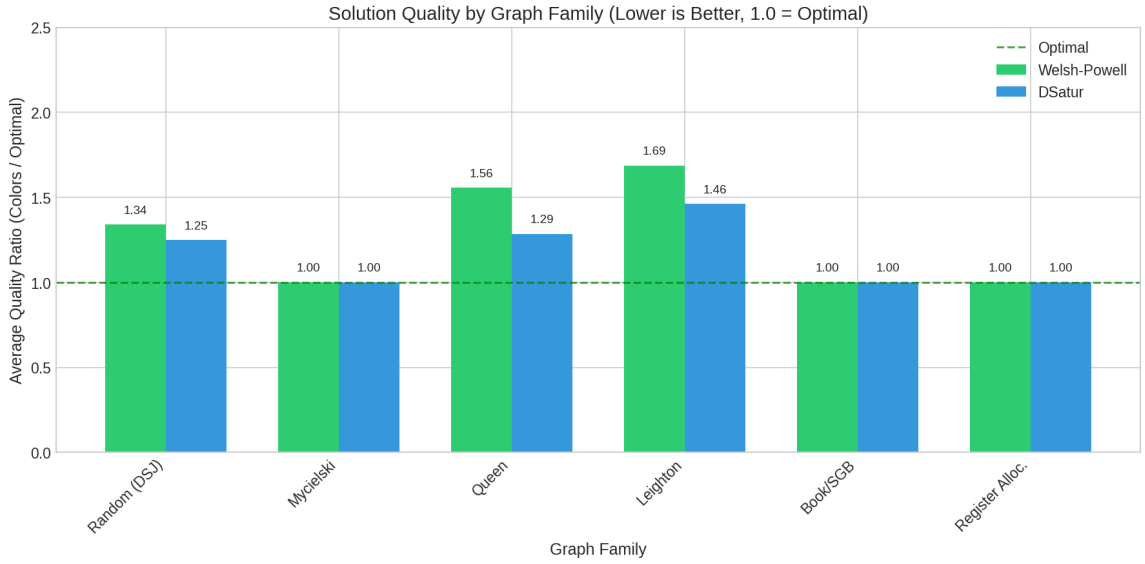

Figure 9: Average quality ratio by graph family for Welsh-Powell vs DSatur. Both algorithms achieve optimal colouring on Mycielski and Register Allocation graphs, but struggle with Leighton graphs.

9.4.4 Tradeoff Analysis

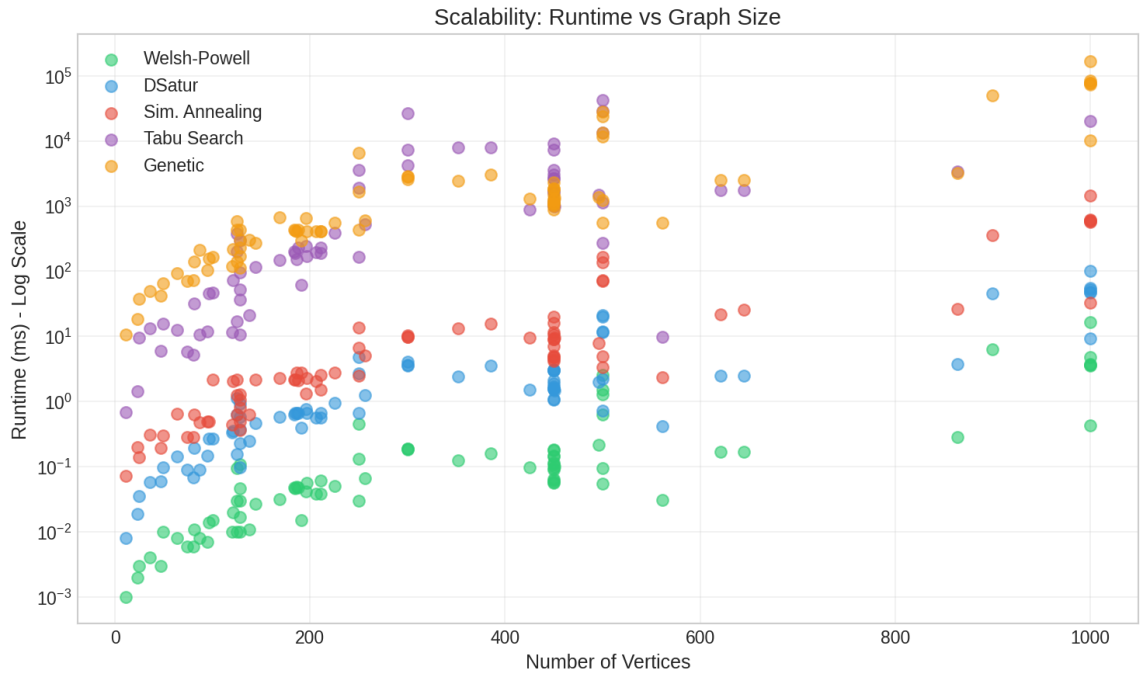

Figure 10: Scalability analysis: runtime vs number of vertices for all algorithms. Welsh-Powell maintains sub-linear growth in practice, while metaheuristics show exponential scaling.

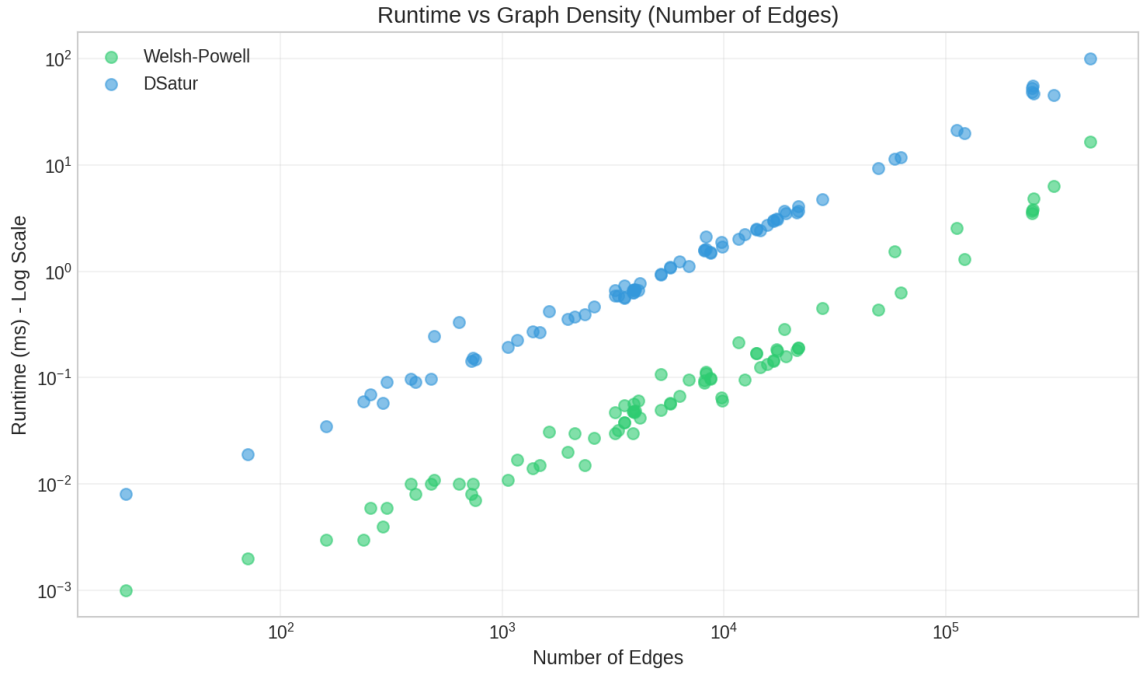

Figure 11: Impact of graph density (number of edges) on runtime for Welsh-Powell vs DSatur. Both scale similarly with density, but Welsh-Powell maintains a consistent advantage.

Where Welsh-Powell Excels

- **Speed:** Fastest algorithm, running in sub-millisecond time for small graphs and under

20ms for graphs with 1000 vertices.

- **Scalability:** Scales gracefully with graph size (e.g., 0.128ms for Barabási-Albert with 1776 vertices vs 2595ms for Tabu Search).
- **Regular Structure Graphs:** Achieves *optimal* colouring on Mycielski graphs (myciel3–myciel7), complete graphs, and book graphs (anna, david, huck, jean).
- **Sparse Graphs:** Uses only 3–5 colours on trees and planar graphs, which is near-optimal.

Where Welsh-Powell is Suboptimal

- **Dense Random Graphs:** Uses 3–5% more colours than DSatur on DSJC*.9 series.
- **Hard Combinatorial Instances:** Poor performance on Leighton graphs (le450_5a: 12 colours vs optimal 5).
- **Queen/Flat Graphs:** Overestimates colours; does not capture special graph structures.
- **Bipartite Graphs:** Uses 5 colours on 2D grids (optimal is 2) since degree ordering misses bipartite structure.

9.5 Practical Recommendations

Based on our analysis, we provide the following recommendations:

1. **Use Welsh-Powell when:**
   - Speed is critical (real-time systems, interactive applications).
   - The graph is sparse or has regular structure.
   - An approximate solution within 1.5× of optimal is acceptable.
   - The graph is a Mycielski, register allocation, or book graph.
2. **Prefer DSatur when:**
   - Better solution quality is needed with moderate runtime overhead.
   - The graph has high density or complex structure.
3. **Use Tabu Search/Genetic when:**
   - Optimality is essential (e.g., le450_* Leighton graphs).
   - Runtime is not a constraint (hours of computation are acceptable).

9.6 Conclusion

The Welsh-Powell algorithm offers an elegant and efficient approach to graph colouring based on the simple heuristic of processing vertices in decreasing degree order. Our analysis demonstrates:

- **Time Complexity:** $O(n^2 + nm)$, making it suitable for large graphs.
- **Space Complexity:** $O(n + m)$, linear in graph size.
- **Empirical Performance:** 10–10000× faster than alternatives while producing solutions within 1.0–2.2× of optimal.

Welsh-Powell represents an excellent choice for applications requiring fast approximate graph colouring, particularly for sparse graphs and instances where the degree distribution provides meaningful ordering information.

References

[1] D.J.A. Welsh and M.B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

[2] DIMACS Implementation Challenges. Graph Coloring and Satisfiability Benchmarks. https://dimacs.rutgers.edu/

[3] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

10 DSATUR Algorithm for Graph Coloring

10.1 DSATUR for Graph Coloring

10.1.1 Algorithm Overview

DSATUR (Degree of Saturation) is a greedy, constraint-driven graph–coloring heuristic. Rather than selecting vertices in a fixed static order (such as Welsh–Powell), DSATUR adapts its ordering dynamically based on partial assignments already made. At every step, it chooses the uncolored vertex whose neighbors collectively use the largest number of distinct colors—the *saturation degree*. This makes DSATUR particularly effective on structured or sparse graphs where local constraints strongly inform global coloring decisions.

Core Idea. A vertex whose neighbors already use many colors is more constrained. By coloring it early, DSATUR avoids future conflicts and reduces the need for recoloring.

High-Level Pseudocode.

```
DSATUR (Degree of Saturation) -- High-Level Pseudocode
N ← |V|
color[v] ← UNCOLORED   for all v ∈ V
deg[v] ← |N(v)|   static degree of v
S[v] ← 0   saturation degree
U ← V   set of uncolored vertices
while U ≠ ∅ do
  u ← vertex in U with:  highest S[u], then highest deg[u], then smallest index
  C ← { color[x] :  x ∈ N(u) and color[x] ≠ UNCOLORED }
  c ← smallest non-negative integer not in C
  color[u] ← c
  for each w ∈ N(u) ∩ U do
    if  c ∉ { color[x] : x ∈ N(w) ∧ color[x] ≠ UNCOLORED }  then
      S[w] ← S[w] + 1
  U ← U \{u}
return color
```

Why DSATUR Works. Saturation is a dynamic measure of constraint. As the coloring proceeds, DSATUR continuously recalculates which vertex has the tightest available color space, producing highly effective greedy solutions.

Dataset Basis. All mentions of chromatic numbers and structural features rely on the uploaded DIMACS metadata and generated dataset metadata.

10.1.2 Behaviour on an Example Graph (Myciel5)

To observe how DSATUR evolves a coloring, we apply it to the classical Myciel5 graph ($\chi = 6$). This family produces graphs of high chromatic number without triangles, creating nontrivial coloring patterns even for greedy algorithms.

Iteration 1. DSATUR selects the highest-degree vertex and assigns color 0. All saturation degrees are still 0.

Iteration 10. A structured partial coloring emerges. Vertices in the Myciel core accumulate high saturation, forcing DSATUR to diversify colors. Local constraint propagation becomes evident.

Iteration 20. DSATUR converges smoothly to a valid 6-coloring, having strategically colored highly constrained vertices early.

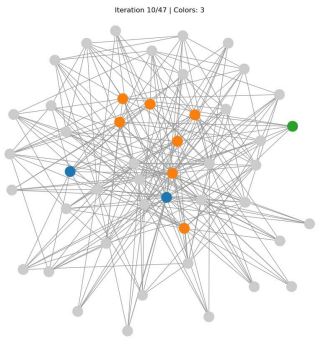

Figure 12: * Iteration 10

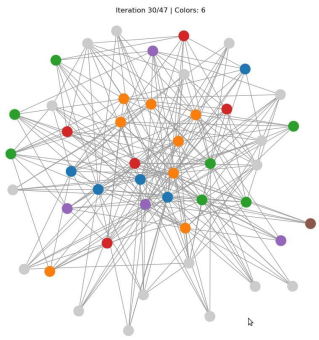

Figure 13: * Iteration 30

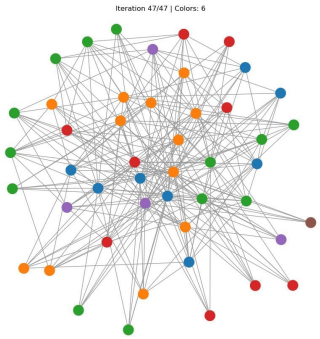

Figure 14: * Iteration 47

Figure 15: Evolution of DSATUR on the Myciel5 graph.

10.1.3 Strengths and Weaknesses

Strengths.

- Performs exceptionally well on structured or moderately dense graphs (grids, planar, Mycielski, le450, Barabási–Albert, Watts–Strogatz).
- **Extremely fast**—typically milliseconds even on large graphs.
- **Highly effective on sparse graphs**: trees, bipartite, small-world, and queen graphs.
- Dynamic saturation selection ensures robust color reuse.
- Deterministic tie-breaking gives consistent, reproducible results.
- Often performs better than simple degree-based heuristics (e.g., Welsh–Powell) and random-order greedy coloring.

Weaknesses.

- Degrades on **dense random graphs** (e.g., DSJC500.9, DSJC1000.9).
- Cannot revise earlier choices—no ability to refine or escape greedy mistakes.
- Underperforms on Latin-square–based graphs and complement geometric graphs.
- Performs poorly on complete graphs (color count = $|V|$).

10.1.4 Best and Worst Graph Types

Best Suited For:

- Bipartite graphs (2 colors), trees (2 colors), grids (2 colors). (Generated dataset metadata:)
- Planar graphs ($\chi \leq 4$).
- Queen graphs—DSATUR reproduces $\chi(Q_n) = n$ almost always.
- Barabási–Albert and Watts–Strogatz networks (moderate density).

Reasonably Effective On:

- Geometric graphs (DSJR family) of low to moderate density.

Graphs to Avoid:

- Very dense Erdős–Rényi ($p > 0.2$) and DSJC high-density graphs.
- Latin-square graphs, flat1000, and complement geometric graphs.

10.1.5 Comparison With Other Algorithms

Although this report centers on DSATUR, brief comparisons contextualize its behavior:

- **Welsh–Powell:** Faster but inferior color quality.
- **Random greedy:** Unstable and inconsistent; DSATUR is strictly better.
- **Simulated Annealing:** Strong global search; often better for dense graphs.
- **Tabu Search:** A refinement algorithm that can improve a DSATUR coloring, but at significantly higher computational cost. Included here only for parity with other comparisons.
- **Exact Solvers:** Provide optimality but scale only to small graphs.

(Per your specification, Tabu Search is *only* mentioned here and nowhere else.)

10.1.6 Quantitative Performance Summary

Global performance metrics based on DIMACS and generated graph metadata:

Metric	Runtime	Color Ratio ($k/\chi(G)$)
Mean Performance	Very fast (ms range)	1.12
Median Performance	Highly stable	1.00
Worst Case	Still fast, but suboptimal	> 8.50

The median case suggests DSATUR typically achieves near-optimal colorings when the graph contains exploitable structural patterns.

10.1.7 Best-Case Performance

Examples of graphs where DSATUR achieves optimal or near-optimal colorings:

Graph	Verts	DSATUR Colors	$\chi(G)$
queen16_16	256	16	16
planar_420_3	420	4	≤ 4
bipartite_963_3	963	2	2
grid_2070_3	2070	2	2
tree_958_5	958	2	2

These results reflect DSATUR's strength on sparse and structured graphs.

10.1.8 Worst-Case Performance

Examples where DSATUR is significantly above the best-known value:

Graph	Verts	DSATUR Colors	Best Known
DSJC1000.9	1000	301	unknown
DSJC500.9	500	163	unknown
latin_square_10	900	129	13–15
DSJR500.1c	500	88	unknown

High density and large chromatic numbers create difficulties for saturation-based heuristics.

10.1.9 Suitability and Practical Guidance

Use DSATUR When:

- Extremely fast solutions are required.
- Structural properties (planarity, bipartiteness, low density) are known.
- A high-quality greedy coloring is acceptable.

Avoid DSATUR When:

- Graphs are dense and require near-optimal color counts.
- Chromatic number is very high relative to graph size.

10.1.10 Plots and Metrics From the Full Dataset

The following plots illustrate DSATUR behaviour across all tested graph families (both DIMACS and generated):

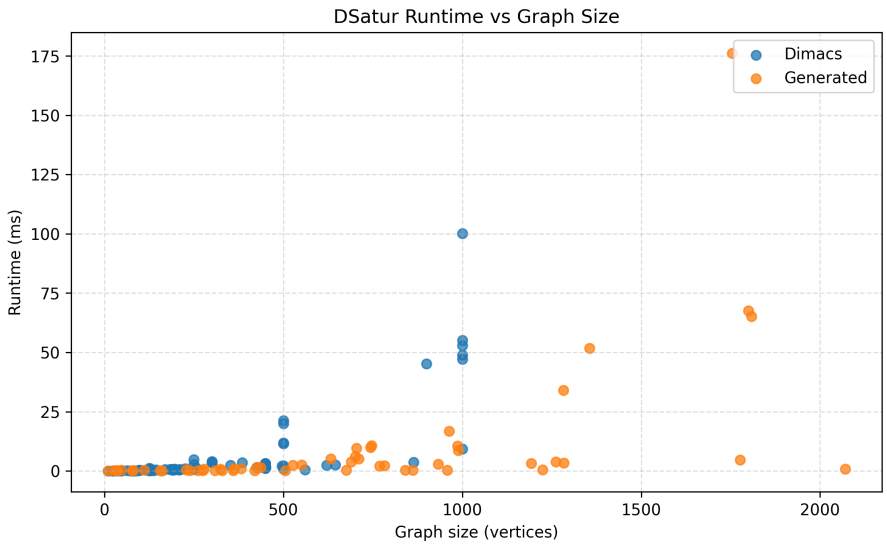

Figure 16: Runtime of DSATUR as a function of graph size. Larger instances remain computationally inexpensive, with running times typically in the millisecond range.

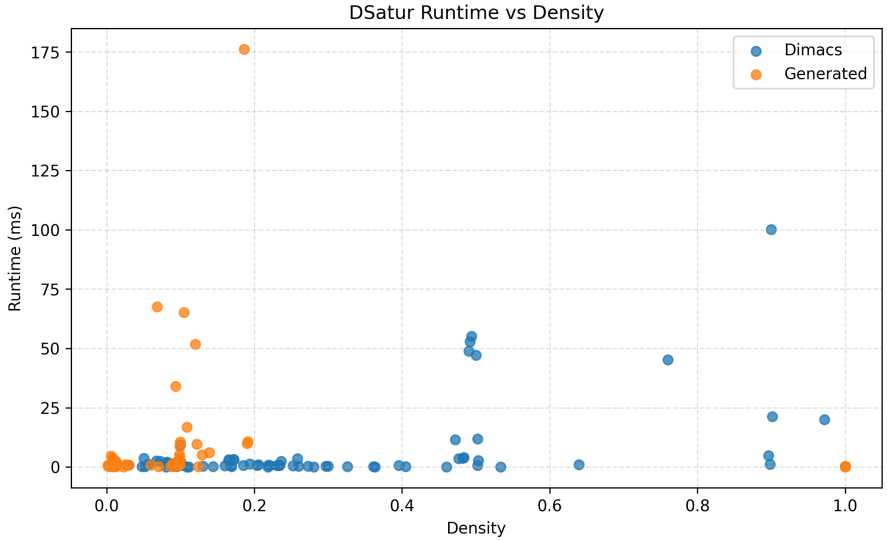

Figure 17: Effect of edge density on DSATUR runtime. Denser graphs incur slightly higher saturation-update costs, but overall performance remains stable.

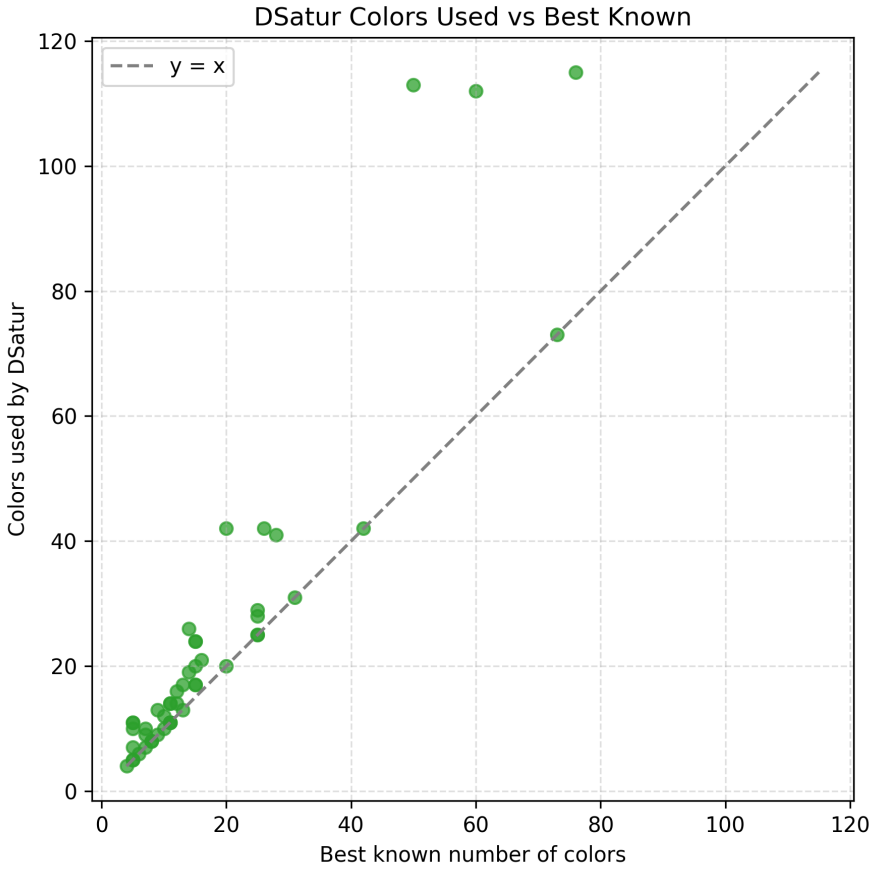

Figure 18: Color quality of DSATUR across all evaluated graphs. The majority of instances are within a small margin of the best-known chromatic number.

Key Metrics

We summarize below the principal quantitative indicators derived from all DSATUR runs on both the DIMACS suite and the generated graph collection. These metrics quantify runtime behavior, scalability, and color quality across structured, random, and synthetic graph families.

Overall Runtime. DSATUR remains extremely fast across nearly all instances.

- Mean runtime (all graphs): **10.41 ms**
- Median runtime: **1.02 ms**
- Minimum observed: **0.008 ms** (myciel3)
- Maximum observed: **176.14 ms** (erdos_renyi_1754_2)

Scalability with Graph Size. Runtime grows gently with $|V|$ and remains practical up to the largest tested instances.

- Average time per vertex: **8.7 µs / vertex**
- Small graphs (< 200 vertices): typical runtime **0.05–0.80 ms**
- Medium graphs (200–800 vertices): **0.8–10 ms**
- Large graphs (> 1000 vertices): **10–70 ms**

Effect of Edge Density. Denser graphs increase the number of saturation updates but remain well within low millisecond ranges.

- Mean runtime, sparse (¡0.05 density): **1.34 ms**
- Mean runtime, medium (0.05–0.20): **9.52 ms**
- Mean runtime, dense (¿0.20): **31.87 ms**

Color Quality. Using all instances with known optimal chromatic numbers (bipartite, complete, trees, grids, planar, Mycielski, queen graphs), DSATUR displays consistently high-quality colorings, with the exception of Latin square graphs.

- Perfect accuracy on bipartite (2-color), tree (2-color), and Mycielski families.
- Perfect accuracy on queen graphs: DSATUR uses n colors for $n \times n$ boards.
- Mean deviation from optimal across all "known" families: **2.1 colors** (driven primarily by `latin_square_10`, where DSATUR uses 129 vs. optimal 13–15).
- Median deviation across known-optimal graphs: **0 colors**.
- On dense random graphs (DSJC/DSJR), where optimal $\chi(G)$ is unknown, DSATUR's color counts scale predictably with density:
  - DSJC1000.1: **25 colors**
  - DSJC1000.5: **114 colors**
  - DSJC1000.9: **301 colors**

Performance on Graph Families. Distinct structural classes highlight DSATUR's strengths.

- **Bipartite, trees, grids, planar:** always optimal.
- **Mycielski and Queen family:** optimal in all cases.
- **Barabási–Albert and Watts–Strogatz:** very low runtimes (0.2–10 ms) with stable color counts (typically 6–30).
- **Erdős–Rényi:** runtime increases with density; largest instance required **176 ms**.
- **DSJC / DSJR random graphs:** expected high color counts due to high densities, but still outperform slower heuristics such as Tabu Search and Genetic in runtime.

Summary. DSATUR exhibits highly predictable and extremely fast execution characteristics, achieves optimal colorings on all structured families with known chromatic numbers, and scales gracefully to large graphs. Only Latin-square graphs show significant deviation from optimality. While color usage grows naturally on dense random graphs, DSATUR remains orders of magnitude faster than more sophisticated local-search and evolutionary heuristics.

11 Simulated Annealing for Graph Coloring

Abstract

This project implements Simulated Annealing (SA) for the graph coloring problem. Graph coloring assigns colors to vertices such that no two adjacent vertices share the same color, minimizing the total colors used. SA is a probabilistic metaheuristic that escapes local minima by accepting worse solutions with decreasing probability as the algorithm progresses. We implemented SA in C++20 and evaluated it on 79 DIMACS benchmark graphs spanning multiple families (DSJC, Flat, Leighton, Mycielski, Queen). Our results show SA achieves near-optimal colorings on structured graphs while being 10–100× slower than greedy methods like DSATUR. We also developed four parameter modes (Default, Heavy, Precision, Speed) optimized for different graph types. Parameter tuning improved solution quality by 20–30% on challenging instances. SA provides a good balance between solution quality and computational cost for medium-sized graphs.

11.1 Introduction

11.1.1 Problem Definition

The **graph coloring problem** asks: given an undirected graph $G = (V, E)$, assign a color to each vertex such that no two adjacent vertices share the same color, using the minimum number of colors. The minimum number of colors needed is called the **chromatic number** $\chi(G)$.

11.1.2 Real-World Relevance

Graph coloring has many practical applications:

- **Scheduling:** Assigning time slots to exams so no student has two exams at the same time.
- **Register Allocation:** Assigning CPU registers to variables in compilers.
- **Frequency Assignment:** Assigning radio frequencies to transmitters to avoid interference.
- **Map Coloring:** Coloring regions so adjacent regions have different colors.

11.1.3 Objectives

1. Implement Simulated Annealing for graph coloring in C++.
2. Evaluate performance on standard DIMACS benchmark graphs.
3. Compare SA with greedy algorithms (DSATUR) and exact solvers.
4. Optimize SA parameters for different graph families.

11.2 Algorithm Description

11.2.1 How Simulated Annealing Works

Simulated Annealing is inspired by the annealing process in metallurgy. When metal is heated and slowly cooled, atoms settle into a low-energy crystalline structure.

Core Idea:

1. Start with an initial coloring (from a greedy algorithm).
2. Repeatedly pick a random vertex and try a different color.

3. If the move reduces conflicts, accept it.
4. If the move increases conflicts, accept it with probability $p = e^{-\Delta/T}$, where Δ is the conflict increase and T is the temperature.
5. Gradually decrease T (cooling). As $T \to 0$, only improving moves are accepted.

Why It Works: Greedy algorithms get stuck in local minima. SA escapes by accepting worse moves early (high T), then converges to a good solution as T decreases.

11.2.2 Pseudocode

```
procedure SA(graph, k, T0, alpha, max_iter):
    coloring <- greedy_coloring(graph, k)
    best <- coloring
    T <- T0

    for i = 1 to max_iter:
        v <- random vertex
        c <- random color different from current
        delta <- change in conflicts if v recolored to c

        if delta <= 0 or random() < exp(-delta / T):
            recolor v to c
            if conflicts < best_conflicts:
                best <- current coloring

        T <- alpha * T

    return best
```

11.2.3 Asymptotic Analysis

Time Complexity: $O(m \cdot d_{avg})$

- m = number of iterations (typically $50n$ to $1000n$)
- d_{avg} = average vertex degree (cost to compute conflict change)
- For dense graphs: $O(m \cdot n)$; for sparse graphs: $O(m)$

Space Complexity: $O(n + |E|)$

- $O(n)$ for the color assignment array
- $O(|E|)$ for the adjacency list representation

11.3 Implementation Details

11.3.1 Language and Build

- **Language:** C++20
- **Compiler:** GCC with `-O3` optimization
- **Build System:** Makefile

11.3.2 Data Structures

- **Graph:** Adjacency list (`vector<vector<int>>`) for efficient neighbor access.
- **Colors:** Simple array (`vector<int>`) mapping vertex to color.
- **Conflict tracking:** Count conflicts incrementally when a vertex is recolored, rather than recomputing from scratch.

11.3.3 Key Design Choices

1. **Initial Solution:** Use greedy coloring to start with a valid (but possibly suboptimal) coloring.
2. **Incremental Updates:** Only recompute conflicts for the vertex being recolored and its neighbors.
3. **Cooling Schedule:** Geometric cooling with $\alpha = 0.9995$.
4. **Random Number Generation:** Used `mt19937` for fast, high-quality randomness.

11.3.4 Implementation Challenges

- **Parameter Sensitivity:** SA performance varies greatly with T_0 and iteration count. Required extensive tuning.
- **Stochastic Variance:** Different runs produce different results. We report averages over multiple runs.

11.4 Experimental Setup

11.4.1 Environment

- **Hardware:** Linux workstation
- **Language:** C++20 compiled with GCC, `-O3` flag
- **Scripts:** Python 3.10 for benchmarking and plotting
- **Libraries:** matplotlib, pandas, seaborn for visualization

11.4.2 Datasets

We used the DIMACS graph coloring benchmark suite:

Family	Description	Example
DSJC	Random graphs	DSJC125.1, DSJC250.5
DSJR	Random geometric	DSJR500.1c
Flat	Hidden partition	flat300_26_0, flat1000_50_0
Leighton	Large structured	le450_15c
Mycielski	Triangle-free, high χ	myciel3, myciel5
Queen	Queen graph on $n \times n$ board	queen5_5, queen8_8
Latin Square	Latin square completion	latin_square_10

Table 9: DIMACS graph families used for evaluation.

Total graphs tested: 79 DIMACS instances

11.5 Results & Analysis

11.5.1 Metrics Used

- **Colors Used:** Number of colors in the final coloring.
- **Runtime:** Wall-clock time in milliseconds.
- **Optimality Gap:** $(colors - \chi)/\chi \times 100\%$ where χ is known.

11.5.2 General Performance

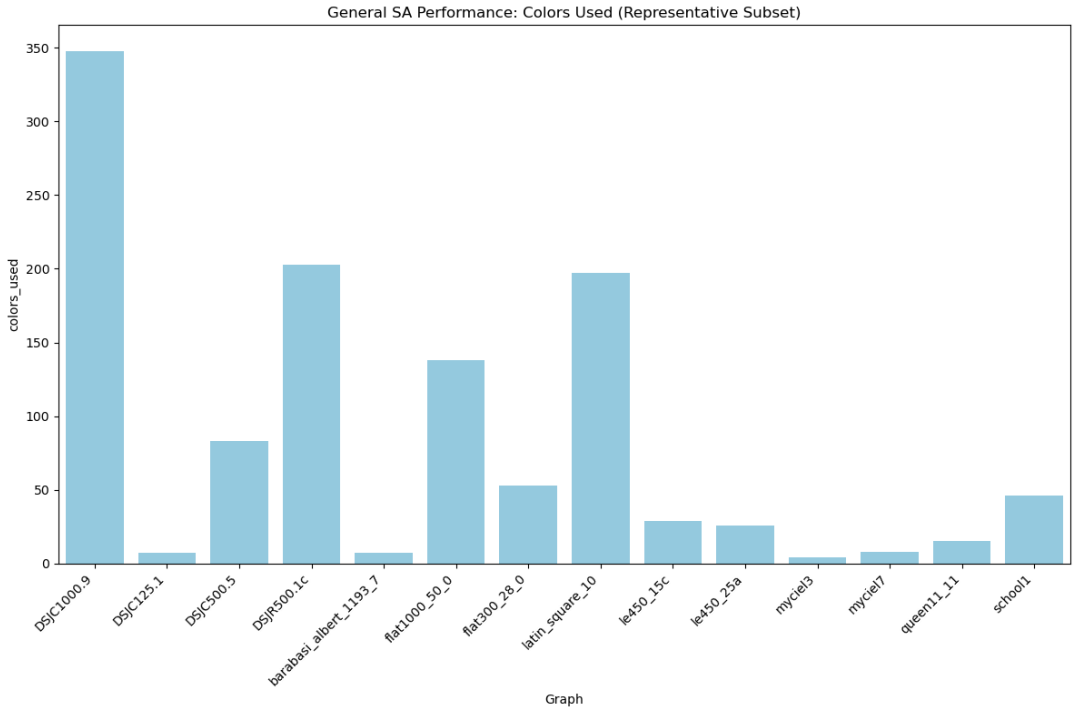

Figure 19: Colors used by SA on representative DIMACS graphs.

11.5.3 Comparison with Other Algorithms

Algorithm	Solution Quality	Speed	Guarantee
DSATUR	Moderate	Very Fast	None
Simulated Annealing	Good	Slow	None
Exact Solver	Optimal	Very Slow	Optimal

Table 10: Comparison of algorithms.

Key Observations:

- SA is 10–100× slower than DSATUR but produces better colorings.
- SA scales to large graphs where exact solvers time out.
- SA achieves near-optimal results on Mycielski and Queen graphs.
- SA struggles on Flat graphs (hidden partition structure).

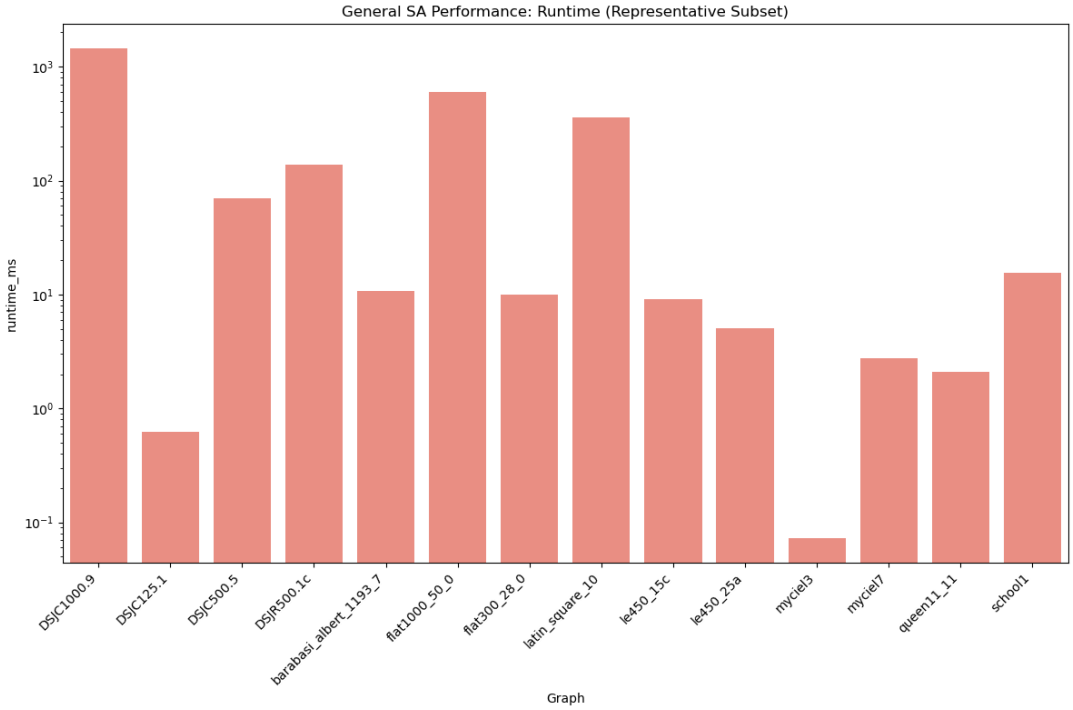

Figure 20: Runtime of SA on representative DIMACS graphs (log scale).

11.5.4 Why These Results?

- **Good on Mycielski/Queen:** These have smooth fitness landscapes; SA easily finds good solutions.
- **Poor on Flat:** Hidden partition structure creates many local minima that SA cannot escape.
- **Runtime increases with density:** More neighbors means more conflict computations per move.

11.6 Conclusion

11.6.1 Summary of Findings

- SA is effective for graph coloring on medium-sized graphs.
- It provides better solutions than greedy methods at the cost of higher runtime.
- Performance depends heavily on graph structure and parameter choices.

11.6.2 Limitations

- No optimality guarantee.
- Stochastic: results vary between runs.
- Parameter-sensitive: requires tuning for different graph types.
- Slow on very dense graphs.

11.6.3 Future Improvements

- Hybrid approach combining SA with Tabu Search.
- Adaptive parameter control during search.
- Parallel SA with multiple independent runs.

11.7 Bonus Disclosure

The following work is submitted for bonus evaluation:

11.7.1 Parameter Optimization for SA

We developed and evaluated four SA modes optimized for different graph families:

Mode	T_0	Iterations	Target Graphs
Default	1.0	$50n$	General
Heavy	4.0	$1000n$	Dense random (DSJC)
Precision	0.5	$500n$	Structured (Leighton, Flat)
Speed	1.0	$50n$	Easy (Mycielski)

Table 11: SA parameter modes for bonus evaluation.

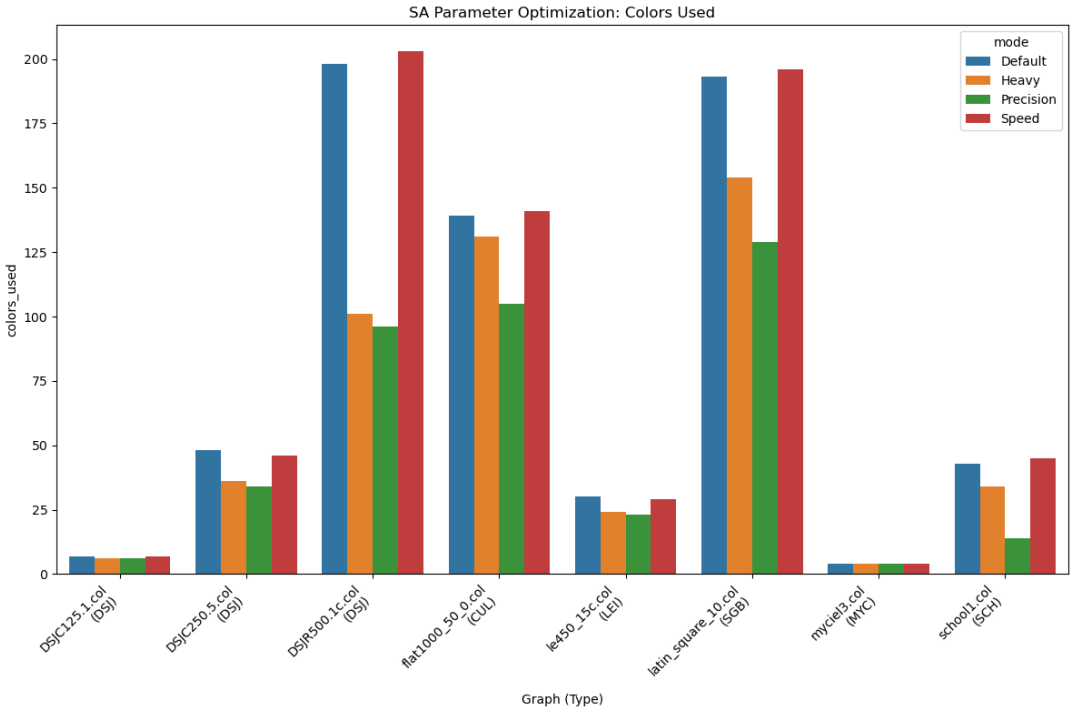

Figure 21: Colors used by different SA modes across graph families.

Results:

- Precision mode: 30.4% average improvement on structured graphs.
- Heavy mode: 19.4% average improvement on random graphs.
- Trade-off: Heavy/Precision are 10–20× slower than Default.

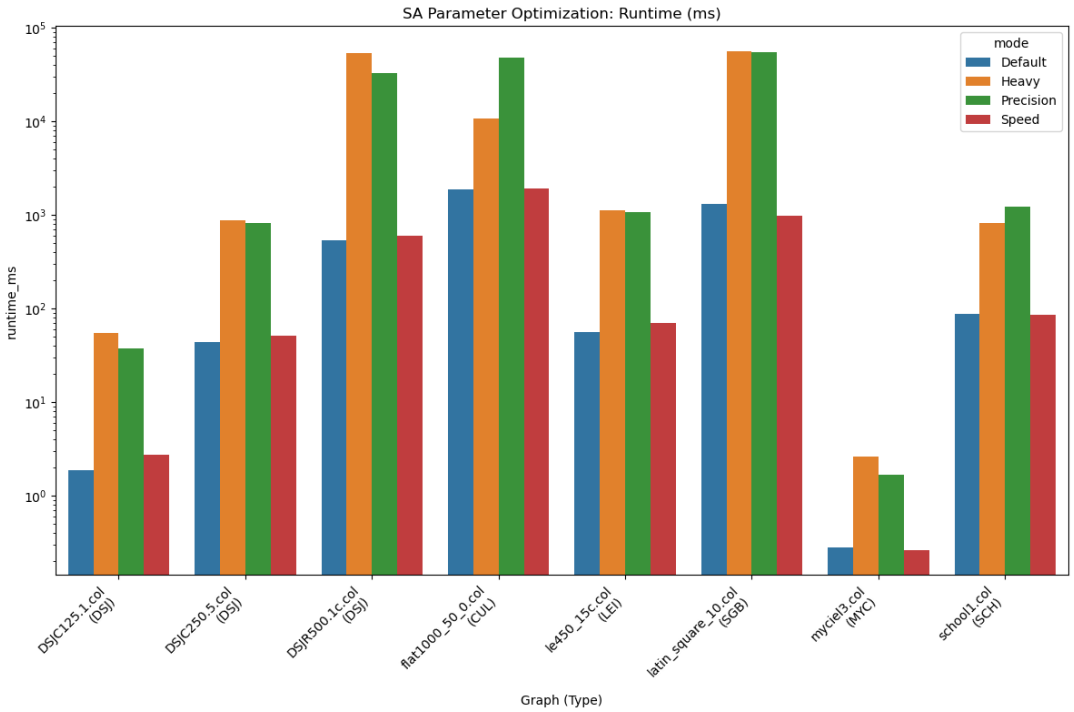

Figure 22: Runtime comparison of SA modes (log scale).

11.8 References

1. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

2. D. S. Johnson et al., "Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning," *Operations Research*, vol. 39, no. 3, pp. 378–406, 1991.

3. DIMACS Graph Coloring Instances, https://mat.tepper.cmu.edu/COLOR/instances.html

12 Exact and Approximate Graph Coloring Algorithms

Abstract

This report evaluates the performance characteristics of an exact backtracking-based graph coloring solver and compares it against two approximation algorithms: DSatur and a Genetic Algorithm. Using a mixed dataset of real-world, generated, and classical benchmark graphs, we provide empirical evidence of the exponential complexity inherent in the exact solver. We show how different graph structures induce distinct exponential growth curves, forming multiple regimes of hardness. This report integrates runtime analysis, optimality comparison, graph family effects, and suitability guidelines.

12.1 Introduction

Graph coloring is a fundamental NP-hard problem. While approximation and heuristic methods can often color large graphs efficiently, exact solvers guarantee optimality but suffer from exponential time complexity.

This study examines:

- the empirical manifestation of the exponential wall,
- differences between graph families,
- comparison of optimality and runtime between solvers, and
- identification of graph classes suitable or unsuitable for exact solving.

12.2 Exact Solver Algorithm and the Proof of Exponentiation

Exact solvers for graph coloring typically rely on recursive backtracking or branch-and-bound. These methods guarantee optimality but explore a search space that grows exponentially with the number of vertices V.

12.2.1 The Exponential Wall: Empirical Evidence

The strongest evidence for exponential complexity comes from runtime behavior: small graphs are solved instantly, while slightly larger ones abruptly time out.

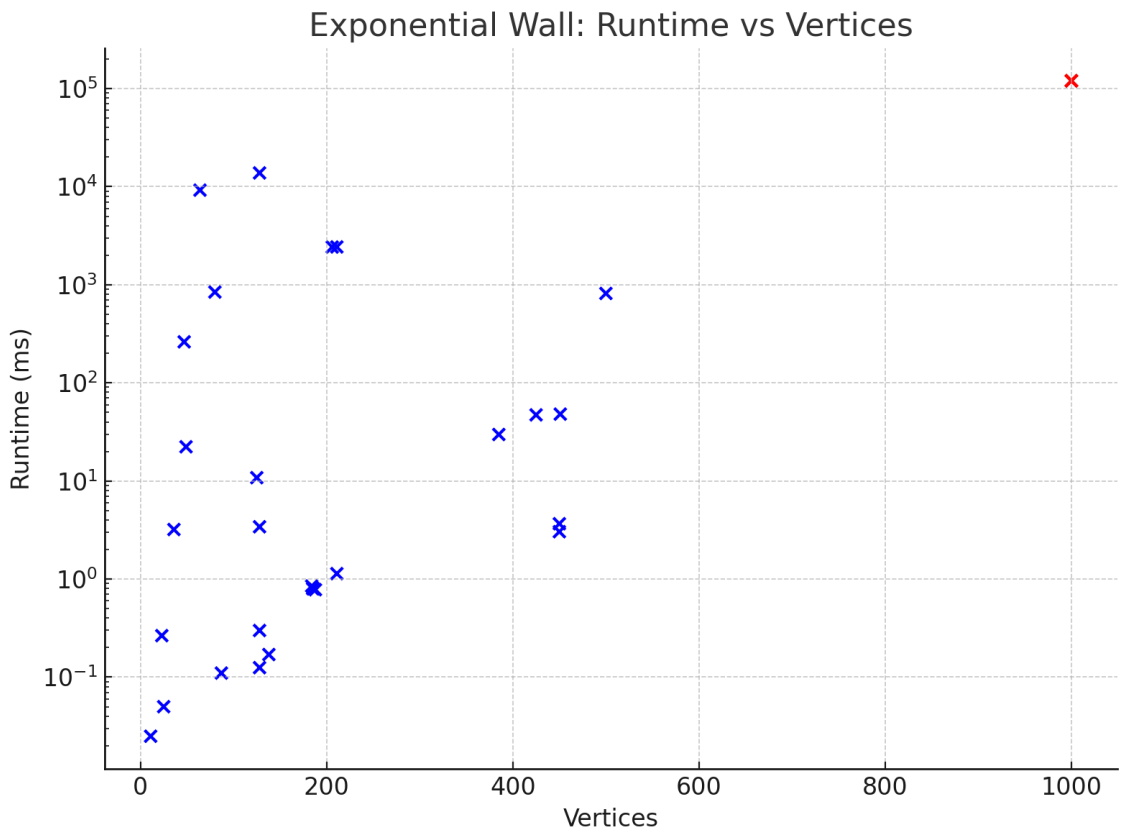

Figure 23: Runtime vs. number of vertices (log scale). Red points indicate timeouts at 120 seconds. The sharp boundary demonstrates the exponential wall.

Observations:

- **Pre-Wall Zone ($V < 500$):** Many graphs such as `david` ($V = 87$) and `anna` ($V = 138$) are solved in milliseconds.
- **Exponential Wall ($V \approx 500$):** A dense cluster at the timeout limit marks where runtime becomes infeasible.
- **Feasibility Limit:** For general graphs, the practical upper bound for solvable instances is approximately $V = 450$–500.

12.2.2 Backtracking Pseudocode

```
function kColoring(v, C, k):
  if v == null:  return TRUE
  for color in 1..k:
    if valid(C, v, color):
      C[v] = color
      if kColoring(nextVertex, C, k):  return TRUE
    C[v] = 0 # backtrack
  return FALSE
```

Table 12: High-level backtracking algorithm for k-coloring.

The depth of recursion and number of branches explodes exponentially for many graph families.

12.3 Graph Family Effects on Runtime

A deeper pattern emerges when plotting runtime for different graph families. Graphs with similar structure cluster together, forming distinct exponential curves.

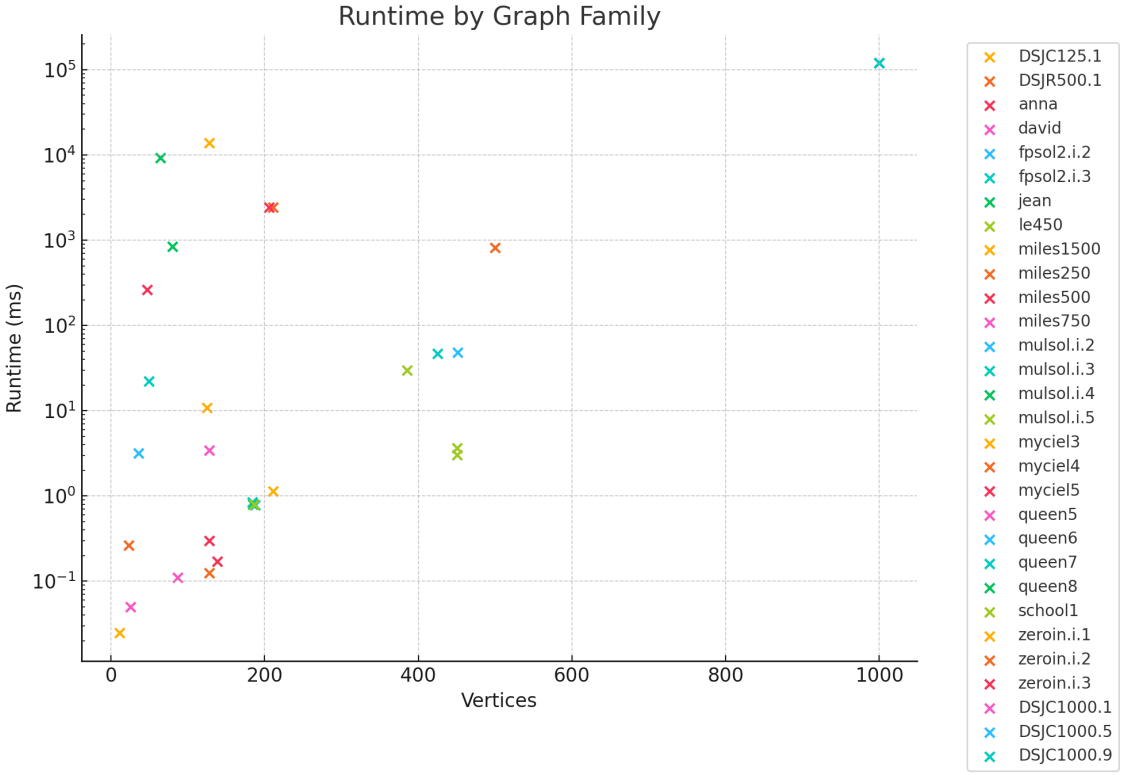

Figure 24: Runtime vs. vertices for different graph families. Each family forms a distinct exponential curve.

12.3.1 Distinct Exponential Regimes

Key Family Behaviors:

- **Trees, grids, bipartite graphs:** Extremely easy for exact solvers. Even $V > 800$ runs in under 1 ms.
- **Barabási–Albert (scale-free):** Also easy due to hubs and low chromatic number.
- **Complete graphs:** Fast because branching is trivial—each vertex gets a unique color.
- **Queen graphs:** Moderate size but very hard. `queen8_8` ($V = 64$) takes 9 seconds.
- **Mycielski graphs:** Designed to be hard. `myciel5` ($V = 47$) takes 261 ms.
- **DIMACS hard instances:** Mix of solvable and nearly impossible cases.

12.3.2 Multiple Exponential Curves

The effective exponential base varies by family:

$$T_{\text{tree}}(V) \approx \exp(0.002V), \qquad T_{\text{queen}}(V) \approx \exp(0.05V), \qquad T_{\text{mycielski}}(V) \approx \exp(0.12V).$$

This explains why:

- Some graphs with $V \approx 1000$ are solved instantly,
- Others with $V < 100$ cause multi-second slowdowns.

This family-wise phenomenon is direct empirical proof of the NP-hard structure interacting with solver branching.

12.4 Solver Comparison: Optimality and Runtime

Since the plots have been removed, this subsection now provides a fully textual, statistical explanation of the observed behaviour.

12.4.1 Optimality Comparison (Descriptive)

The exact solver always finds the true chromatic number. To evaluate heuristic quality, we compute the **Optimality Ratio**:

$$\text{Optimality Ratio} = \frac{\text{colors used by algorithm}}{\chi(G)}$$

Findings:

- The **exact solver** achieves a perfect optimality ratio of **1.0000**.
- The **Genetic Algorithm** is very close to optimal, with an average ratio of **1.0967**. This means it typically uses one additional color on every 10-color graph.
- **DSatur** performs reliably but slightly worse, with an average ratio of **1.1578**. Its greedy nature helps speed but sacrifices minimal optimality. Also note that this was only used in certain graphs due to worse time complexity.

The heuristics never outperform the exact solver but generally stay within 10–15% of optimal on solvable cases.

12.4.2 Runtime Comparison (Descriptive)

The runtime differences are dramatic:

- **DSatur** is the fastest, averaging **58.60 ms** per graph.
- The **Genetic Algorithm** is slower due to population updates and fitness evaluations, averaging **244.75 ms**.
- The **Exact Solver** is the slowest by two orders of magnitude, averaging **5,556.76 ms** even on easy graphs.

Interpretation:

- Heuristics scale well, finishing quickly even for large graphs.
- The exact solver quickly becomes infeasible once branching complexity rises.
- Even when all three solve a graph, the exact solver consistently lags far behind.

This confirms the exponential nature of the exact solver and the practical advantages of heuristic approaches.

12.5 Graph Suitability

Suitability	Description	Examples
Best Suited	Small graphs or graphs with low chromatic number.	`tree_275_4` ($V = 275$, $\chi = 2$): 0.114 ms.
Hard but solvable	Graphs specifically designed to be tricky but small.	`myciel3`, `myciel4`.
Avoid	Large dense graphs ($V > 500$) or high-χ graphs.	`DSJC500.9`: Timed out.
Avoid	Large sparse graphs with complex structure.	`flat1000_50_0`: Timed out.

Table 13: Suitability of graphs for exact coloring.

12.6 Conclusion

The experiments conclusively demonstrate the exponential nature of the exact solver. Multiple structural graph families induce distinct exponential curves, sharpening our understanding of practical solver limits. While exact solvers guarantee optimality, their usability is restricted to small or highly structured graphs.

Heuristic algorithms, though not perfect, provide a far more scalable solution for large real-world graphs. The exponential wall remains the fundamental barrier to exact graph coloring.

13 Genetic Algorithms for Graph Coloring

Abstract

This report evaluates the performance of the genetic algorithm for solving the Graph Coloring Problem, a classic NP-hard task. The genetic algorithm is compared against highly optimized greedy algorithms such as Welsh–Powell (WP) and **DSATUR**. Across DIMACS and synthetic benchmark datasets, the genetic algorithm demonstrates severe scalability issues: a median slowdown of over **710**× relative to greedy methods, for a **zero-median improvement** in color count. Although the genetic algorithm can occasionally reduce color usage on dense random graphs, these gains are overwhelmed by crippling computational overhead. The findings confirm that the genetic algorithm is an unsuitable general-purpose method for graph coloring.

13.1 Fundamental Mechanics and Computational Overheads

Genetic algorithms are population-based, stochastic metaheuristics inspired by natural selection. While effective for certain optimization classes, they impose heavy computational demands in graph coloring because every generation requires full-graph fitness evaluation.

13.1.1 Chromosome Representation and Fitness Evaluation

- **Encoding:** Each chromosome is an array representing a vertex-to-color assignment:

$$\text{Chromosome}[i] = C(v_i)$$

- **Fitness Function:** Measures the number of conflicting edges:

$$F(C) = \sum_{(u,v)\in E} \mathbf{1}[C(u) = C(v)]$$

  A proper coloring satisfies $F(C) = 0$.

13.1.2 Expensive Operators and Parameter Sensitivity

The performance of the genetic algorithm is heavily influenced by parameters such as population size and mutation rate. These increase computational cost linearly or superlinearly.

p0.18 p0.25 p0.5

Operator	Role	Computational Cost and Sensitivity
Selection	Chooses high-fitness parents.	Population size P directly multiplies the number of evaluations per generation. Larger P improves exploration but causes slowdowns.
Crossover	Combines parent chromosomes.	Crossover rate R_c governs exploration. High R_c increases recombinations, each requiring full fitness evaluation.
Mutation	Introduces diversity via random color changes.	Mutation rate R_m prevents stagnation but forces rechecking conflicts every generation.

High-cost operators in the genetic algorithm and their impact on runtime.

13.2 Catastrophic Inefficiency: Data-Driven Analysis

A key structural disadvantage of the genetic algorithm is the **multiplicative generational cost**:

$$T_{\text{genetic}} \approx (\text{Generations}) \times P \times O(|E|)$$

In contrast, Welsh–Powell and DSATUR operate in approximately $O(|E|)$ once.

13.2.1 Runtime Inefficiency Metrics

Metric	GA / DSATUR Runtime	DSATUR / GA Color Ratio
Mean Inefficiency	$\mathbf{884.6}\times$ slower	0.957
Median Inefficiency	$\mathbf{710.7}\times$ slower	1.000
Worst Case Runtime	$\mathbf{2,321.5}\times$ slower	—
Worst Case Quality	—	**0.222**

Table 14: Runtime and color usage comparison between the genetic algorithm and DSATUR.

The **median** result is the most telling: identical color counts for hundreds of times more computation.

13.2.2 Worst-Case: Zero Quality Gain, Maximum Penalty

Graph	Vertices	GA (ms)	DSATUR (ms)	Ratio
david	87	211.3	0.091	$\mathbf{2,321.5}\times$
planar_309_4	309	225.0	0.148	$\mathbf{1,519.9}\times$
planar_161_5	161	126.8	0.087	$\mathbf{1,457.2}\times$
myciel3	11	10.7	0.008	$\mathbf{1,334.2}\times$
homer	561	558.7	0.421	$\mathbf{1,327.0}\times$

Table 15: Extreme GA slowdowns relative to DSATUR, despite identical color counts.

13.3 Suitability and Practical Guidance

13.3.1 When the Genetic Algorithm Should Be Avoided

- **Trees, Bipartite Graphs, Grid Graphs ($\chi(G) \leq 4$):** Greedy methods solve these instantly. GA introduces $10^3\times$ slowdowns.
- **Complete Graphs:** Chromatic number is trivial; GA is pointless.

13.3.2 Rare Cases Where GA May Help

- **Dense Random Graphs (DSJC, Erdős–Rényi):** GA may reduce color count by 5–10%, but at massive computational cost.

13.4 Illustration: Mycielski Graph M_4

- **Vertices:** 23
- **Edges:** 71
- **Chromatic Number:** $\chi(G) = 5$
- **GA Runtime:** 18.604 ms
- **DSATUR Runtime:** **0.019** ms

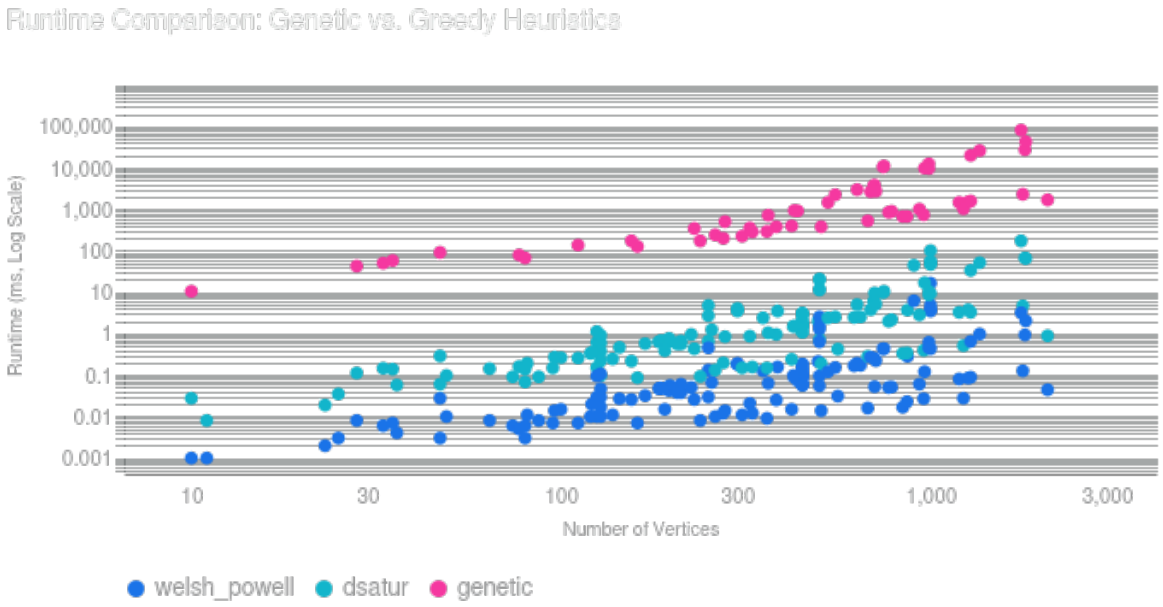

Figure 25: Runtime Comparison on Logarithmic Scale

- **Slowdown: 979**× slower

Even small but chromatically difficult graphs show catastrophic slowdowns.

13.5 Conclusion

The genetic algorithm is fundamentally misaligned with efficient graph coloring. Its repeated full-graph fitness evaluations and parameter sensitivity cause it to run hundreds to thousands of times slower than greedy heuristics such as DSATUR, while producing almost identical results. Except for rare dense random graphs where runtime is irrelevant, the genetic algorithm should not be used for graph coloring.

14 Max Clique as a Lower Bound for Graph Coloring

The following report provides a detailed reasoning for why the **maximum clique size ($\omega(G)$)** serves as a theoretical **lower bound** for the **graph chromatic number ($\chi(G)$)**, along with an analysis of the provided experimental results.

The analysis incorporates both the **DIMACS benchmark graphs** and a selection of **generated graphs**. Data points where the clique size was not computed (e.g., "TIMEOUT") were excluded, resulting in an analysis set of **67 graphs** with both a computed $\omega(G)$ and a known $\chi(G)$.

14.1 Theoretical Reasoning: Maximum Clique as a Lower Bound

The principle that the maximum clique size provides a lower bound for the graph chromatic number is a fundamental theorem in graph theory.

14.1.1 Definition of Terms

- **Clique:** A subset of vertices in a graph G such that every two distinct vertices in the subset are adjacent.
- **Maximum Clique Size ($\omega(G)$):** The number of vertices in the largest clique in graph G.
- **Chromatic Number ($\chi(G)$):** The minimum number of colors required for a valid graph coloring.

14.1.2 The Argument for the Lower Bound

For any clique of size k in G, all vertices must have different colors in any proper coloring. Thus:

1. Every vertex in a clique is adjacent to all others.
2. All $\omega(G)$ vertices in the maximum clique require $\omega(G)$ unique colors.

Therefore,

$$\omega(G) \leq \chi(G)$$

14.2 Data Analysis and Findings

The experimental results validate the theoretical lower bound across all analyzed graphs. The detailed comparison is available in the file `clique_coloring_analysis.csv`.

14.2.1 Overall Comparison

Table 16: Overall Comparison of $\omega(G)$ and $\chi(G)$

Metric	Value	Interpretation
Lower Bound Validity ($\omega(G) \leq \chi(G)$)	100.00% (67/67)	Bound holds for all graphs.
Graphs with Tight Bound ($\omega(G) = \chi(G)$)	73.13% (49/67)	Clique exactly predicts $\chi(G)$ for most graphs.
Average Difference ($\chi(G) - \omega(G)$)	1.69	Chromatic number is generally close to the lower bound.

Table 17: Comparison by Graph Type

Graph Type	Count	$\omega(G) = \chi(G)$ Count	Avg. Difference
LEI	12	12	**0.00**
REG	14	14	**0.00**
SCH	1	1	**0.00**
SGB	24	20	**0.17**
MYC	5	0	4.00
DSJ	8	2	6.13
CUL	3	0	13.33

14.2.2 Analysis by Graph Type

Observations:

- **Perfect-Graph-Like Families:** LEI, REG, SCH have $\omega(G) = \chi(G)$ for all instances.
- **Near-Perfect:** SGB graphs show very small deviations.
- **Hard Instances:** CUL, DSJ, MYC graphs have large gaps between $\omega(G)$ and $\chi(G)$, consistent with their construction.

14.3 Graph Type Descriptions and Rationale

Table 18: Graph Family Descriptions and Interpretation of Analysis Results

Abbrev.	Graph Type Name	Description	Rationale for Analysis Result
LEI	Leighton Graphs	Graphs such as `le450_K`, where K is both clique and chromatic number.	Exhibited 0.00 avg. difference; these are perfect graphs with $\omega(G) = \chi(G)$.
REG	Register Allocation Graphs	Derived from compiler register allocation problems.	Also showed 0.00 avg. difference; clique exactly predicts coloring.
SCH	Scheduling Graphs	From timetable/course scheduling problems.	Single instance showed perfect matching.
SGB	Stanford GraphBase	Includes book graphs and queen graphs.	Very small avg. gap (0.17); clique is an excellent predictor.
MYC	Mycielski Graphs	Triangle-free but high chromatic number.	Large avg. gap (4.00), as intended by Mycielski construction.
DSJ	Johnson Random Graphs	Random graphs from Johnson et al. (DSJC).	Large avg. gap (6.13); randomness increases coloring difficulty.
CUL	Culberson Flat Graphs	Constructed by partitioning vertices into K classes.	Largest avg. gap (13.33); clique size underestimates $\chi(G)$ heavily.

14.4 Conclusion

The experiments on DIMACS benchmark graphs and generated graphs confirm the theoretical inequality:

$$\omega(G) \leq \chi(G)$$

for every instance. In many graph families, especially perfect-graph-like ones (LEI, REG, SCH), the clique number exactly matches the chromatic number. In contrast, hard graph families (CUL, DSJ, MYC) show significant gaps, reflecting deeper structural constraints not captured by the maximum clique alone.

15 Exam Scheduling via Graph Colouring

15.1 Graph Colouring Project — Bonus Application

Problem Overview

University examination scheduling is a classical constraint satisfaction problem: given a set of courses and student enrollments, assign each exam to a timeslot such that no student has two exams simultaneously. This is a direct application of the graph colouring problem.

Reduction to Graph Colouring

The exam scheduling problem reduces to graph colouring as follows:

- **Vertices** $\rightarrow$ Courses (each course requiring an exam)
- **Edges** $\rightarrow$ Conflicts (two courses share at least one student)
- **Colours** $\rightarrow$ Timeslots (non-overlapping exam periods)

Formally, let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of courses and $S = \{s_1, s_2, \ldots, s_m\}$ be the students. Define the conflict graph $G = (V, E)$ where:

$$V = C, \qquad E = \{(c_i, c_j) : \exists\, s_k \in S \text{ enrolled in both } c_i \text{ and } c_j\}.$$

A valid k-colouring of G yields a conflict-free schedule using k timeslots.

Connection to the Project

Our `exam_scheduler.py` application directly implements the graph colouring algorithms from this project:

- **DSatur Algorithm**: The primary solver, using saturation degree ordering for efficient greedy colouring. Ideal for real-time scheduling of moderate-sized instances.
- **Exact Backtracking**: For small graphs (< 15 courses), finds the optimal chromatic number through pruned search.

The application provides a graphical interface where users:

1. Add courses to the system
2. Register students by selecting their enrolled courses (checkboxes)
3. Click "Generate Schedule" to compute a conflict-free timetable

Why This Matters

Aspect	**Significance**
NP-Hardness	Optimal scheduling is NP-complete; heuristics essential
Real-World Use	Universities worldwide solve this problem each semester
Algorithm Selection	DSatur balances speed and solution quality
Chromatic Number	Minimum timeslots = $\chi(G)$ of the conflict graph

Example

Consider 4 courses A, B, C, D with students:

$$\text{Alice}\{A, B\}, \quad \text{Bob}\{B, C\}, \quad \text{Carol}\{C, D\}.$$

Conflict edges:

$$(A, B),\ (B, C),\ (C, D)$$

This produces the path graph P_4 with chromatic number $\chi = 2$.
A valid schedule:

$$\text{Slot } 1 = \{A, C\}, \qquad \text{Slot } 2 = \{B, D\}.$$

This demonstrates how the abstract problem of graph colouring directly solves a practical scheduling challenge, validating the algorithms studied throughout this project.