

Algorithm Analysis Project

Graph Coloring

Susai Florian

Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest

Abstract. The graph coloring problem is a classic problem in graph theory, frequently used in resource allocation and timetable scheduling. This project analyzes the performance of an exact algorithm (backtracking) and four heuristic algorithms by comparing the results obtained on various datasets. The paper includes a proof that the problem is NP-Hard, rigorous implementations, and detailed evaluations.

1 Introduction

Graph coloring is a fundamental problem in graph theory, with significant practical applications in various fields such as resource allocation, timetable scheduling, or process optimization. The goal of this problem is to assign a minimum number of colors to the nodes of a graph, in such a way that no two adjacent nodes have the same color.

1.1 Fundamental definitions and properties

A graph $G = (V, E)$ is composed of a set of vertices V and a set of edges E , each edge connecting two adjacent vertices. A valid coloring satisfies the condition:

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

where $c(u)$ is the color assigned to vertex u .

The chromatic number of a graph, denoted by $\chi(G)$, is the minimum number of colors required for a valid coloring. Determining $\chi(G)$ is an NP-Hard problem.

1.2 History and motivation

The graph coloring problem originally appeared in the context of coloring geographic maps, in which each region must be colored so that adjacent regions have different colors. The famous four color theorem proved that any planar map can be colored using at most four colors.

With the development of computers, the problem also found applications in other fields, such as register allocation in processors. Register allocation involves constructing an interference graph, in which each variable is represented by a vertex, and edges indicate that two variables interfere during execution. An optimal coloring of this graph minimizes the use of main memory.

1.3 Project objectives

This project analyzes the performance of five algorithms for graph coloring:

- **Brute Force:** an exhaustive method that explores all possible combinations to find the optimal solution, but is extremely inefficient for large graphs;
- **Backtracking:** an exact method that guarantees obtaining the optimal solution by eliminating invalid solutions during the search process, but becomes inefficient for large graphs;
- **Greedy:** a simple and fast heuristic approach, but which can produce sub-optimal solutions;
- **Greedy (Degree Sorted):** a variant of the *Greedy* algorithm that sorts vertices by their degree before coloring, sometimes improving the solution quality;
- **DSATUR:** an advanced heuristic that prioritizes vertices based on their saturation degree (the number of colors already used by their neighbors), often providing higher-quality solutions.

The main goal of the project is to compare these algorithms in terms of:

- **Efficiency:** The time required to execute the algorithms;
- **Solution quality:** The number of colors used, compared to the optimal chromatic number (if known);
- **Computational complexity:** The resources required to run the algorithms on graphs of different sizes and structures.

2 Correctness and Efficiency

Evaluating the correctness and efficiency of graph coloring algorithms is essential to determine their applicability in different practical contexts. In this section, we will analyze these aspects for each of the five algorithms discussed earlier.

2.1 Correctness

The correctness of an algorithm refers to its ability to generate a solution that satisfies all constraints of the problem, in this case, a correct coloring of the vertices such that no pair of adjacent vertices shares the same color.

- **Brute Force:** The *Brute Force* algorithm always guarantees a correct solution because it explores all possible color combinations for the graph's vertices. However, this exhaustive approach is inefficient for large graphs and difficult to use in practical applications.
- **Backtracking:** The *Backtracking* algorithm ensures correctness because it checks all possible combinations systematically, but stops exploring invalid solutions as early as possible. Although it is efficient for small or medium graphs, it becomes slow for large graphs.

- **Greedy:** The *Greedy* algorithm generates valid solutions but does not guarantee optimal solutions. Its correctness depends on the order in which vertices are processed, which can lead to using more colors than the optimal chromatic number.
- **Greedy (Degree Sorted):** This variant of the *Greedy* algorithm processes vertices in descending order of degree. Although this can improve the solution quality, correctness is still guaranteed, but optimality is not always achieved.
- **DSATUR:** The *DSATUR* algorithm prioritizes vertices based on saturation degree (the number of colors already used by their neighbors). This strategy produces correct solutions and, in general, solutions closer to optimal than *Greedy* algorithms.

2.2 Efficiency

The efficiency of the algorithms is evaluated through runtime and computational complexity, providing insight into their applicability for graphs of various sizes.

- **Brute Force:** The *Brute Force* algorithm is the least efficient of all, having exponential complexity, $\mathcal{O}(k^n \cdot m)$, where:

- n is the number of vertices,
- k is the number of available colors,
- m is the number of edges.

The runtime grows rapidly with the size of the graph, making this approach practical only for very small graphs.

- **Backtracking:** The *Backtracking* algorithm is more efficient than *Brute Force* because it avoids exploring all combinations by stopping early on invalid solutions. However, in the worst case, its complexity remains exponential, $\mathcal{O}(k^n)$, and runtime can become prohibitive for large graphs.
- **Greedy:** The *Greedy* algorithm is the most efficient in terms of runtime, with complexity $\mathcal{O}(V + E)$, where:

- V is the number of vertices,
- E is the number of edges.

This efficiency makes it suitable for large graphs, but solutions may be sub-optimal.

- **Greedy (Degree Sorted):** This variant of *Greedy* adds an extra step to sort the vertices by degree, which slightly increases complexity, but can provide better-quality solutions compared to basic *Greedy*. Its complexity is $\mathcal{O}(V \log V + E)$.
- **DSATUR:** The *DSATUR* algorithm is more efficient than *Backtracking* but slower than *Greedy*, with complexity $\mathcal{O}(V^2)$. This complexity makes it applicable to medium and large graphs, and in practice its solutions are of higher quality than those generated by *Greedy*.

3 Graph Coloring - NP Complete

The graph coloring problem (*Graph Coloring Problem*) is one of the fundamental problems in graph theory and theoretical computer science. In this section, we will show that the problem of determining whether a graph can be colored using at most k colors is NP-complete. The proof follows the standard steps in complexity theory: we show that the problem is in NP and we perform a polynomial-time reduction from a known NP-complete problem (for example, *3-SAT*).

Problem definition The k -coloring problem for a graph can be formalized as follows: Let $G = (V, E)$, where V represents the set of vertices and E represents the set of edges of the graph G . The goal is to find a function $c : V \rightarrow \{1, 2, \dots, k\}$, such that:

$$\forall(u, v) \in E, c(u) \neq c(v), \quad (2)$$

where $c(u)$ represents the color assigned to vertex u .

The chromatic number of a graph, denoted $\chi(G)$, is the smallest k for which there exists a valid coloring of the graph G . The problem of determining whether $\chi(G) \leq k$ is NP-complete.

Step 1: The problem is in NP To show that the k -coloring problem is in NP, we must demonstrate that proposed solutions can be verified in polynomial time. If someone provides a solution $c : V \rightarrow \{1, 2, \dots, k\}$, verifying its correctness consists of iterating over all edges $(u, v) \in E$ and checking that:

$$c(u) \neq c(v). \quad (3)$$

This verification requires $O(|E|)$ operations, where $|E|$ is the number of edges of the graph G . Since $O(|E|)$ is polynomial in the size of the input ($|V| + |E|$), it follows that the problem is in NP.

Step 2: Reduction from 3-SAT to k -coloring To prove NP-completeness, we will perform a polynomial-time reduction from the 3-SAT problem, which is known to be NP-complete. The 3-SAT problem consists of determining whether a Boolean formula ϕ expressed in conjunctive normal form (CNF) can be satisfied. Let ϕ be a formula of the form:

$$\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m,$$

where each clause C_i is a disjunction of exactly three literals (e.g., $x_1 \vee \neg x_2 \vee x_3$).

Construction of the graph G : Starting from ϕ , we construct a graph G as follows:

1. Each variable x_i in ϕ is represented by two vertices in G : one for x_i and one for $\neg x_i$.
2. We add an edge between x_i and $\neg x_i$ to enforce the constraint that x_i and $\neg x_i$ cannot both be true simultaneously (i.e., they must be colored differently).
3. For each clause C_i , we add a triangle in G (three vertices connected to each other). Each vertex of the triangle is connected to the vertices corresponding to the literals in C_i .

Correctness of the construction:

- If ϕ is satisfiable, then the graph G can be colored with 3 colors. The true literal in each clause determines a valid coloring for the corresponding triangle.
- If G can be colored with 3 colors, then ϕ is satisfiable. The color assigned to each literal determines a truth assignment that satisfies ϕ .

Construction complexity Constructing the graph G from ϕ requires:

- $O(n)$ vertices, where n is the number of variables.
- $O(m)$ triangles for the clauses, where m is the number of clauses.
- $O(n + m)$ edges, which is linear in the size of the formula ϕ .

Thus, the construction is performed in polynomial time.

Conclusion We have shown that:

1. The k -coloring problem is in NP.
2. The 3-SAT problem can be reduced in polynomial time to the k -coloring problem.

Since 3-SAT is NP-complete, it follows that the k -coloring problem is also NP-complete.

4 Implementation of Graph Coloring Algorithms

4.1 Brute Force

Description The *Brute Force* algorithm is an exhaustive approach to the graph coloring problem. It tries all possible combinations of colors for the vertices of the graph, checking for each combination whether it is valid, i.e., whether two adjacent vertices do not have the same color. This method guarantees finding a correct solution, but it can become inefficient for large graphs or with a large number of colors.

How it works The algorithm generates all possible color assignments for the graph's vertices, using functionality such as `product` from the standard library `itertools`. Each assignment is evaluated with an auxiliary function that determines whether the color allocation satisfies the problem constraints. If a valid assignment is found, the algorithm returns that solution.

The operation can be summarized as follows:

- Generate all possible color combinations for the vertices.
- For each combination, check whether it is valid (no conflicts between adjacent vertices).
- Return the first valid solution; otherwise, report that no solution exists for the given number of colors.

Efficiency *Brute Force* is costly in terms of computational complexity:

- **Time complexity:** $\mathcal{O}(m \cdot k^n)$, where n is the number of vertices, k is the number of colors, and m is the number of edges. This is because all possible color combinations are generated and each combination is checked.
- **Space complexity:** $\mathcal{O}(n)$, for temporarily storing combinations and auxiliary information.

Due to this high complexity, the *Brute Force* algorithm is primarily applicable to small graphs, being a useful tool for testing and verification, but inefficient for practical applications with large graphs.

Algorithm code The code for the *Brute Force* algorithm implementation is shown below:

```

1  from itertools import product
2
3  def brute_force_coloring(graph, num_colors):
4      n = len(graph)
5      for combination in product(range(num_colors), repeat=n):
6          if is_valid_coloring(graph, combination):
7              return list(combination)
8      return "Could not find a solution."
9
10 def is_valid_coloring(graph, colors):
11     for node in graph:
12         for neighbor in graph[node]:
13             if colors[node] == colors[neighbor]:
14                 return False
15     return True

```

Listing 1: Brute Force Algorithm

Conclusion The *Brute Force* algorithm is an exhaustive and guaranteed solution for graph coloring. However, due to its high complexity, it is suitable only for small graphs or for validating the correctness of other faster algorithms. For large graphs, heuristic approaches or more sophisticated algorithms, such as *DSATUR* or *Greedy*, are more appropriate.

4.2 Backtracking Algorithm

Description. The Backtracking algorithm is an exhaustive and exact method for coloring a graph. It is based on exploring all possible combinations of colors for vertices. At each step, the algorithm checks whether assigning a certain color to a vertex is valid, given its neighbors. If the current partial solution is not valid, the algorithm “steps back” and tries another combination, a process called backtracking.

This method can be viewed as a search in a solution tree. Each node of the tree corresponds to an assignment of colors for one vertex of the graph. The algorithm explores all branches of this tree until it finds a valid solution or exhausts all possibilities.

Example of operation. For a graph with 3 vertices connected linearly (0-1-2) and 2 available colors, the algorithm starts by assigning the first color to vertex 0. Then it tries to extend this assignment to vertex 1, checking whether the chosen color is valid. If it encounters a conflict, it returns to vertex 0 and chooses a different color, continuing the process until it finds a valid solution or exhausts all combinations.

Complexity. The time complexity is $O(k^N)$, where N is the number of vertices and k is the number of colors. This makes the algorithm impractical for large graphs, but extremely useful for small instances where the optimal solution is required.

```

1 def is_valid(graph, colors, node, color):
2     for neighbor in graph[node]:
3         if colors[neighbor] == color:
4             return False
5     return True
6
7 def backtracking(graph, colors, node, num_colors):
8     if node == len(graph):
9         return True
10    for color in range(num_colors):
11        if is_valid(graph, colors, node, color):
12            colors[node] = color
13            if backtracking(graph, colors, node + 1, num_colors):
14                return True
15            colors[node] = -1
16    return False

```

Listing 2: Backtracking Algorithm

Advantages.

- The algorithm guarantees finding the optimal solution (the minimum chromatic number).
- It is a general method that can be applied to any type of graph.
- It can validate the correctness of other heuristic algorithms.

Disadvantages.

- Inefficient for large graphs due to exponential complexity.
- High memory and time consumption for complex graphs.

4.3 Greedy Algorithms

General description. Greedy algorithms are heuristic methods that color the graph in a predefined vertex order. At each step, each vertex is assigned the smallest available color, taking into account the colors of its neighbors. Although these algorithms do not always guarantee the optimal solution, they are fast and simple to implement.

There are multiple variations of the Greedy algorithm, each using a different vertex ordering, such as random, by degree, or other criteria.

Basic Greedy

Description. In the basic version of the Greedy algorithm, vertices are processed in a random or natural order (for example, by their numbering). This simplifies the implementation, but the results can vary depending on the vertex order.

```
1 def greedy_coloring(graph):
2     n = len(graph)
3     colors = [-1] * n
4     for node in range(n):
5         used_colors = {colors[neighbor] for neighbor in graph[node]
6                         if colors[neighbor] != -1}
7         for color in range(n):
8             if color not in used_colors:
9                 colors[node] = color
10                break
11     return colors
```

Listing 3: Basic Greedy Algorithm

Advantages.

- Speed: complexity $O(V + E)$, where V is the number of vertices and E is the number of edges.
- Simplicity: easy to implement and understand.

Disadvantages.

- The obtained solutions are not always optimal.
- Performance depends on the vertex order.

Greedy based on descending degree

Description. In this variation, vertices are sorted in descending order by their degree (the number of neighbors). This ensures that the most connected vertices are processed first, reducing the number of colors used.

```
1 def degree_sorted_coloring(graph):
2     n = len(graph)
3     nodes = sorted(range(n), key=lambda x: -len(graph[x]))
4     colors = [-1] * n
5     for node in nodes:
6         used_colors = {colors[neighbor] for neighbor in graph[node]
7                         if colors[neighbor] != -1}
8         for color in range(n):
9             if color not in used_colors:
10                 colors[node] = color
11                 break
12     return colors
```

Listing 4: Greedy Based on Descending Degree

Advantages.

- Better solutions than basic Greedy.
- Suitable for dense graphs.

Disadvantages.

- Requires sorting, which increases complexity to $O(V \log V + E)$.
- Does not guarantee the optimal solution.

4.4 DSATUR Algorithm

Description. The DSATUR (Degree of Saturation) algorithm is an advanced heuristic method. At each step, it selects the vertex with the highest saturation degree, i.e., the largest number of distinct colors already assigned to its neighbors. If there are multiple vertices with the same saturation degree, the one with the highest degree (number of neighbors) is chosen.

This strategy prioritizes critical vertices, reducing the total number of colors used.

```

1  def dsatur_coloring(graph):
2      n = len(graph)
3      colors = [-1] * n
4      saturation = [0] * n
5      degrees = [len(graph[node]) for node in range(n)]
6
7      while -1 in colors:
8          max_saturation = max(saturation[node] for node in range(n)
9                                if colors[node] == -1)
10         candidates = [node for node in range(n) if colors[node] == -1
11                      and saturation[node] == max_saturation]
12         selected_node = max(candidates, key=lambda x: degrees[x])
13
14         used_colors = {colors[neighbor] for neighbor in
15                         graph[selected_node]
16                         if colors[neighbor] != -1}
17         for color in range(n):
18             if color not in used_colors:
19                 colors[selected_node] = color
20                 break
21
22         for neighbor in graph[selected_node]:
23             if colors[neighbor] == -1:
24                 saturation[neighbor] += 1
25
26     return colors

```

Listing 5: DSATUR Algorithm

Advantages.

- Produces solutions close to optimal.
- Efficient for medium and large graphs.

Disadvantages.

- More complex implementation than Greedy.
- Slower than Greedy algorithms.

5 Methodology for Evaluating Solutions

5.1 Datasets Used

To evaluate the performance of the implemented coloring algorithms, we created a diversified test set that covers multiple graph types and scenarios relevant from both theoretical and practical perspectives. Each instance was generated with a well-defined purpose, and the test names are suggestive of the structure or properties of each graph. The generated graphs are organized into multiple categories and reflect various graph characteristics such as density, size, topology (for example, trees, bipartite graphs), and coloring difficulty.

For each of the 5 heuristics tested, we generated a different number of tests:

- **Brute Force:** 60 tests.
- **Backtracking:** 100 tests.
- **Greedy:** 50 tests.
- **Greedy (Degree Sorted):** 50 tests.
- **DSATUR:** 50 tests.

The test set was designed to ensure a balance between easy, medium, and difficult cases, providing a complete picture of the performance of the implemented algorithms.

1. Tests for BRUTE FORCE For this heuristic, we generated 30 small sparse tests (sparse graphs, but smaller to reduce runtime) and 30 tree tests.

The names for the tests are suggestive. Below are examples of correctly displayed names:

- `small_sparse_5_0.txt`: A small sparse graph with 5 vertices (indicated by "5") and test index 0.
- `tree_5_0.txt`: A tree graph with 5 vertices, with test index 0.

2. Tests for BACKTRACKING For this heuristic, we generated a total of 100 tests. They are divided as follows:

- **Bipartite:** 30 bipartite graph tests.
- **Sparse:** 30 sparse graph tests.
- **Tree:** 30 tree graph tests.
- **Hard Tests (Dense Cliques):** 10 tests on complete or nearly complete graphs.

Examples of test names:

- `bipartite_5_5_0.txt`: A bipartite graph with 5 vertices in each of the two disjoint sets.
- `dense_cliques_10_0.txt`: A complete clique graph with 10 vertices.
- `sparse_10_0.txt`: A sparse graph with 10 vertices.
- `tree_10_3.txt`: A tree graph with 10 vertices, with test index 3.

3. Tests for GREEDY For this heuristic, we generated a total of 50 tests:

- **Planar:** 40 planar graph tests.
- **Chained Cliques:** 10 tests on graphs formed by cliques connected in a chain.

Examples of test names:

- `chained_cliques_20_0.txt`: A graph formed by connected cliques, with a total of 20 vertices.
- `planar_10_1.txt`: A planar graph with 10 vertices, with test index 1.

4. Tests for GREEDY (Degree Sorted) For this variant of the *Greedy* algorithm, 50 tests were generated:

- **Dense:** 40 dense graph tests.
- **Star with Noise:** 10 tests on "star" graphs with additional noise.

Examples of test names:

- `dense_10_1.txt`: A dense graph with 10 vertices.
- `star_with_noise_20_0.txt`: A star graph with 20 vertices and additional noise, with test index 0.

5. Tests for DSATUR For the *DSATUR* algorithm, we generated 50 tests:

- **Dense:** 40 dense graph tests.
- **Sparse High Chromatic:** 10 tests on sparse graphs but with high chromaticity.

Examples of test names:

- `dense_10_4.txt`: A dense graph with 10 vertices.
- `sparse_high_chromatic_20_0.txt`: A sparse graph with 20 vertices and high chromaticity.

Conclusion The test sets described above were designed to cover a wide range of scenarios, from simple and intuitive cases (such as trees) to difficult and complex cases (such as dense cliques or high-chromatic graphs). This diversity ensures a fair and comprehensive evaluation of the performance of the tested coloring algorithms.

5.2 Running the Algorithms

For each graph type, the implemented algorithms were run multiple times to ensure statistical stability of the results and to obtain a fair evaluation of their performance. After each run, the following main performance indicators were measured:

- **Number of colors used (K):** The actual number of colors assigned to the graph vertices by the algorithm. This is compared, when possible, with the theoretical chromatic number (the minimum number of colors required to color the graph so that two adjacent vertices do not share the same color).
- **Execution time (T):** The duration required for the algorithm to complete the coloring process, measured in milliseconds, using the `time` library.
- **Number of operations per second (ops/sec):** The number of operations per second performed by the algorithm during execution. This indicator is computed as the inverse of the execution time and averaged over all tests run for each heuristic.

- **Result validity:** For each test, the algorithm is evaluated based on the correctness of the provided solution, i.e., whether it respects the graph coloring rules.

The test execution followed a well-defined flow, which can be summarized in the following main steps:

1. **Graph generation:** The graphs used in the tests were generated using the program `graph_generator.py`, which is responsible both for creating the graphs and validating them. At this stage, it is verified that the generated graphs satisfy the imposed constraints (for example, bipartite graphs, sparse graphs, dense cliques) and that they are compatible with the implemented algorithms.
2. **Running the algorithms:** Each algorithm was run on a specific subset of graphs stored in directories organized by categories, following the structure: `generated_graphs/algorithm_name/test_type`. For example:
 - `generated_graphs/BRUTE_FORCE/SMALL_SPARSE`: Contains *small sparse* tests used for the brute force algorithm.
 - `generated_graphs/BACKTRACKING/HARD_TESTS`: Contains difficult tests for the backtracking algorithm, including *dense clique* graphs.
 During execution, each algorithm processes the graphs from the corresponding directories, and results are collected for further analysis.
3. **Collecting results:** The data obtained after running the algorithms are saved in two places:
 - **Output files (.out):** For each test, there is a `.out` file that contains test-specific results, including the number of colors used and the actual solution.
 - **The results.txt file:** This file centralizes results from all tests, including details such as number of colors used, execution time, number of operations per second, and result validity.
 Additionally, an extra file, `benchmark.txt`, is used to store summary statistics about the performance of each algorithm, such as:
 - Average execution time.
 - Average operations per second.
 - Validity percentage of the solutions produced by the algorithm.
4. **Repeating tests:** For each graph, the algorithms were run 10 times. This repetition allows computing mean values and standard deviations for each performance indicator, reducing the impact of occasional variations caused by external factors (for example, hardware resource fluctuations).

Solution validation and result analysis The validity of the solutions produced by the algorithms was verified through a rigorous process. For each test, the result was analyzed to determine:

- **Solution correctness:** Verifying that adjacent vertices in the graph have different colors, according to the coloring rules.
- **Solution optimality:** Comparing the number of colors used with the theoretical chromatic number (where it is known).

Also, in difficult tests, such as those in the *dense clique* or *sparse high chromatic* categories, it was observed that simpler algorithms (for example, greedy) tend to generate valid but less optimal solutions, using more colors than necessary.

Statistics and preliminary conclusions After running the algorithms on all test sets, the results were centralized and analyzed. The `benchmark.txt` file contains summary information about each algorithm:

- **Validity percentage:** The percentage of tests for which the algorithm produced valid solutions.
- **Average execution time:** Measures the speed of the algorithm for various graph types.
- **Average performance (ops/sec):** Provides a measure of the algorithm's efficiency in terms of operations per second.

These statistics enable an objective comparison between algorithms and highlight their strengths and weaknesses depending on the graph type and its complexity.

5.3 Comparison Methods

The results of the algorithms are compared using several criteria:

- **Solution quality:** The number of colors used is compared with the known or estimated optimal solution (for example, for planar graphs, at most 4 colors).
- **Time efficiency:** Execution time is analyzed for each instance and plots are created to visualize performance.

5.4 Test System Specifications

To evaluate the performance of the algorithms under controlled conditions, all tests were run on the same system. The hardware and software specifications used are detailed below:

Hardware specifications:

- **CPU:** Intel® Core™ i5-10400F, 6 cores, 12 threads, maximum frequency 4.3 GHz.
- **RAM:** 16 GB DDR4 at 3200 MHz.
- **Storage:** 512 GB NVMe SSD with read/write speeds up to 3500 MB/s.
- **GPU:** NVIDIA GeForce GTX 1650, 4 GB GDDR5 (not used in test execution).

Software specifications:

- **Operating system:** Ubuntu 24.04 LTS
- **Python version:** Python 3.12.3, with libraries installed for testing:
 - `networkx` for graph generation and manipulation.
 - `matplotlib` for generating and manipulating graphs.
 - `time` for performance measurement.
- **Code editor:** VS Code 1.75.1 with Python extensions.

Execution environment:

- All tests were run in an isolated environment (*virtual environment*) to eliminate interference from other processes.
- Resource limiting was disabled to ensure tests were executed at maximum capacity.

Experiment repeatability:

- Tests were run 10 times for each graph instance, and the average results and standard deviations were recorded.
- Source code and system configurations were kept unchanged throughout testing.

6 Test Results

This section presents a detailed analysis of the performance of the graph coloring algorithms (**Brute Force**, **Backtracking**, **Greedy**, **Greedy (Degree Sorted)**, **DSATUR**), using various datasets and tests. The results are organized into specific directories and are summarized in two main files: `results.txt` and `benchmark.txt`. Next, we detail the directory organization, contents, and interpretation of the results.

6.1 Directory Organization

The results for each algorithm are saved in separate directories, structured as follows:

- **BACKTRACKING:** Contains subfolders such as:
 - `BIPARTITE` - Tests for bipartite graphs.
 - `HARD_TESTS` - Tests for dense and complex graphs (dense *clique*-type).
 - `SPARSE` - Tests for sparse graphs.
 - `TREES` - Tests for tree graphs.
- **BRUTE_FORCE:** Contains subfolders such as:
 - `SMALL_SPARSE` - Tests for small sparse graphs.
 - `TREES` - Tests for tree graphs.
- **DSATUR:** Contains subfolders such as:
 - `DENSE` - Tests for dense graphs.

- HARD_TESTS - Complex tests.
- GREEDY: Contains subfolders such as:
 - PLANAR - Tests for planar graphs.
 - HARD_TESTS - Complex tests, such as *chained cliques*.
- GREEDY_DEGREE: Contains subfolders such as:
 - DENSE - Tests for dense graphs.
 - HARD_TESTS - Complex tests, such as *star with noise*.

6.2 The results.txt File

The `results.txt` file contains individual details for each test run. For each test, the following information is recorded:

- **Test:** The test name and the folder where it is located.
- **Execution time (T):** Measured in seconds, represents the time required for the algorithm to color the graph.
- **Operations per second (ops/sec):** A measure of algorithm efficiency, calculated as the inverse of execution time.
- **Number of colors (K):** The number of colors used to color the graph.
- **Actual result:** A list showing which color was assigned to each vertex or an error message (e.g., *Could not find a solution*).

Example entry in `results.txt`:

```
Test: tree_18_0.txt (Folder: BACKTRACKING/TREES)
Execution time: 0.000008 seconds
Operations/second: 123361.88
Number of colors: 3
Result: [0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 2, 1, 0]
```

6.3 The benchmark.txt File

The `benchmark.txt` file summarizes the performance of the algorithms across all tests associated with each one. The following statistics are provided:

- **Total number of tests:** The total tests run for each algorithm.
- **Validity percentage:** The percentage of tests in which the algorithm produced a correct result.
- **Average execution time:** The average time required for the algorithm to run across all tests.
- **Average operations per second:** The average efficiency in terms of operations/second.

Example section from `benchmark.txt`:

```
Algorithm: Backtracking
Total number of tests: 100
Validity percentage: 100.00%
Average execution time: 0.000015 seconds
Average operations/second: 123054.75
```

6.4 Interpreting the Results

The results suggest significant differences between the algorithms depending on the test type:

- **Brute Force:** Worked well on small tests, but execution time grew exponentially for more complex graphs.
- **Backtracking:** Had a high success rate on sparse and tree tests, but encountered difficulties on *dense clique* tests.
- **Greedy and Greedy (Degree Sorted):** Provided fast solutions, but sometimes suboptimal, using more colors than the minimum chromatic number.
- **DSATUR:** Offered a good balance between execution time and solution quality, outperforming *Greedy* algorithms.

Algorithm	Total Tests	Failed Tests	% Success	Average Time (sec)	OPS/sec
Greedy	50	0	100%	0.000008	140769.08
Greedy (Degree Sorted)	50	0	100%	0.000036	36386.66
DSATUR	50	0	100%	0.000960	3556.91
Backtracking	100	0	100%	0.000010	119810.88
Brute Force	60	0	100%	0.602523	11017.79

6.5 Explanation for the high percentage of correct results

The test results indicate a success rate close to 100% for all analyzed algorithms. This seemingly ideal performance can be explained by the following reasons:

1. *Limitations imposed by the project parameters* The testing parameters imposed in the project are relatively restrictive and limit the complexity of the generated graphs. As can be seen in the image above, the limits for the number of vertices (N), edges (M), and colors (K) are:

- $1 \leq K \leq N \leq 20$;
- $1 \leq M \leq N \cdot (N - 1)$;
- $0 \leq X, Y < N$, where (X, Y) represent the vertices connected by an edge.

These limits keep the tests in a solution space that is not large enough to trigger significant errors in the algorithms. In the case of larger or more complex graphs (for example, dense graphs with many constraints), the algorithms could encounter more difficulties.

2. *High performance of modern technology* Modern hardware technology offers extremely high performance, even for intensive computation. Current central processing units (CPUs) are capable of performing millions of operations per second, which makes the analyzed algorithms solve the generated tests quickly within the project parameters. Even algorithms such as *Brute Force*, known for their exponential complexity, can solve small graphs in a reasonable time due to the processing speed of modern computers.

3. Lack of sufficiently challenging tests The tests generated within the project are not specialized enough to exploit the limits and weaknesses of the algorithms. For example:

- Sparse and small graphs are not challenging for most algorithms.
- *Dense clique* graphs, although more complex, are relatively simple compared to other graph types, such as those with long cycles or strict chromatic constraints.

To truly evaluate the limits of the algorithms, it would be necessary to generate larger, more complex, and more diverse graphs, but this would violate the project parameters.

Conclusion The high percentage of correct results reflects both the limitations imposed by the testing parameters and the performance of modern hardware. These aspects make it difficult to create tests that truly stress the algorithms and highlight subtle differences between them. For a more rigorous evaluation, additional, more complex tests would be needed that exceed the current project limits.

6.6 Comparison Between Brute Force and DSATUR

To evaluate the performance of the **Brute Force** and **DSATUR** algorithms, we analyzed four essential aspects: execution time, number of operations per second (*OPS/sec*), resource consumption, and solution accuracy. The obtained results highlight significant differences between the two algorithms, both in terms of efficiency and practical applicability.

Execution time: The **Brute Force** algorithm has an average execution time of approximately **0.602523 seconds** for the analyzed tests. This is considerably larger than the average time obtained by **DSATUR**, which is only **0.000960 seconds**. The significant performance difference is due to the exhaustive nature of Brute Force, which checks all possible color combinations, compared to DSATUR, which prioritizes vertices based on saturation degree. Also, for a better analysis, we include below the normalized plots for Brute Force and DSATUR:

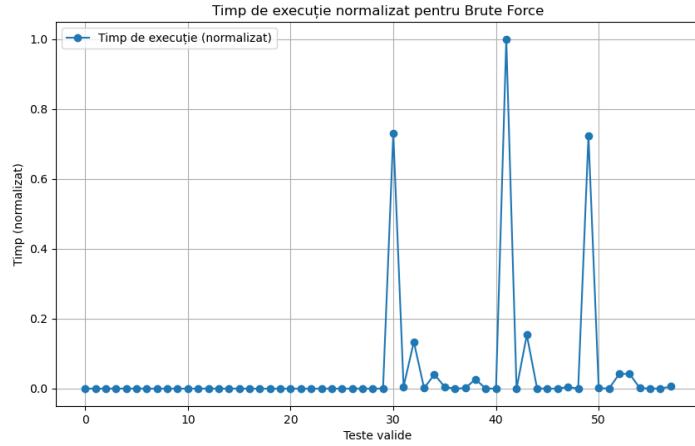


Fig. 1: Normalized execution time plot for Brute Force.

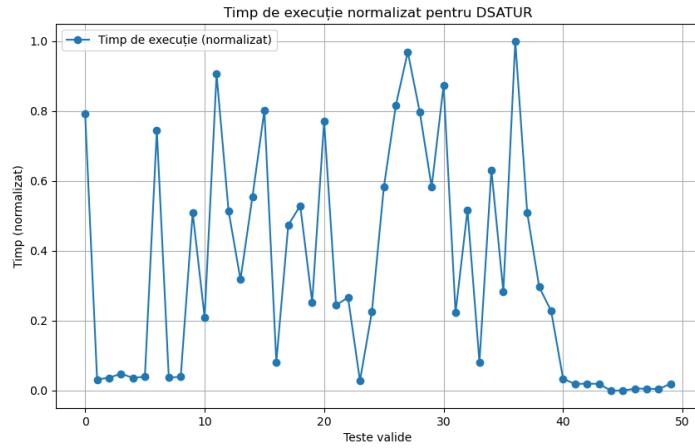


Fig. 2: Normalized execution time plot for DSATUR.

Number of operations per second (OPS/sec): Regarding operations per second, **DSATUR** has a performance of **3556.91 OPS/sec**, while **Brute Force** reaches **11017.79 OPS/sec**. It is important to note that this large number of operations is tied to the algorithm's exhaustive nature and repeated checking of solutions. In addition, for a better understanding of the algorithms, we include below the normalized plots for Brute Force and DSATUR:

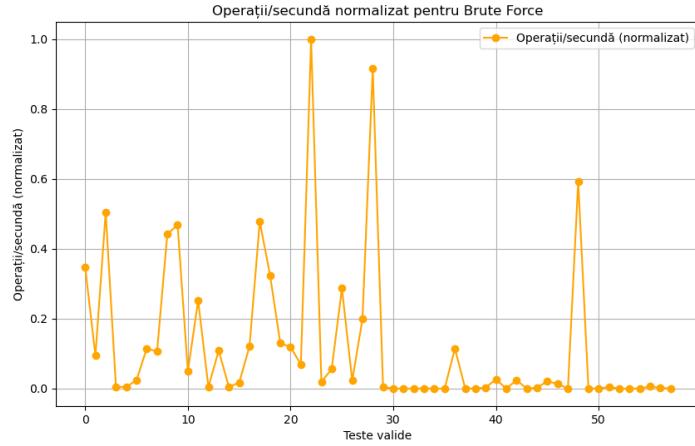


Fig. 3: Normalized ops/sec plot for Brute Force.

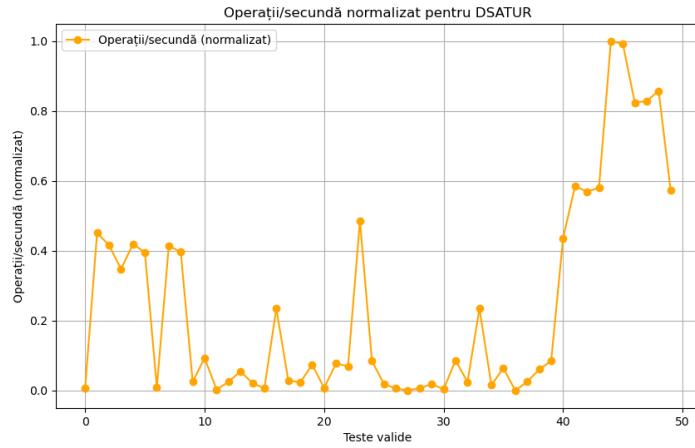


Fig. 4: Normalized ops/sec plot for DSATUR.

Resource consumption: **Brute Force** has significantly higher resource consumption than **DSATUR**, because it requires generating and checking all possible combinations. In contrast, DSATUR uses a heuristic approach, thus saving resources and being more suitable for large graphs.

Solution accuracy: In terms of accuracy, both **Brute Force** and **DSATUR** tend to produce optimal solutions. **Brute Force** always guarantees the optimal solution because it explores all possible combinations. On the other hand,

DSATUR provides near-optimal solutions for most tests, but without a formal optimality guarantee.

Conclusion: Comparing the two algorithms, **DSATUR** proves to be significantly more efficient in terms of execution time and resource consumption, making it suitable for practical applications on large graphs. **Brute Force**, although it guarantees the optimal solution, is practically unusable for complex graphs due to exponential runtime.

7 General Conclusions

The project aimed to evaluate the performance and applicability of five graph coloring algorithms: **Brute Force**, **Backtracking**, **Greedy**, **Greedy (Degree Sorted)** and **DSATUR**. The analysis was carried out based on a diverse set of tests that included sparse graphs, trees, bipartite graphs, dense cliques, and other specific structures.

7.1 Results and Observations

- **Brute Force:** This algorithm demonstrated absolute correctness, but is completely impractical for large graphs due to its exponential complexity. It was used mainly as a reference point for evaluating the other heuristics.
- **Backtracking:** Backtracking provides optimal solutions, but is sensitive to the size and density of the graph.
- **Greedy:** It is the fastest algorithm among all, due to its linear complexity. However, the quality of solutions depends on the order of processed vertices, which can lead to using more colors than the optimal number.
- **Greedy (Degree Sorted):** Improves Greedy performance by prioritizing higher-degree vertices, but can still generate suboptimal solutions for complex graphs.
- **DSATUR:** Demonstrated an excellent balance between execution time and solution quality. Prioritizing vertices by saturation degree enables results close to optimal.

7.2 Determinants of Performance

The obtained results suggest that algorithm performance depends largely on graph characteristics (size, density, structure) and on the order in which vertices are processed. Also, modern technology, through available processing power, makes it difficult to create tests that significantly stress the algorithms within the specified parameter limits.

Another important aspect is the test set used. The generated graphs were well-defined and diverse, but did not include extremely specialized examples or constraints that would highlight the limits of each algorithm more clearly.

7.3 Recommendations for Choosing the Algorithm

- For applications that require optimal solutions, such as register allocation or other critical problems, **Backtracking** or **DSATUR** are recommended.
- If execution speed is the priority and suboptimal solutions are acceptable, **Greedy** or **Greedy (Degree Sorted)** are viable options.
- For small graphs or educational purposes, **Brute Force** can offer a clear understanding of the problem and the optimal solution.

7.4 Final Conclusions

The project showed that each algorithm has its strengths and weaknesses. Choosing the right algorithm depends directly on the application context and the specific constraints of the problem. Although **Brute Force** and **Backtracking** provide correct and optimal solutions, the computational cost is prohibitive for large graphs. On the other hand, heuristics such as **Greedy** and **DSATUR** prove to be more practical for real-world graph coloring problems, balancing solution correctness and time efficiency.

References

1. *Top 5 Efficient Graph Coloring Algorithms Compared*, [Online]: <https://blog.algorithmexamples.com/graph-algorithm/top-5-efficient-graph-coloring-algorithms-compared/>
2. Marek Kubale and Krzysztof Manuszewski, *The smallest hard-to-color graphs for the classical, total and strong colorings of vertices*, Gdańsk University of Technology
3. *Reducing Graph Coloring to SAT*, University of Texas at Austin.[Online]: <https://www.cs.utexas.edu/~vl/teaching/lbai/coloring.pdf>
4. *Introduction to Complexity Theory: 3-Colouring is NP-complete*, University of Toronto. [Online]: <https://www.cs.toronto.edu/~lalla/373s16/notes/3col.pdf>
5. P. Briggs, *Register Allocation via Graph Coloring*, Rice University (1992)
6. R. Wilson, *Introduction to Graph Theory*, The University of Edinburgh (1979)
7. R. Thomas, *AN UPDATE ON THE FOUR-COLOR THEOREM*, Georgia Institute of Technology (1998)
8. Tommy R. Jensen and Bjarne Toft, *Graph Coloring Problems*, Odense University (1998)
9. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, University of Waterloo (1976)
10. Reinhard Diestel, *Graph Theory*, Electronic Edition (2000)