Simulated Annealing for Graph Coloring

Team Member: Poluri Abhijit Suhas
December 2025

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.

1 Introduction

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** $\chi(G)$.

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.

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.

2 Algorithm Description

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.

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
```

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

3 Implementation Details

3.1 Language and Build

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

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.

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.

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.

4 Experimental Setup

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

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 1: DIMACS graph families used for evaluation.

Total graphs tested: 79 DIMACS instances

5 Results & Analysis

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.

5.2 General Performance

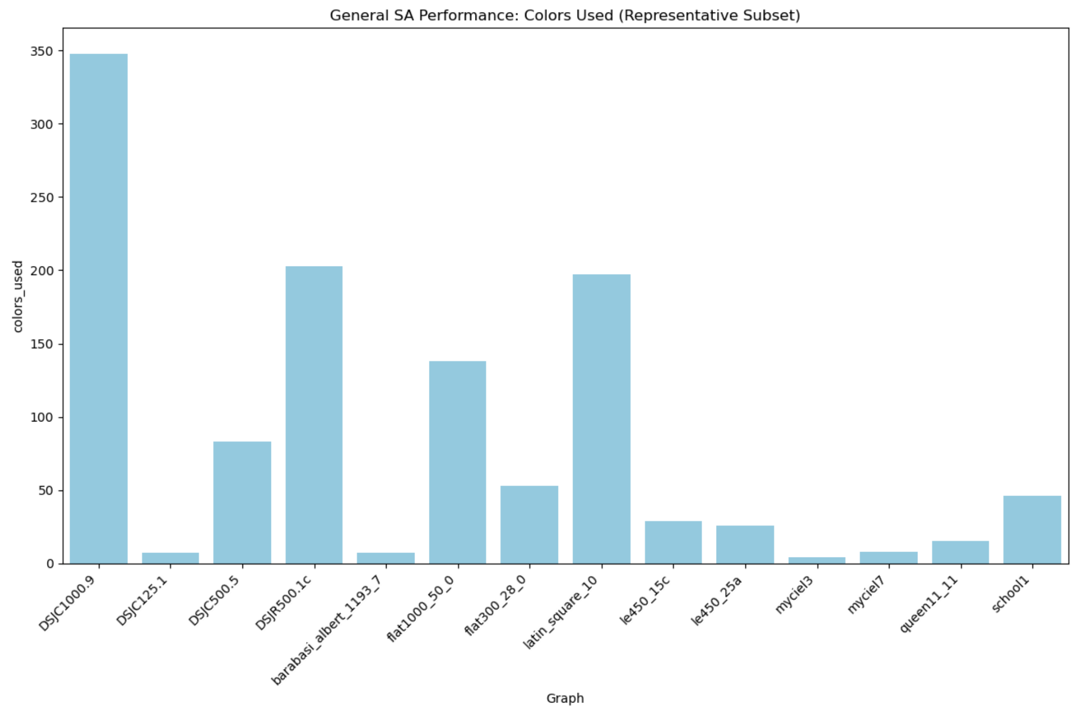

Figure 1: Colors used by SA on representative DIMACS graphs.

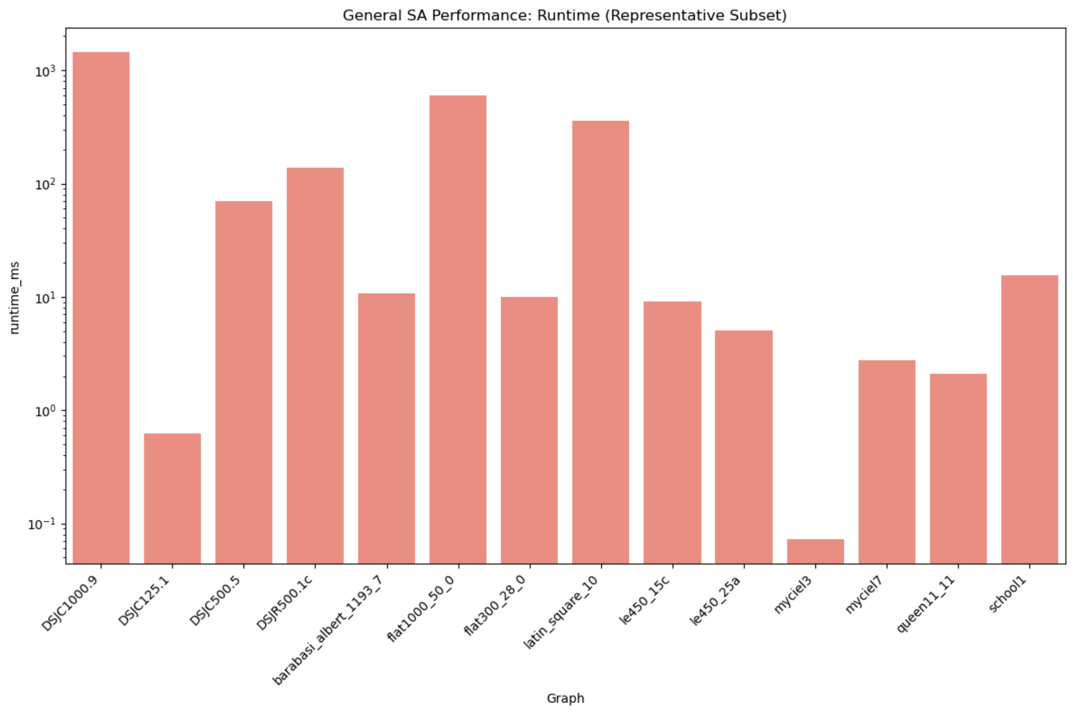

Figure 2: Runtime of SA on representative DIMACS graphs (log scale).

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 2: 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).

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.

6 Conclusion

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.

6.2 Limitations

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

6.3 Future Improvements

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

7 Bonus Disclosure

The following work is submitted for bonus evaluation:

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 3: SA parameter modes for bonus evaluation.

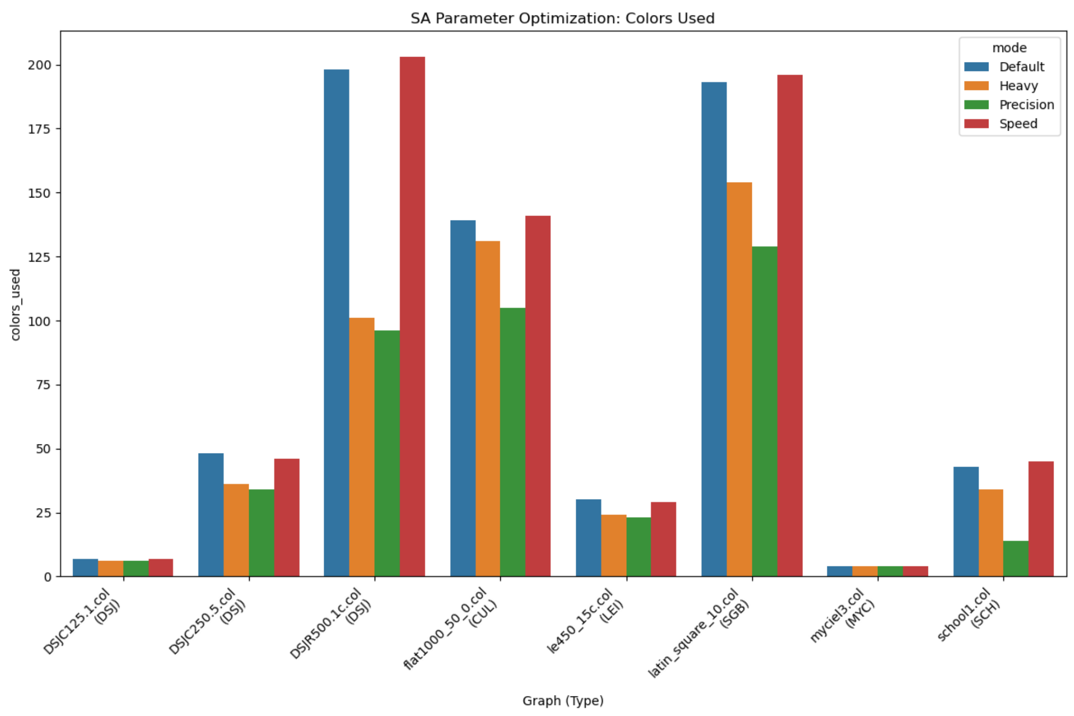

Figure 3: Colors used by different SA modes across graph families.

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.

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.

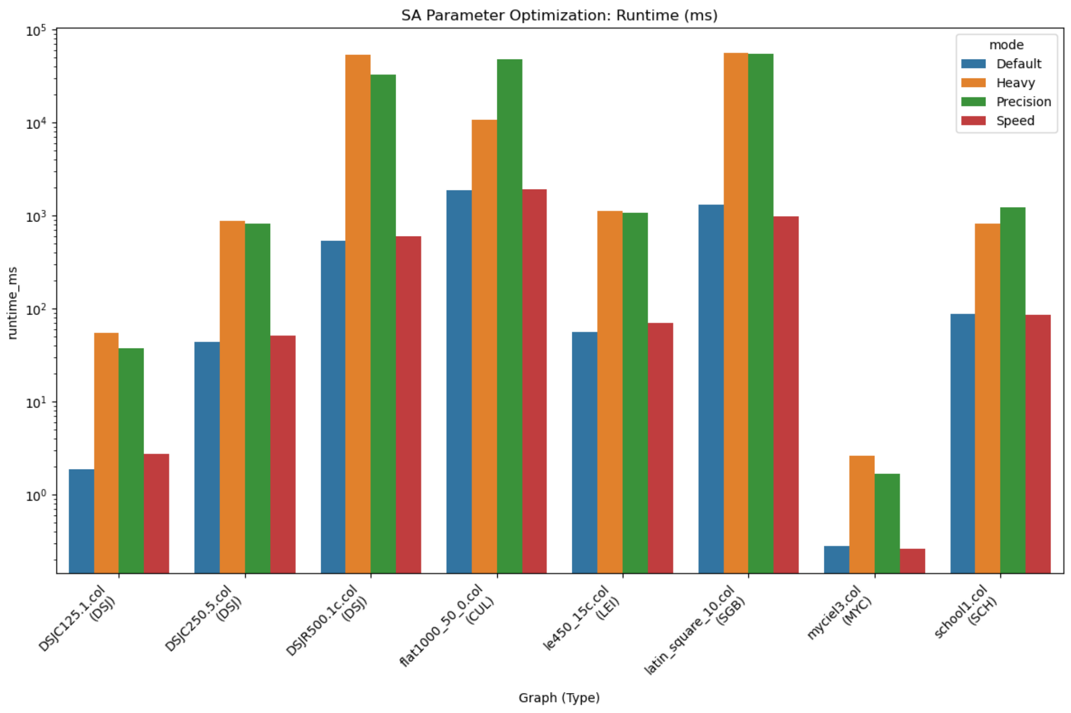

Figure 4: Runtime comparison of SA modes (log scale).

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`