Python Project Checkpoint - Graph Coloring

Songnan Lin, Zhanfei Shi, Yue Yu

November 2025

1 Introduction

The goal of our project is to study the graph coloring problem, a classical problem in combinatorial optimization. In graph coloring, the objective is to assign a color to each vertex of a graph so that no two adjacent vertices share the same color. The main numerical challenge is to find a coloring that uses as few colors as possible. Determining the minimum number of colors needed, known as the chromatic number, is an NP-hard problem, making it a rich setting for exploring exact, heuristic, and stochastic algorithms.

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

- **Greedy Coloring**, a simple heuristic that colors vertices sequentially based on available colors.
- **DSatur**, a more advanced heuristic that selects vertices in the order of highest saturation degree, producing strong results on many graph families.
- **Backtracking Search**, an exact method that explores the combinatorial search space with pruning, guaranteeing optimal solutions but with potentially exponential runtime.
- **Simulated Annealing**, a stochastic local-search method inspired by thermodynamic cooling, which attempts to reduce conflicts under a fixed number of colors and often finds near-optimal solutions efficiently.

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

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

2 Numerical Methods and Notation

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

2.1 Greedy Coloring

2.1.1 Algorithm Overview

The greedy algorithm colors vertices one by one in a given order. For each vertex, it picks the smallest color that does not create a conflict. This method is very fast but does not guarantee an optimal answer.

2.1.2 Numerical Formulation

If we process vertices in order $\{v_1, v_2, \dots\}$, then the color of v_i is

$$x_{v_i} = \min\{c : c \neq x_u \text{ for all } u \in N(v_i)\}.$$

Here $N(v_i)$ is the neighbor set of v_i.

2.1.3 Notation

- v_i: current vertex.
- $N(v)$: neighbors of v.
- x_{v_i}: assigned color.

2.1.4 What the Method Computes

- a valid coloring using possibly more colors than optimal,
- runtime,
- number of colors used.

2.1.5 Packages / Data Structures

- Python lists for colors,
- `Graph` adjacency lists,
- `time` module for timing.

2.2 DSATUR Coloring

2.2.1 Algorithm Overview

DSATUR is a well-known heuristic that chooses the next vertex based on "saturation", which is the number of different colors already used by its neighbors. At each step, it picks the vertex with the highest saturation degree. This usually produces much better results than simple greedy.

2.2.2 Numerical Formulation

For each vertex v, the saturation is

$$\text{sat}(v) = \#\{x_u : u \in N(v)\}.$$

The algorithm repeatedly:

1. picks v with largest saturation,
2. assigns the smallest available color.

2.2.3 Notation

- $\text{sat}(v)$: saturation degree,
- $N(v)$: neighbors of v,
- x_v: color assigned.

2.2.4 What the Method Computes

- a coloring (usually good quality),
- number of colors used,
- runtime.

2.2.5 Packages / Data Structures

- lists and sets for saturation tracking,
- `Graph` adjacency structure,
- Python built-ins for max-selection.

2.3 Backtracking Coloring

2.3.1 Algorithm Overview

Backtracking is an exact search method. It tries to assign colors to vertices one by one and checks whether the current partial coloring is still valid. If it reaches a point where no color is allowed, it "backs up" (backtracks) and tries a different color. This method always finds an optimal coloring but can be slow for large graphs.

2.3.2 Numerical Formulation

We keep a list of colors, one for each vertex. At step i, the algorithm tries to color vertex v_i with one of the available colors. The method checks whether using color c creates a conflict:

$$\text{safe}(v_i, c) = \begin{cases} \text{True}, & \text{if no neighbor of } v_i \text{ has color } c, \\ \text{False}, & \text{otherwise.} \end{cases}$$

If safe, we continue to the next vertex. If no color works, the algorithm undoes the last move and tries another choice.

2.3.3 Notation

- $G = (V, E)$: the graph.
- x_v: color of vertex v.
- v_i: the i-th vertex in the chosen order.
- `safe`: function that checks whether color c is allowed.

2.3.4 What the Method Computes

The method computes:

- the best coloring found (minimum number of colors),
- the number of nodes visited in the search,
- the total runtime.

2.3.5 Packages / Data Structures

We use:

- our custom `Graph` class for adjacency lists,
- Python lists for color storage,
- Python recursion for the search,
- `time` to measure runtime.

2.4 Simulated Annealing

2.4.1 Algorithm Overview

Simulated annealing (SA) is a randomized optimization method inspired by cooling in physics. It starts with a random coloring and makes small random changes. A change that reduces conflicts is always accepted. A change that increases conflicts may be accepted with some probability. The temperature slowly decreases so the algorithm becomes more strict over time.

2.4.2 Numerical Formulation

We define the conflict count

$$E(x) = \#\{(u, v) \in E : x_u = x_v\}.$$

When recoloring a vertex v, the change in conflicts is

$$\Delta = E(x_{\text{new}}) - E(x_{\text{old}}).$$

The move is accepted with probability

$$P = \begin{cases} 1, & \Delta \leq 0, \\ e^{-\Delta/T}, & \Delta > 0. \end{cases}$$

The temperature is updated by

$$T_{t+1} = \alpha T_t.$$

2.4.3 Notation

- x_v: color of vertex v,
- $E(x)$: number of conflicts,
- Δ: change in conflicts,
- T: temperature,
- α: cooling rate.

2.4.4 What the Method Computes

- a low-conflict coloring (possibly proper if lucky),
- the final number of conflicts,
- total runtime.

2.4.5 Packages / Data Structures

- `random` for sampling moves,
- `math` for the acceptance probability,
- `time` for timing,
- `Graph` class for adjacency,
- Python lists for color storage.

3 Package Structure

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

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

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

4 Tests

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

4.1 Greedy Coloring

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

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

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

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

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

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

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

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

4.2 DSATUR Coloring

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

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

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

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

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

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

We further test DSATUR on the triangle graph K_3, which has chromatic number 3. The test checks that the algorithm returns a valid proper coloring and that exactly three colors are used. In all cases, correctness is verified by ensuring that no two adjacent vertices share the same color.

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

4.3 Backtracking Search

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

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

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

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

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

- **Path on four vertices:** In `test_backtracking_solves_path_graph`, we construct a path graph on four vertices, which is known to be 2-colorable. The test asserts that the algorithm finds a proper coloring and that `num_colors` is exactly 2.
- **Triangle K_3:** In `test_backtracking_solves_triangle`, we use the complete graph on three vertices, whose chromatic number is 3. We check that the returned coloring is proper and that the algorithm uses exactly three colors.
- **Single vertex:** The test `test_backtracking_single_vertex` considers a graph with a single vertex and verifies that the algorithm returns a proper coloring with `num_colors` equal to 1.
- **Complete graph K_4:** Finally, in `test_backtracking_complete_graph_k4`, we construct the complete graph on four vertices. Since K_4 has chromatic number 4, we check that backtracking produces a proper coloring using exactly four distinct colors.

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

4.4 Simulated Annealing

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

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

5 Preliminary Investigations into the Effectiveness of Algorithms

5.1 Greedy Coloring

We first evaluate the greedy algorithm on random graphs of increasing size. For each value of n, we generate a new random graph $G(n, 0.3)$ and run the greedy coloring procedure once using the natural vertex order $(0, 1, 2, \ldots, n-1)$.

5.1.1 Results and Observations

Figure 1 shows the runtime and number of colors used. Several clear patterns emerge from the experiment:

- **Runtime grows steadily with n:** Even though greedy is very fast, the runtime increases roughly linearly as n increases from 5 to 40. All runtimes stay under 6×10^{-5} seconds.
- **Number of colors fluctuates:** Greedy does not always use a smooth or predictable number of colors. For example, the experiment shows a jump from 4 colors at $n = 20$ to 6 colors at $n = 25$ and a peak at 7 colors for $n = 30$ and $n = 40$.
- **Ordering effect**: Since we use only one vertex ordering (natural order), variations in color count reflect sensitivity to the graph structure rather than randomness in ordering.
- **Visualization**: Example greedy colorings for $n = 15$ and $n = 20$ are shown in Figures 2a and 2b. They illustrate how the greedy algorithm may produce valid colorings but sometimes uses more colors than necessary.

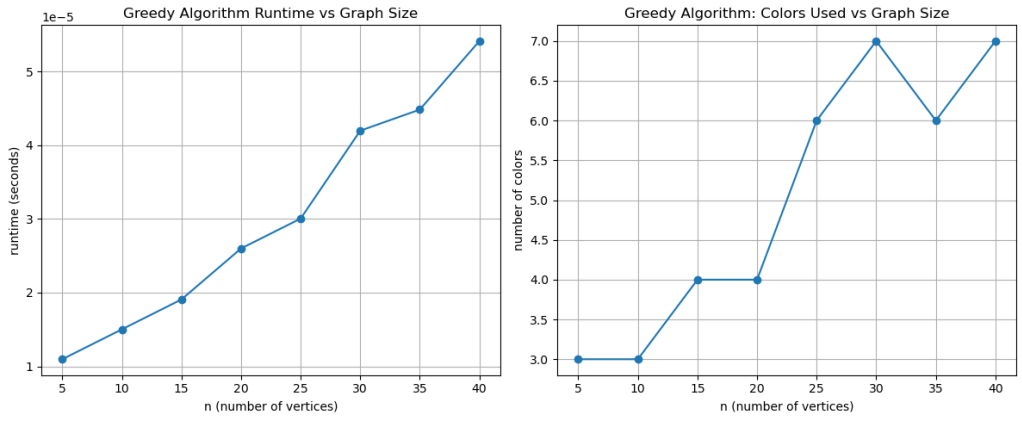
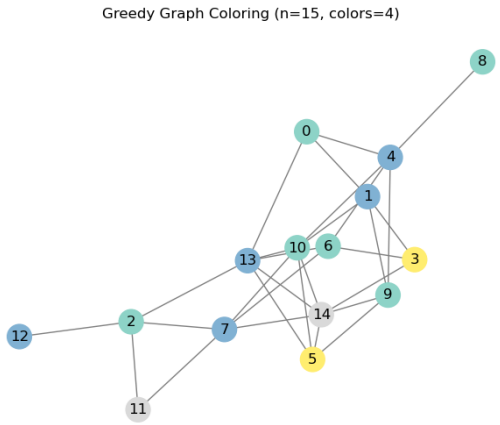
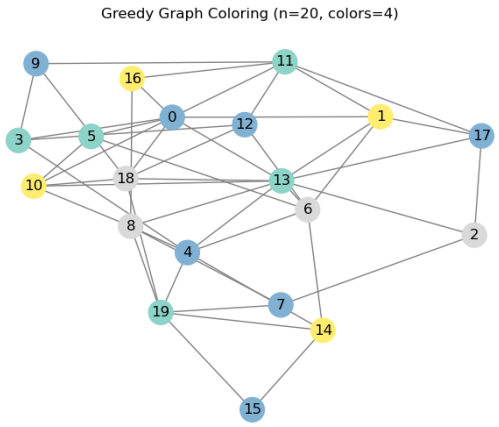


Figure 1: Greedy algorithm runtime and color usage as a function of graph size for $G(n, 0.3)$.



(a) Example greedy coloring for $n = 15$ on a random graph $G(15, 0.3)$ using 4 colors.



(b) Example greedy coloring for $n = 20$ on a random graph $G(20, 0.3)$ using 4 colors.

Figure 2: Greedy algorithm colorings for two random graphs of different sizes.

5.2 DSatur

Next, we test the DSATUR algorithm on the same sequence of graphs $G(n, 0.3)$. Because DSATUR chooses the next vertex based on saturation, it generally produces better colorings than simple greedy.

5.2.1 Results and Observations

Figure 3 summarizes runtime and color usage. The results match common expectations for DSATUR:

- **Better color quality:** DSATUR uses fewer colors overall compared to the greedy method. For example, greedy jumps to 7 colors for $n = 30$ and $n = 40$, while DSATUR only uses 5 or 6 colors.
- **More stable behavior:** Color counts in DSATUR increase gradually and smoothly (e.g., 2 colors at $n = 5$, 3 at $n = 10$, 4 at $n = 15$–25, and 5–6 at larger n).
- **Runtime grows faster than greedy but stays small:** DSATUR requires additional computation to track saturation, so runtimes are higher than greedy and grow from 2×10^{-5} seconds to approximately 2.4×10^{-4} seconds at $n = 40$. However, these times are still very fast.
- **Visualization:** Figures 4a and 4b show example colorings for $n = 15$ and $n = 20$, both using 4 colors. These examples illustrate how DSATUR assigns colors systematically based on saturation.

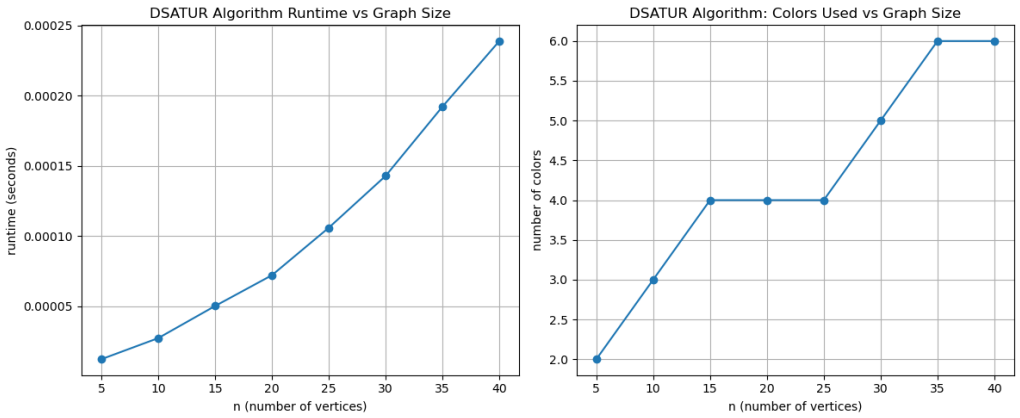
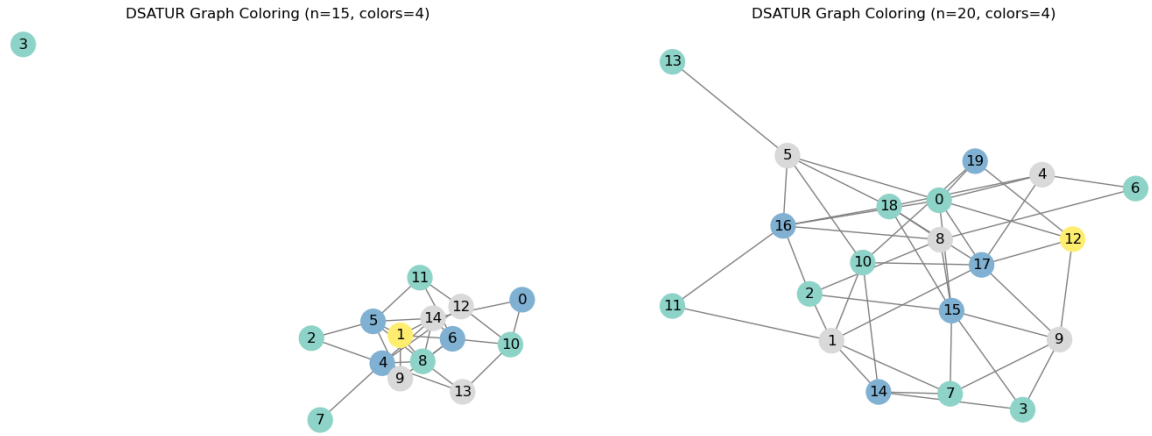


Figure 3: DSATUR runtime and color usage as a function of graph size for $G(n, 0.3)$.



(a) Example DSATUR coloring for $n = 15$ on $G(15, 0.3)$ using 4 colors.

(b) Example DSATUR coloring for $n = 20$ on $G(20, 0.3)$ using 4 colors.

Figure 4: DSATUR algorithm colorings for two random graphs of different sizes.

5.3 Backtracking Search

In this section, we present a detailed empirical evaluation of the backtracking graph coloring algorithm. Because backtracking is an exact exponential-time algorithm, our experiments focus on how its performance changes with graph size, graph density, and vertex-ordering strategies. All tests are conducted on random graphs $G(n, p)$.

For each run, we record:

- the chromatic number obtained (which is optimal),
- the number of search nodes (recursive DFS calls),
- the total runtime (in seconds).

5.3.1 Effect of Graph Size

We first vary the graph size n while fixing $p = 0.5$. Each result below represents an average over three random graphs.

n	Avg. Colors	Avg. Nodes	Avg. Runtime (s)
5	2.33	10	9.30e-06
10	4.33	23	1.47e-05
15	4.67	70	6.54e-05
20	5.67	92	9.50e-05
25	6.33	2029	2.88e-03
30	7.00	3681	6.03e-03

Table 1: Backtracking performance for increasing graph size ($p = 0.5$).

Analysis. As n increases, the chromatic number grows slowly, but the number of search nodes grows extremely rapidly:

- a modest increase from $n = 20$ to $n = 25$ results in search nodes jumping from 92 to 2029,
- at $n = 30$, the search explodes to more than 3,600 recursive calls,
- runtime increases from microseconds to several milliseconds.

This behavior illustrates the exponential nature of backtracking even on moderate-sized random graphs. This trend is consistent with the analysis of Bender and Wilf [3], who proved that the expected number of search nodes on a random graph behaves like $\exp(Cn^2/q)$.

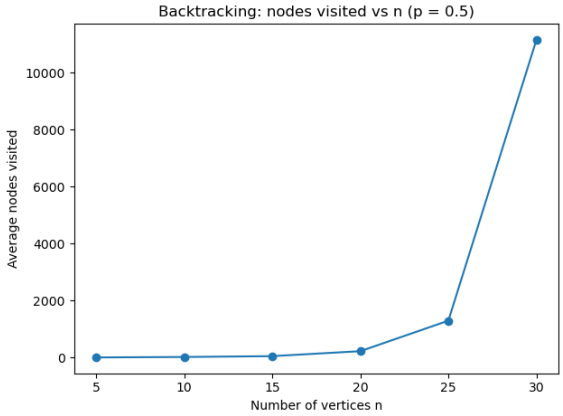


Figure 5: Nodes visited vs. n ($p = 0.5$).

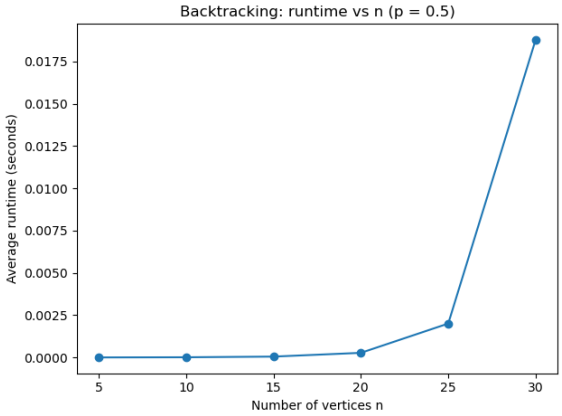


Figure 6: Runtime vs. n ($p = 0.5$).

5.3.2 Effect of Graph Density

Next, we fix the graph size at $n = 20$ and vary the edge probability p. The results are summarized as follows:

p	Avg. Colors	Avg. Nodes	Avg. Runtime (s)
0.1	2.4	60	1.13e-05
0.3	4.0	110	8.53e-05
0.5	5.8	234	2.83e-04
0.7	7.8	195	3.58e-04
0.9	12.0	78	2.97e-05

Table 2: Effect of density p on backtracking performance ($n = 20$).

Analysis. This experiment reveals a well-known pattern:

- very sparse graphs ($p = 0.1$) require few colors and little search,
- very dense graphs ($p = 0.9$) also become easier because almost all vertices are connected,
- **intermediate densities ($p = 0.5$–0.7) are the hardest**, requiring the most search nodes and longest runtime.

This matches classical combinatorial behavior: medium-density random graphs tend to have the most complex coloring structure.

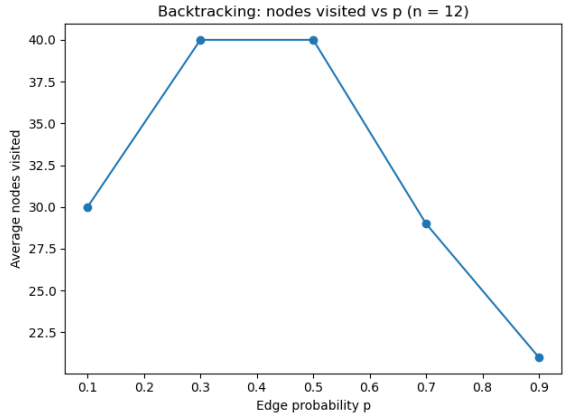


Figure 7: Nodes visited vs. density p ($n = 20$).

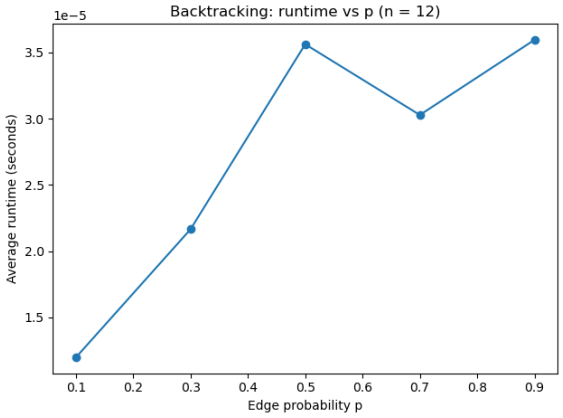


Figure 8: Runtime vs. density p ($n = 20$).

5.3.3 Effect of Vertex Ordering

Finally, we compare two vertex-ordering strategies:

- natural order (0, 1, 2, ...),
- degree-based ordering (highest degree first).

Using $G(20, 0.3)$, we observe:

Ordering	Avg. Colors	Avg. Nodes	Avg. Runtime (s)
Natural order	4.0	1174	8.66e-04
Degree order	4.0	94	7.57e-05

Table 3: Comparison of natural vs. degree ordering ($n = 20$, $p = 0.3$).

Analysis. Both strategies find the same optimal number of colors (as expected for an exact algorithm), but degree ordering reduces:

- search nodes by a factor of ≈ 12.5,
- runtime from 8.6×10^{-4} to 7.6×10^{-5}.

This demonstrates that the choice of vertex ordering is critical in pruning the search tree. As noted by Bender and Wilf [3], the number of nodes at depth L in the backtracking tree equals the number of proper colorings of the first L vertices, so improved vertex ordering reduces the branching factor early and greatly shrinks the search space.

5.3.4 Summary

Across all experiments, we observe:

- Backtracking handles small graphs efficiently but scales exponentially with n.
- Graph density strongly influences difficulty, with intermediate densities being the hardest.
- Vertex ordering dramatically improves performance; degree-based ordering reduces search dramatically.

These results confirm both the theoretical behavior of backtracking and the practical need for strong heuristics in exact graph coloring algorithms.

5.4 Simulated Annealing

In this section we present preliminary experiments for the simulated annealing (SA) algorithm implemented in `annealing.py`. Our goal is to understand how well SA performs as a graph-coloring heuristic and how its runtime scales with the graph size.

5.4.1 Experimental Setup

We generate random graphs with edge probability $p = 0.3$. For each value of n in

$$n \in \{10, 20, 40, 80, 160, 320, 640, 1280\},$$

we construct a new random graph and run the simulated annealing algorithm with a fixed number of colors $k = 4$. The SA hyperparameters are kept constant across all experiments: maximum number of iterations `max_iter` $= 20000$, initial temperature $T_0 = 1.0$, and cooling rate $\alpha = 0.999$.

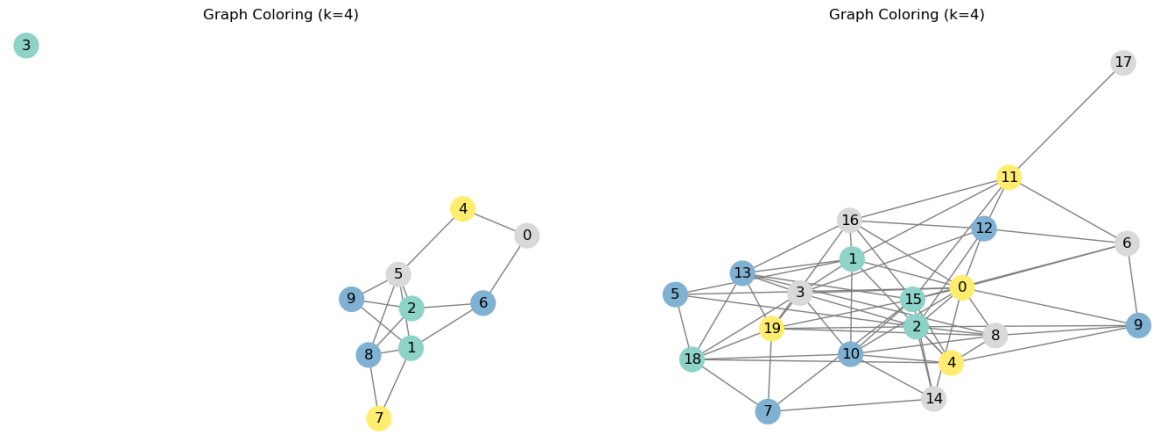
For each run we record:

- whether the algorithm finds a proper coloring (zero conflicts),
- the final coloring (when successful),
- the total runtime in seconds.

Whenever SA returns a valid coloring (no conflicts), we visualize the result using a NetworkX drawing where each vertex is colored according to the output coloring.

5.4.2 Graph Coloring Examples

Figures 9a and 9b show example colorings produced by simulated annealing on random graphs of size $n = 10$ and $n = 20$ respectively, both with $p = 0.3$ and $k = 4$ colors.



(a) Simulated annealing coloring of random graph with $n = 10$ and $p = 0.3$ using $k = 4$ colors.

(b) Simulated annealing coloring of random graph with $n = 20$ and $p = 0.3$ using $k = 4$ colors.

Figure 9: Simulated annealing colorings for two random graphs of different sizes.

In these two cases the algorithm is able to find a proper coloring, so every edge connects vertices of different colors. For larger graphs ($n \geq 40$) with the same parameter choices ($p = 0.3$, $k = 4$, fixed iteration budget), simulated annealing often fails to reach a conflict-free configuration. As a result, we do not show visualizations for those sizes: either the graph is not 4-colorable at this density or the algorithm becomes trapped in local minima within the allowed number of iterations. This behavior already illustrates a key limitation of SA in this setting: with a small fixed number of colors and relatively dense graphs, the success probability drops quickly as n grows.

5.4.3 Runtime Scaling

Figure 10 shows the runtime of simulated annealing as a function of the number of vertices n for the same set of graphs. Each point corresponds to a single run of SA on a fresh random graph. Even when SA fails to find a proper coloring, the runtime is still well defined, since the algorithm runs for (almost) the full iteration budget. The runtime grows smoothly with n, reflecting the cost of evaluating conflicts at each iteration. For the chosen parameters, the growth is roughly linear in this range of n.

5.4.4 Discussion of Effectiveness

These preliminary experiments suggest that simulated annealing is effective at finding proper colorings for small random graphs (e.g., $n = 10$ and $n = 20$) with a modest number of colors ($k = 4$). The visual examples confirm that, when SA succeeds, it produces valid colorings that respect the graph structure. However, as the graph size increases while keeping $p = 0.3$ and $k = 4$ fixed, the problem becomes much harder: the algorithm frequently fails to eliminate all conflicts within the allotted iteration budget. This indicates that, for larger or denser graphs, simulated annealing in its current form may require

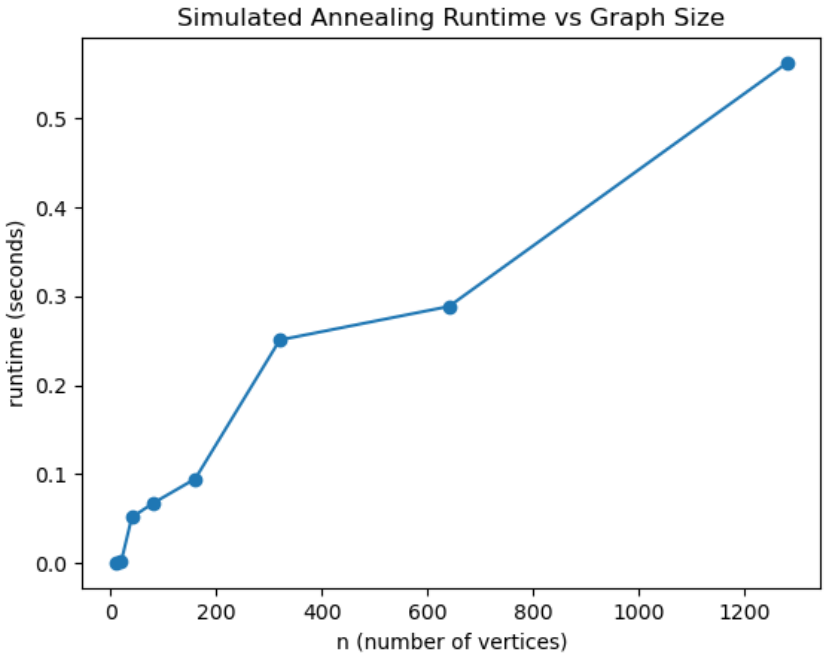


Figure 10: Simulated annealing runtime as a function of graph size n with $p = 0.3$ and $k = 4$ colors.

more colors, a slower cooling schedule, or more iterations to remain competitive. This observation is consistent with previous studies on SA for graph coloring, which report that while simulated annealing can efficiently find good colorings for small to medium graphs, its success probability decreases as the graph size or density increases [5]. Chams, Hertz, and de Werra also noted that combining SA with other heuristics, such as the RLF method, can improve performance on larger graphs by using SA for the final crucial steps of the coloring process [5].

From a performance standpoint, the runtime results show that simulated annealing scales in a predictable way with the number of vertices, and remains computationally feasible for the sizes considered in this experiment. Overall, SA appears to be a useful heuristic for small to medium graphs or more generous color budgets, but it is less reliable as a stand-alone method for large, relatively dense graphs when the number of colors is tightly constrained.

6 Proposed Variations and Extensions

6.1 Greedy Coloring and DSatur

Based on the research by San Segundo [2] and other recent developments in graph coloring algorithms, we propose the following extensions for the final project.

6.1.1 Variation 1: PASS-enhanced DSATUR Algorithm

1. Research Questions:
    - How does the PASS tiebreaking strategy improve upon standard DSatur?
    - What is the optimal threshold for activating PASS based on available colors?

- How does PASS compare in computation time and solution quality?

2. Implementation Approach: We will modify the DSatur algorithm to use PASS tiebreaking only when vertices have limited color choices (2-3 available colors). This selective approach balances computational cost and solution quality.

3. Data Sources:

   - DIMACS graph coloring benchmark
   - Random graphs with varying sizes and densities

6.1.2 Variation 2: Hybrid Greedy-DSatur Approach

1. Research Questions:

   - Can we combine the speed of greedy algorithms with the quality of DSatur?
   - What is the best switching point between greedy and DSatur?
   - How does hybrid performance compare to individual algorithms?

2. Implementation Approach: We will develop a hybrid algorithm that starts with fast greedy coloring and switches to DSatur when the coloring becomes difficult. The switching point will be determined by the number of conflicts or available colors.

6.2 Backtracking Search

Based on our exact backtracking solver and its role as a ground-truth method for small graphs, we propose the following extension for the final project.

6.2.1 Variation: Warm-Started, Heuristic-Guided Backtracking

1. Research Questions.

- How much can a good initial upper bound on the chromatic number (obtained from a heuristic such as greedy, DSatur, or simulated annealing) reduce the search tree explored by backtracking?
- Does dynamically updating the vertex ordering (e.g., using DSatur-style saturation information during search) significantly improve pruning compared to a fixed static order?
- For which classes of graphs (random graphs, structured benchmarks) does warm-starting backtracking from heuristic colorings give the largest benefit in terms of runtime and nodes visited?

2. Implementation Approach. We will augment the existing backtracking implementation with two main ideas:

- **Warm-start upper bound.** Before running backtracking, we apply a fast heuristic (such as DSatur or simulated annealing with a generous iteration budget) to obtain a valid coloring with k_{heur} colors. We then initialize `best_num_colors` to k_{heur} instead of n. This immediately tight upper bound should allow the pruning rule `used_colors` $\geq$ `best_num_colors` to cut off many branches.

- **Heuristic-guided vertex ordering.** Rather than using a fixed static ordering by degree, we will experiment with dynamic orderings inspired by DSatur: at each step, we choose the uncolored vertex with the highest saturation degree (and break ties by degree). This is expected to create conflicts early and thus strengthen pruning.

We will implement these extensions as options on top of the existing `backtracking_coloring` function, so that we can directly compare vanilla backtracking with its heuristic-guided variants.

3. Evaluation Plan and Data Sources. Results from engineering optimization further demonstrate that combining backtracking with domain-specific heuristics yields substantial practical speedups without compromising optimality [4]. We adopt this idea in our heuristic-guided graph-coloring variants. To evaluate the impact of warm-starting and heuristic guidance, we will measure:

- the number of search nodes visited,
- the total runtime,
- whether the optimal chromatic number is still found.

We plan to test on:

- random graphs with varying n and edge probability p,
- a subset of DIMACS graph-coloring benchmark instances.

By comparing plain backtracking with the warm-started, heuristic-guided version on these datasets, we hope to quantify how much practical speedup can be obtained without sacrificing optimality.

6.3 Simulated Annealing

Simulated annealing (SA) is a stochastic optimization method that has been successfully applied to graph coloring due to its ability to escape local minima through controlled randomization. As Alons [6] notes, SA differs from traditional local optimization by probabilistically accepting uphill moves, which allows it to escape local optimum regions and explore the global solution space. Building on the classical work and subsequent improvements, we propose the following extensions for simulated annealing within the context of our final project.

6.3.1 Variation: Adaptive-Temperature Simulated Annealing for Graph Coloring

1. Research Questions.

- How does an adaptive temperature schedule compare with fixed cooling schedules (e.g., geometric cooling) in terms of solution quality and convergence speed?
- Can dynamically adjusting the neighborhood selection (e.g., selecting conflicts first) significantly reduce the number of iterations needed?

- What is the trade-off between runtime and coloring quality under different annealing intensities (number of iterations, reheating strategies)?
- On which classes of graphs (sparse, dense, or structured) does simulated annealing provide the greatest improvement over greedy algorithms?

2. Implementation Approach. We plan to implement an enhanced simulated annealing solver with the following modifications over the standard formulation:

- **Conflict-directed neighborhood moves.** Rather than choosing any vertex uniformly, we select a conflicting vertex with higher probability, making each step more meaningful.
- **Adaptive temperature schedule.** The temperature T will be updated based on progress:
$$T_{t+1} = \begin{cases} \alpha T_t, & \text{if no improvements occur,} \\ \beta T_t, & \text{if an improvement was found,} \end{cases}$$
where $\alpha < 1$ and $\beta > 1$ are cooling and reheating parameters.
- **Reheating strategy.** If the search stagnates for a pre-specified number of iterations, the algorithm performs a mild reheating step to escape deep local minima.
- **Hybrid initialization.** We will experiment with initializing SA using greedy or DSatur colorings instead of random initial colorings to reduce the search space.

3. Evaluation Plan and Data Sources. To assess the performance of the enhanced simulated annealing algorithm, we will measure:

- the best coloring obtained (number of colors used),
- convergence speed (iterations until stability),
- average runtime per instance,
- variability across multiple stochastic runs.

We will evaluate the method on the following datasets:

- **Erdős–Rényi random graphs** with various n and density levels p,
- **DIMACS benchmark instances**, especially medium-sized graphs where heuristics can outperform greedy but exact algorithms become infeasible.

Our goal is to determine whether adaptive-temperature simulated annealing offers a significant improvement over standard stochastic methods and whether it can serve as a strong baseline for heuristic warm-starting of the backtracking algorithm in Section 6.2.

References

[1] Aditi Dudeja, Rashmika Goswami, and Michael Saks. *Randomized Greedy Online Edge Coloring Succeeds for Dense and Randomly-Ordered Graphs*. arXiv:2406.13000 [cs.DS], 2024. doi:10.48550/arXiv.2406.13000.

[2] P. San Segundo. *A new DSATUR-based algorithm for exact vertex coloring*. Computers & Operations Research, 39(7):1724–1733, 2012.

[3] E. A. Bender and H. S. Wilf. *A Theoretical Analysis of Backtracking in the Graph Coloring Problem*. Journal of Algorithms, 6(2):275–282. Academic Press, 1985.

[4] M. A. Abido. *An application of backtracking search algorithm in designing power system stabilizers for large multi-machine system*. International Journal of Electrical Power & Energy Systems, 34(1):136–143, 2012.

[5] M. Chams, A. Hertz, and D. de Werra, *Some experiments with simulated annealing for coloring graphs*, European Journal of Operational Research, vol. 32, pp. 260–266, 1987.

[6] K. Alons, *Simulated Annealing on the Composite Graph Coloring Problem*, Opportunities for Undergraduate Research Experience Program (OURE), Missouri University of Science and Technology, 1991.