Universidade Federal do ABC
Centro de Matemática, Computação e Cognição
Programa de Graduação em Ciência da Computação

An Experimental Study of Graph Coloring Algorithms and a Visualization Web Tool

Rafael Calpena Rodrigues

Santo André - SP, Maio de 2023

Rafael Calpena Rodrigues

An Experimental Study of Graph Coloring Algorithms and a Visualization Web Tool

Relatório apresentado ao Programa de Graduação em Ciência da Computação, como parte dos requisitos necessários para a conclusão no Bacharelado em Ciência da Computação.

Universidade Federal do ABC – UFABC

Centro de Matemática, Computação e Cognição

Programa de Graduação em Ciência da Computação

Orientador: Fabrício Olivetti de França

Coorientadora: Carla Negri Lintzmayer

Santo André - SP

Maio de 2023

Abstract

Graph coloring is an NP-hard problem in the combinatorial optimization category with several real-world applications. Here, we create dockerized algorithms and cloud benchmarking tools to evaluate constructive heuristics, exact algorithms and constraint propagation on a subset of graphs from the DIMACS challenges dataset. We also create a web visualizer targeted at smaller graphs to replay graph coloring actions in a step-by-step manner, including incrementally displaying the backtracking tree. Our cloud test results show that the Pass implementation solved the most number of graphs in one hour, while exact DSATUR was often the fastest algorithm. Our proposed custom GAC algorithm only solved more graphs than the arbitrary ordering algorithm. The presence of DSATUR for both heuristics and exact algorithms resulted, in general, in better approximations for the chromatic number of a graph.

Keywords: graph coloring, greedy, dsatur, sewell, pass, web visualizer, backtracking, ac-3, constraint satisfaction problem, constraint propagation, arc consistency, cloud, test, docker.

Contents

1 Introduction

A graph is a mathematical structure consisting of a set of vertices and a set of edges that represent relationships between the vertices. When modeling a problem, vertices can be considered any entities for which pairwise relationships are important. Some practical uses of graphs require that these entities are grouped in a way such that no two vertices which are adjacent share the same label, while using the minimum amount of labels. This problem is known as the graph coloring problem.

It is believed that the study of graph coloring started when Francis Guthrie conjectured that a map could always be colored with four colors avoiding clashes between adjacent regions. In this conjecture, the adjacency relationship would require at least one boundary, and would not apply to a single point in common between two regions [1]. Francis asked his brother Frederick Guthrie, a mathematics undegraduate student at Cambridge University at the time, about the existence of such theoretical upper bound on the number of colors for map colorings. Frederick then asked his teacher Augustus de Morgan – the creator of De Morgan's law – who also did not have an answer to the conjecture [2]. The problem went unsolved for a while, until an upper bound of five colors was proven indirectly by Alfred Kempe in 1879 in an incorrect attempt to prove the four color theorem (later disproven by Percy John Heawood in 1890) [3]. In 1989, Kenneth Appel and Wolfgang Haken finally proved, with the aid of computers, that every planar graph (and thus every planar map) can be colored with four colors [4].

The graph coloring problem is widely applied in the cartography area for coloring maps [5, 6], as any planar map can be transformed into a planar graph representation with vertices defining states, countries or arbitrary regions, and edges reflecting adjacency relationships. Naturally, one way of obtaining a trivial, non-optimal solution is to assign a single different color to each vertex, but doing so would create a map with many similar adjacent hues when a large number of vertices are present. Therefore, it is desired to reduce the quantity of used colors to a minimum.

The graph coloring problem is also used in other fields involving applications that deal with assignment restrictions, such as register allocation for compiler optimization [7, 8]. Registers are faster than RAM and cache memory for value access, although they come in fewer units. Given a large set of variables (long-lived or temporary), the compiler has to decide which register should allocate which variable, possibly reusing the same register across multiple variable assignments. The goal is to improve execution performance by storing as many independent variables concurrently, since a register can only hold the result of a single computation at a given time and no two variables assigned to the same

register can live concurrently. A solution can be obtained by applying the graph coloring algorithm, where temporary variables are represented by vertices, registers are represented as colors, and constraints are given by an interference graph, where edges connect the variables that live simultaneously in the program.

Besides register allocation, detecting bipartite graphs is another application of the graph coloring problem. A graph is said to be bipartite if its vertices can be rearranged into two disjoint sets such that all edges connect vertices from different sets. If the chromatic number is lower than or equal to 2, then the graph is bipartite [9].

Finally, a fourth use case is generating a schedule for school classes where one teacher may lecture multiple subjects, implying that two subjects cannot be lectured at the same time by the same teacher. The problem can be modeled as a graph coloring instance, with each vertex representing a subject, and edges connecting subjects that contain the same teacher as lecturer.

Graph coloring is considered to be in the category of combinatorial optimization, where solutions are optimal objects from a finite set of objects [10]. More specifically to the problem, an *optimal solution* is one that partitions graph vertices into the smallest set of independent sets [11]. An *independent set* is a set of vertices where each vertex is not adjacent to any other vertex contained in it. The *chromatic number* is defined as the number of colors used in an optimal solution.

The graph coloring problem was proven to be NP-Hard by Stephen Arthur Cook [12]. Let G be some graph to be analyzed. The main coloring problems can be expressed by the following questions [11]:

1. Can G be colored with at most k colors?
2. What is the chromatic number of G?
3. What is some optimal coloring of G?

The three questions above are interconnected, although the latter is the most desirable and difficult one to be answered. By performing an enumeration in the space of solutions one can find an optimal coloring in exponential time. As the amount of vertices and edges grow in a graph, coloring it tends to become an intractable problem [13].

Extensive research has been conducted to overcome the intrinsic complexity of the graph coloring problem. The development of heuristics have been successful for applications where the optimal solution is desired but not necessary. For example, both the Classic Greedy Coloring heuristic and DSATUR heuristic were developed to obtain approximate solutions in polynomial time, with DSATUR usually offering best results (closer to optimal solutions) [13].

Another area of research that has helped to simplify the problem is the study of upper and lower bounds for the chromatic number. Setting boundary ranges provides more insight about the problem to be solved, although it still leaves a wide room of possibilities. By using them, it is possible to reduce the search space considering only combinations where the chromatic number is between certain bounds. For example, an upper bound related to the maximum vertex degree ($\Delta(G) + 1$) can be applied to any graph. An even more specific upper bound of four colors can be used only on planar graphs [4], although it is often not implemented in exact algorithms, because denser graphs tend to have a smaller probability of being planar. Likewise, a lower bound of some maximal *clique* size can be used to avoid redundant coloring assignments and branching during a complete search for solutions. By combining these approaches with techniques to avoid coloring symmetries, exact algorithms for graph coloring can discard a great part of the initial search space by using the branch-and-bound strategy with backtracking.

The search space for the graph coloring problem can be further reduced by pruning unfeasible branches with *constraint propagation*. First, the problem is modeled as a *Constraint Satisfaction Problem* (CSP), described by a triple of *variables*, *domains* and *constraints*. In case one or more domains become empty, the branch can be discarded earlier, as no more valid solutions may be obtained from the current assignment. Given a *Constraint Network*, which represents a set of variables, constraints and their interrelationships, constraint propagation is achieved by applying an algorithm that guarantees *Generalized Arc Consistency (GAC)*, where "every value in a domain is consistent with every constraint" [14]. These are called *Propagator Functions*, capable of pruning off unfeasible assignments and invalidating previously assumed solutions. Some known algorithms are AC-3, MDDC, and STR2, as well as the *GenTree Algorithm* [15].

1.1 Goals

The goal of this work is to implement and compare the performance of heuristics and exact constructive algorithms such as the Arbitrary Ordering Coloring algorithm, DSATUR, and some of its variations (Sewell and Pass) [13, 16], by measuring their running time and creating step-by-step visualization tools to analyze color assignments and heuristic-dependent metrics. The performance of each algorithm will be compared on a set of graphs from the DIMACS challenge [1]. For heuristics implementations, the primary goal is to analyze both the running time and how close the obtained results are to the chromatic number of a DIMACS graph. For exact algorithm implementations, the primary goal is to find the fastest algorithms for a given graph type, with the secondary goal being to analyze the size of generated backtracking trees during execution. For the CSP approach,

[1] Challenge information and datasets are available at https://dimacs.rutgers.edu/programs/challenge/ and https://mat.tepper.cmu.edu/COLOR/instances.html

we also analyze the execution of AC-3 propagators, capable of achieving GAC for the contraint network of the graph coloring problem.

1.2 Text Structure

The rest of this document is divided as follows. Section 2 contains a brief history about graph coloring origins and formal concepts from a more technical perspective. In Section 3, we will understand about the problem's intractability, lower and upper bounds for the chromatic number, and will introduce algorithms for obtaining both exact and approximate solutions, as well as some concepts related to the CSP approach. In Section 4, we explain our methodology and algorithm implementations used for comparing graph coloring instances, and in Section 5 we present our test bench implementation, which will coordinate the benchmarking of our algorithms. Section 6 introduces a visualizer which can be used to improve understanding of some colored graphs. Finally, in Section 7 we will demonstrate the obtained results and make considerations about the overall performance of the tests, as well as to suggest further improvements and final conclusions.

2 Formal Definitions

In this section, we present the fundamental concepts for the study of algorithms for graph coloring by first introducing some definition of graphs and later formalizing graph coloring concepts.

Definition 1. *An* (undirected) graph *is an ordered pair* $G = (V(G), E(G))$ *where* $V(G)$ *is a set containing elements called* vertices *of the graph* G*, and* $E(G)$ *is a set disjoint from* $V(G)$ *containing elements called* edges *of the graph* G *which are pairs of vertices.*

Definition 2. *Two vertices* $v_1, v_2 \in V(G)$ *are* adjacent *if there exists an edge* $e \in E(G)$ *such that* $e = \{v_1, v_2\}$*. We also write* $e = v_1v_2$ *for convenience. A vertex is* incident *with an edge if the vertex is one of the two vertices of the edge. Two edges are said to be* adjacent *if they share a vertex.*

Example 1. *The following sets define a graph* G *with eight vertices and 11 edges:*

$$V(G) = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$$
$$E(G) = \{x_1x_2, x_1x_3, x_1x_8, x_2x_3, x_2x_8, x_3x_4, x_3x_8, x_4x_6, x_4x_7, x_5x_6, x_6x_7\}$$

Figure 1 is a visual representation of the undirected graph given in Example 1, where vertices are drawn as circles and edges are drawn as lines connecting adjacent vertices.

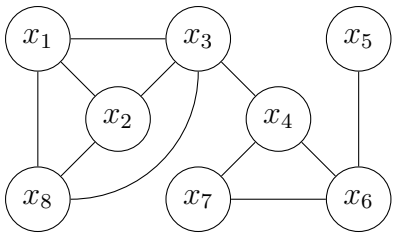

Figure 1 – Graphic representation of the graph of Example 1.

Definition 3. *The* degree $deg(v)$ *of some vertex* v *is defined as the number of edges that are incident to* v*.*

In Example 1, $deg(x_1) = deg(x_2) = 3$, $deg(x_3) = 4$, and $deg(x_7) = 2$.

Definition 4. *A* complete graph *is a graph where all of its distinct pairs of vertices are adjacent. By convention,* K_n *denotes the complete graph that contains exactly* n *vertices.*

Figure 2 shows some examples of complete graphs.

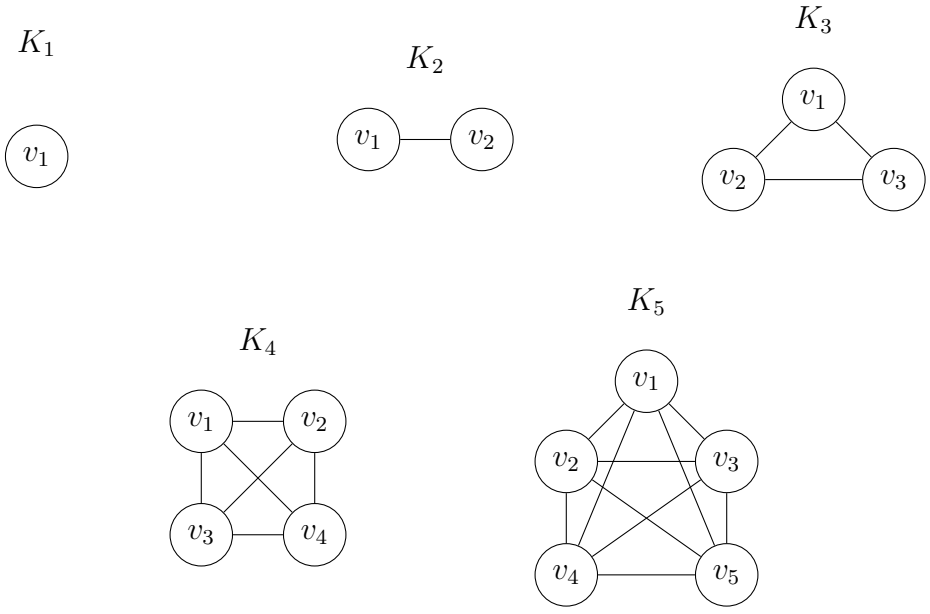

Figure 2 – Examples of K_n for $1 \leq n \leq 5$.

Definition 5. *A* cycle graph C_n*, with $n \geq 3$ is a graph defined by $V(C_n) = \{v_1, \ldots, v_n\}$ and $E(C_n) = \{v_1v_2, v_2v_3, \ldots, v_{n-1}v_n, v_nv_1\}$. A* wheel graph W_n*, with $n \geq 4$, is composed by the cycle C_{n-1} plus a vertex v_n connected to all other vertices of the cycle.*

Definition 6. *A* partition *of a set is a grouping of its elements into subsets such that none of the subsets are empty and every element is contained in exactly one subset.*

Definition 7. *A* bipartite graph *is a graph whose vertices can be partitioned into sets V_A and V_B such that for all $e \in E(G)$, e connects two vertices from distinct sets.*

Figures 3 and 4 illustrate the cycle, wheel, and bipartite graphs.

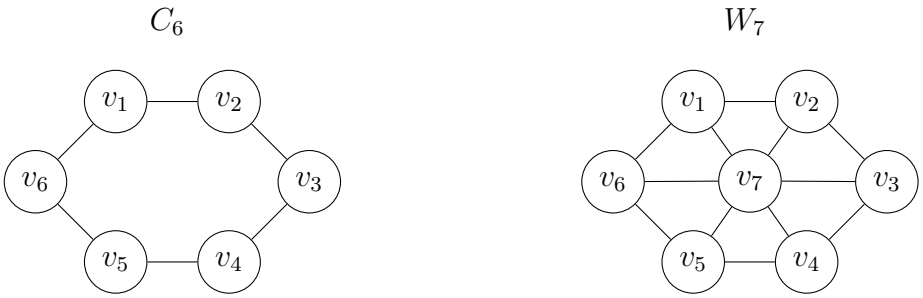

Figure 3 – Examples of C_6 (cycle graph) and W_7 (wheel graph). W_7 contains all vertices and edges from C_6, plus a vertex v_7 which is connected to the other vertices.

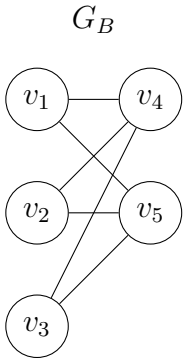

Figure 4 – Example of bipartite graph G_B, with two sets $V_A = \{v_1, v_2, v_3\}$ and $V_B = \{v_4, v_5\}$.

Definition 8. *A graph G' is a* subgraph *of a graph G if $V'(G) \subseteq V(G)$ and $E'(G) \subseteq E(G)$. Given a set $S \subseteq V(G)$, the subgraph $G[S]$ is the* subgraph *of G* induced *by the set S and is defined in a way such that $V(G[S]) = S$ and $E(G[S]) \subseteq E(G)$ contains all edges of G whose endpoints are vertices in S.*

Definition 9. *A* clique *of G is a subset of vertices that induces a complete graph. The* size *of a clique is defined by the number of vertices contained in it.*

Figure 5 shows some cliques contained in the graph of Example 1. Each combination of colored vertices represents a clique contained in Example 1, respectively from left to right and top to bottom: $\{x_1, x_2, x_3, x_8\}$, $\{x_4, x_7\}$, and $\{x_5\}$.

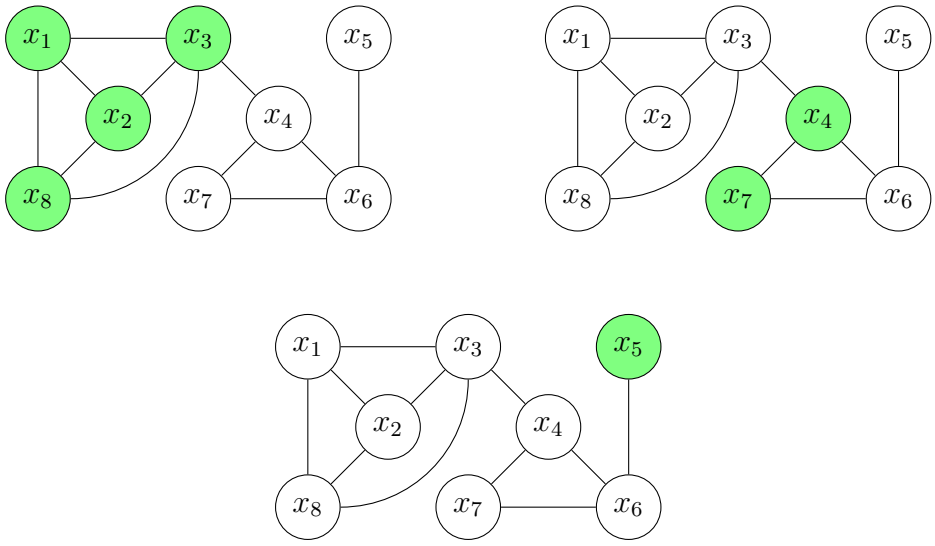

Figure 5 – In green, some cliques contained in the graph of Example 1.

Definition 10. *A* maximal clique *of G is a clique that is not included in a larger clique. A* maximum clique *of some graph G is a clique of largest size in G. The* clique number *$\omega(G)$ is the size of a maximum clique in G.*

A maximum clique of some graph is always a maximal clique, but the converse might not be true. Figure 6 shows some maximal and maximum cliques for the graph in Example 1.

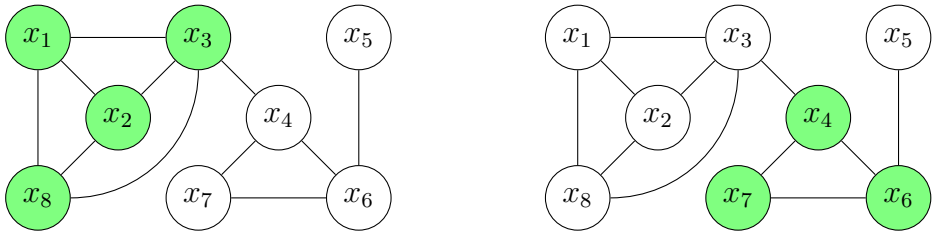

Figure 6 – Maximal cliques for the graph in Example 1. Note that the one to the left is maximum, and thus $\omega(G) = 4$ for the graph in Example 1.

Definition 11. *An* independent set *in a graph is a set of vertices that are not pairwise adjacent.*

Figure 7 shows sets $\{x_1, x_4, x_5\}$, and $\{x_3, x_6\}$, which are independent sets for the graph in Example 1.

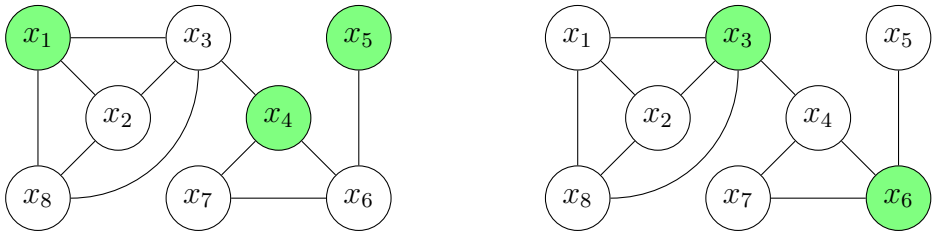

Figure 7 – Some independent sets for the graph in Example 1.

Definition 12. *The* order $|V(G)|$ *of a graph is given by the number of vertices in the graph.*

Definition 13. *The* size $|E(G)|$ *of a graph is given by the number of edges in the graph.*

Definition 14. *The graph* density *of a graph G is given by the ratio between its size and the size of the complete graph K_n with the same order as G, that is*

$$D(G) = \frac{|E(G)|}{|E(K_{|V(G)|})|} = \frac{|E(G)|}{\frac{|V(G)|(|V(G)|-1)}{2}} = \frac{2|E(G)|}{|V(G)|(|V(G)|-1)}.$$

Note that the density of a graph can range between 0 and 1.

Example 2. *The graph in Example 1 has order $|V(G)| = 8$, size $|E(G)| = 11$ and density $D(G) = \frac{2 \cdot 11}{8 \cdot 7} = \frac{11}{28}$.*

2.1 Graph Coloring Definitions

In this section, we present concepts that are specific to the graph coloring problem.

Definition 15. *A* k-vertex-coloring*, for* $k \in \mathbb{N}$*, of a graph* G *is a function* $c : V(G) \to \{1, \ldots, k\}$ *such that* $c(u) \neq c(v)$ *for all* u *and* v *that are adjacent vertices. Additionally, one can see a* k-vertex-coloring *as a partition of* $V(G)$ *into* k *independent sets, where* k *is said to be the number of colors used, and each independent set is called a* color class*.*

Figure 8 shows a 4-coloring for the graph in Example 1, where each color class is represented in the drawing with a different color.

$$f(x_1) = f(x_4) = f(x_5) = 1, f(x_2) = f(x_6) = 2, f(x_3) = f(x_7) = 3, f(x_8) = 4$$

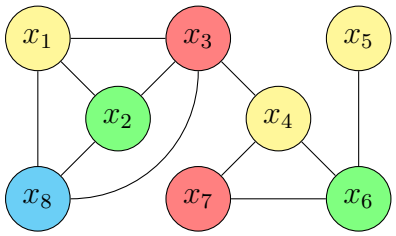

Figure 8 – A 4-coloring for the graph in Example 1 where yellow *represents the color for number 1,* green *for number 2,* red *for number 3 and* cyan *for number 4.*

Definition 16. *A coloring of* G *is* optimal *if it partitions the vertices of* G *into the smallest possible number of independent sets. The* chromatic number $\chi(G)$ *is defined as the number of colors used by an optimal coloring of* G*.*

Example 3. *The coloring in Figure 8 is an optimal coloring for the graph in Example 1 because it partitions the vertices into 4 independent sets and it is not possible to partition them into 3 or fewer independent sets. Thus, the chromatic number for Example 1 is 4.*

Lemma 1. *The chromatic number is greater than or equal to the number of vertices in a maximum clique of* G*, that is,*

$$\chi(G) \geq \omega(G).$$

Proof. Let K be a maximum clique contained in G. Then K contains $\omega(G)$ vertices ($|K| = \omega(G)$). Since all vertices in K are pairwise adjacent in G, then a valid coloring for G must contain at least a unique color for each vertex in K to satisfy Definition 15. □

In Problem 1, we present the definition of the graph coloring problem.

Problem 1. *Given a graph* G *and a* $k \in \mathbb{N}_{\geq 1}$*, can* G *be colored with at most* k *colors?*

There exists a polynomial-time algorithm, which simply uses breadth-first search (BFS), for deciding whether some graph G is colorable when $k = 2$. However, when $k \geq 3$ the problem of colorability is *NP-complete*, as it can be reduced to the *Satisfiability Problem (SAT)*, which was proven to be NP-complete [12] by Cook in 1971. This implies that finding the chromatic number of a graph G is *NP-hard* [13, 17]. In practice, one will often resort to making use of heuristics and trade execution time with colorings that are only approximate to optimal solutions.

3 Algorithms

In this chapter, we will introduce the algorithms implemented in this work that are commonly used to solve the graph coloring problem. We begin by defining the concept of search space, and show some techniques to find the optimal solution using exact algorithms or sub-optimal with heuristics.

3.1 Preliminaries

An algorithm that solves the graph coloring problem must return some k-vertex-coloring that satisfies Definition 15. Often, the optimal value $\chi(G)$ is not known prior to the algorithm execution. Since the graph coloring problem is NP-hard, we have separated the study of such algorithms into two classes: *exact* algorithms, which only return optimal solutions, and *heuristic* algorithms, which return local optimal solutions of the chromatic number of the analyzed graph.

Next, we present some definitions to analyze the possible number of combinations required for obtaining solutions to the graph coloring problem.

Definition 17. *The* search space *of some problem is the set of all the feasible solutions.*

Specifically to the graph coloring problem, the search space can be visually represented by a *tree* structure, where each level corresponds to some vertex chosen from the graph, and each branch represents a color assigned to the vertex in question [11, 13]. The leaves represent complete assignments that are valid. For example, the graph K_2 has a search space tree represented in Figure 9.

The size of the search space is correlated to the running time of the coloring algorithm and therefore it is desired to try to reduce its size as much as possible to improve the overall algorithm performance. A naive attempt, given an arbitrary graph with n vertices, would be to find the optimal solution by enumerating all coloring possibilities, where each of the n vertices contain n possible colors, generating a total number of combinations of n^n. This is clearly intractable for graphs with a large number of vertices.

To reduce the search space size, the solution could then be modeled as a set of independent sets, where given a graph of n vertices it is desired to separate such vertices into sets where vertices in each set are not mutually adjacent. This differs from the initial enumeration approach because it addresses the coloring symmetry issue [13]. For example, in a graph with three isolated vertices v_1, v_2, and v_3, with each coloring identified respectively in an ordered set (c_1, c_2, c_3), the assignments $(0, 1, 2)$, $(1, 2, 0)$ and $(2, 0, 1)$ are isomorphic.

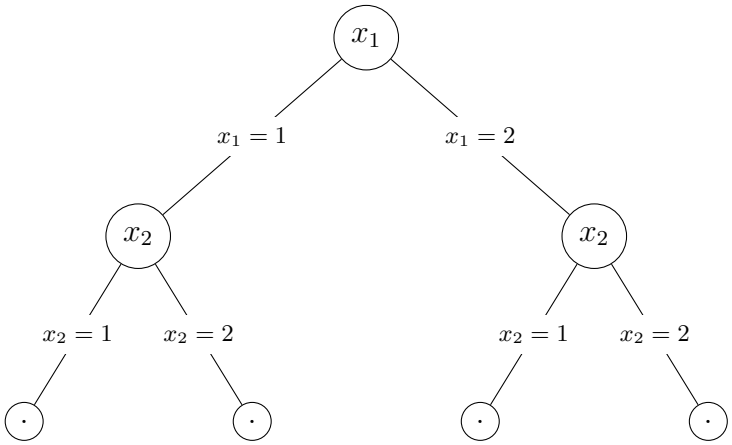

Figure 9 – Search space tree for the complete graph with two vertices (K_2) and coloring possibilities $x_1 = \{1, 2\}$, $x_2 = \{1, 2\}$.

Definitions 18 and 19 give us the total number of combinations obtained for the *partition* approach.

Definition 18. *The* Stirling number *defines the number of ways that n items can be partitioned into k non-empty subsets. It is defined as:*

$$\left\{ \begin{array}{c} n \\ k \end{array} \right\} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \left(\begin{array}{c} k \\ j \end{array} \right) j^n.$$

Definition 19. *The maximum number of combinations C to be analyzed from $k = 1$ up to $k = \chi(G)$ for the optimal solution is given by:*

$$C = \sum_{k=1}^{\chi(G)} \left\{ \begin{array}{c} n \\ k \end{array} \right\}.$$

Using the number C, an algorithm could attempt all k-colorings starting from $k = 1$ until some solution where $k = \chi(G)$ [13]. Unfortunately this strategy is also intractable for larger graphs, as it takes an exponential number of attempts to get closer to $k = \chi(G)$.

Instead of applying the enumeration techniques described above, making use of upper and lower bounds and heuristics can provide us more insight on a better starting point for k and more approximate results for non-optimal scenarios, which we will discuss more in-depth next.

3.2 Constructive Graph Coloring Algorithms

Here, we introduce the concept of *Constructive Algorithms*, which by definition create the solution by assigning a color to each vertex one at a time [13].

3.2.1 Heuristics

Heuristics have been developed to find approximate solutions when the general solution is considered to be too costly in terms of running time. Instead of exploring all of the search space, heuristic algorithms will simply return one feasible solution without guaranteeing optimality. We will now present two heuristics that will be used throughout our work.

3.2.1.1 Classic Greedy Coloring Heuristic

The Classic Greedy Coloring Heuristic (CGCH) is the simplest and most intuitive coloring heuristic. Given a graph, an arbitrary ordering of its vertices is initially generated and will be subsequently iterated by the algorithm. During the setup phase, the heuristic initializes $totalColors = 0$, as the coloring procedure begins with no vertices assigned to any of the colors. In the iteration phase, each vertex will be analyzed in an attempt to get the next available color, which is a color that none of its adjacent vertices have been assigned to. This is done by the FindNextAvailableColor function. However, if all colors have been analyzed but none satisfy the graph constraints, then a new color should be created and assigned to the vertex, being followed by an update to the *totalColors* value. The output of the algorithm will be an ordered list where each value represents a color for the respective vertex index. The CGCH algorithm is the fastest and simplest heuristic, however it rarely produces the best results when compared to other algorithms. A pseudocode is illustrated in Algorithm 1.

Algorithm 1 ClassicGreedyColoringHeuristic(G)

```
1: let vertexColors be an array of size |V(G)|
2: totalColors ← 0
3: for each v in V(G) do
4:     color ← FindNextAvailableColor(G, vertexColors, v, totalColors)
5:     if (color == −1) then
6:         vertexColors[v] ← totalColors
7:         totalColors ← totalColors + 1
8:     end if
9: end for
10: return vertexColors
```

The CGCH has worst-case time complexity of $O(n^2)$, because the outer loop in Algorithm 1 will call FindNextAvailableColor for each vertex, which obtains the vertex color by comparing it against its at most $n - 1$ neighbors, thus generating at most $n \cdot (n - 1)$ comparisons in total, and resulting on a time complexity of $O(n^2)$.

The order in which vertices are assigned colors in the CGCH will also influence the quality of the final result.

Lemma 2. *When given the vertices of some feasible solution sorted by respective color classes, the CGCH will produce a solution whose number of colors is less than or equal to the number of colors in the provided solution.*

Proof. Let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be the partition of all vertices into color classes for some feasible solution. By analyzing each of the vertices for each color class in ascending order, the CGCH algorithm will try to assign a color to it that has its index lower than or equal to the current color in the feasible solution. □

Note that sets $S_1, \ldots, S_k$ from a partition $\mathcal{S}$ and the vertices contained inside the sets themselves may be fed to the CGCH algorithm in different permutations, implying that more than one input choice may lead to the optimal solution. The number of permutations resulting in the optimal solution when fed to the CGCH algorithm is given by the following equation [13], where $k = \chi(G)$ is the number of colors of the optimal solution (chromatic number) and S_i represents a color class from the solution:

$$\chi(G)! \prod_{i=1}^{\chi(G)} |S_i|!$$

Using the Classic Greedy Coloring Heuristic, it is possible to set an upper bound to the chromatic number, as explained in Lemma 3.

Definition 20. *The maximum degree of a graph G, denoted as $\Delta(G)$, is the maximum degree of all vertices in G.*

Lemma 3. *For any graph G, $\chi(G) \leq \Delta(G) + 1$.*

Proof. Let $v \in V(G)$ be some vertex about to be colored by the CGCH. Given $deg(v)$, the number of neighbors of v already colored is not greater than $deg(v)$, and by definition $deg(v) \leq \Delta(G)$. Therefore, in the worst case, v will require a new color $\Delta(G) + 1$, implying that $\chi(G) \leq \Delta(G) + 1$. □

3.2.1.2 DSATUR

Created by Daniel Brélaz in 1979, DSATUR is a widely used graph coloring algorithm [18]. Unlike the CGCH, DSATUR rearranges the order of vertices dynamically during the execution based on the concept of *degree of saturation*. At any given time, the DSATUR heuristic chooses the vertex with the highest saturation degree, which is the vertex with the highest number of colors used by its adjacent vertices. Ties are broken by choosing the vertex of largest degree and, in case of a further tie, any of the compared vertices may be chosen. The goal of using degree of saturation for ordering is to prioritize vertices that have the least amount of colors remaining for use.

Algorithm 2 DSATURHeuristic(G)

```
1: let vertexColors be an array of size |V(G)|
2: totalColors ← 0
3: dsaturOrdering ← (1, 2, ..., |V(G)|)
4: for i = 1 to |V(G)| do
5:     dsaturOrdering ← ReorderDSATUR(G, dsaturOrdering, i)
6:     v ← dsaturOrdering[i]
7:     color ← FindNextAvailableColor(G, vertexColors, v, totalColors)
8:     if (color == −1) then
9:         vertexColors[v] ← totalColors
10:        totalColors ← totalColors + 1
11:    end if
12: end for
13: return vertexColors
```

A pseudocode is illustrated in Algorithm 2. The *dsaturOrdering* array is responsible for maintaining the order of vertices of the *vertexColors* array based on their degree of saturation. On each iteration, the ReorderDSATUR function gets the next unvisited vertex that contains the highest degree of saturation, breaking ties with largest degree and subsequent ties with arbitrary choices, respectively, and swaps the iterated vertex index with the newly chosen vertex in the $dsaturOrdering[i]$ position. Then, the DSATUR index is translated back to the vertex index in $vertexColors$ and assigned to a variable v. From this point on, the coloring of $vertexColors[v]$ is identical to the CGCH until the next iteration, with the difference being that *dsaturOrdering* updates the DSATUR ordering dynamically, unlike the CGCH that has a fixed ordering since the beginning. Its complexity is also $O(n^2)$, although it runs slightly slower than the CGCH algorithm because it needs to keep track of the saturation degree vertex order.

The DSATUR heuristic usually provides better solutions than the Classic Greedy Coloring heuristic, and it always returns the optimal solution for specific types of graphs such as cycle, wheel, and bipartite [13]. Moreover, DSATUR always finds a maximal clique for the first colored vertices which can be later used to compute a lower bound for the graph's chromatic number.

Both CGCH and DSATUR heuristics are not exact algorithms and thus are not suitable for all use cases. In order to get the optimal solution, one must use an exact algorithm with the trade-off of a potentially exponential complexity. Several strategies have been created for obtaining exact results, including integrating constructive algorithms with backtracking, which we will see next.

3.2.2 Exact Algorithms

An exact graph coloring algorithm must potentially explore the entire search space of color assignments to find an optimal solution. At first sight, exact algorithms could enumerate all combinations of the problem instance, but this approach would not work well for larger and/or denser graphs. Here, we discuss some techniques used to reduce the search space size, and we present a version of an exact, constructive graph coloring algorithm.

3.2.2.1 Tight Ordering

Enumerating the entire search space of solutions does not cancel the symmetry shared between multiple assignments. To avoid exploring all permutations, Brown created the concept of a tight ordering [16], where vertices and colors must respect the following restrictions. Let $c(v_i)$ be a function that returns some label for the vertex color v_i, $colors(i)$ be a function that returns the number of assigned colors in the current partial coloring up to the vertex i, and let $V(G) = \{v_1, \ldots, v_n\}$ with $n = |V(G)|$. Then c is a tight coloring if:

$$\begin{aligned} & c\left(v_1\right) = 1 \\ & c\left(v_{i+1}\right) \leq \operatorname{colors}(i) + 1 \quad \forall i = 1, \ldots, n-1 \\ & \operatorname{colors}(i) = \max_s c\left(v_s\right) \quad 1 < s \leq i \end{aligned}$$

Tight coloring restrictions mitigate the coloring symmetry issue by forcing all labels to respect the maximum number of colors in a partial coloring. The label of a vertex v_i is bounded by the formula using $c(v_i)$, and the number of colors in a partial coloring of i vertices is given by the formula for colors(i). For example, when constructing a feasible solution, vertex v_1 must always be colored with the label 1, and vertex v_2 must contain a label not greater than 2. Assuming that $c(v_1) = c(v_2) = 1$, then $c(v_3)$ will be limited by the maximum color of its predecessors such that $c(v_3) \leq 2$, implying that $c(v_4) \leq 3$ and so on. In this way, part of the search search space tree that was already represented symmetrically by previously visited branches will be avoided.

3.2.2.2 Backtracking

Backtracking is a procedure used to refine brute-force search algorithms by possibly pruning the search tree at an earlier stage. It eliminates the need for an explicit check of all candidate solutions of the problem by discarding partially invalid assignment branches [19].

A *backtracking* example code for an exact graph coloring algorithm can be seen in Algorithm 3. It receives as input a graph instance, an array for the current assignment and an array for the current best coloring. The algorithm works recursively with a depth-first search approach and only returns the current assignment, but it mutates the *bestColoring* array reference for later use. During the first invocation, both *assignment* and *bestColoring* are empty. The algorithm chooses some uncolored vertex with the GetNextVertex

Algorithm 3 Backtrack(G, *assignment*, *bestColoring*)

```
1: if all vertices are colored in assignment then
2:     if assignment uses fewer colors than bestColoring then
3:         bestColoring ← assignment
4:     end if
5:     return assignment
6: else
7:     v ← GetNextVertex(assignment, G)
8:     for each color in GetVertexColors(v, G) do
9:         if IsConsistent(v, color, assignment, bestColoring) then
10:            AddToAssignment({⟨v, color⟩}, assignment)
11:            inferences ← InferNextColors(assignment, bestColoring)
12:            if inferences ≠ failure then
13:                AddToAssignment(inferences, assignment)
14:                result ← Backtrack(G, assignment, bestColoring)
15:                if result ≠ failure then
16:                    return result
17:                end if
18:            end if
19:        end if
20:        RemoveFromAssignment({⟨v, color⟩}, assignment)
21:    end for
22:    return failure
23: end if
```

method by following some arbitrary criteria (for example, CGCH or DSATUR heuristics), loops through possible colors for the vertex with GetVertexColors, and checks whether each color is consistent with the current partial assignment. If all constraints of the problem remain valid, the tuple $\langle v, color \rangle$ will be added to the assignment. Next, the InferNextColors function will add any extra tuples inferred from the assignment made by the AddToAssignment call. This includes, for example, vertices which now contain only a single color left for assignment. If, however, there are any vertices with no colors available, the InferNextColors call will fail, thus forcing the algorithm to ignore the branch of the new current assignment. The algorithm continues to explore remaining coloring assignments by recursively calling the Backtrack function in the successful branches. Finally, if there are no further possibilities, the algorithm returns either *failure* or a complete assignment [1]. Once the execution is finished, the *bestColoring* variable will contain the optimal coloring for the analyzed graph instance.

3.2.2.3 Enforcing Upper and Lower Bounds

Enforcing upper and lower bounds for the chromatic number, previously seen on Lemmas 1 and 3, may reduce the number of explored combinations of the search space and converge

[1] Original backtracking pseudocode available at https://www.youtube.com/watch?v=lCrHYT_EhDs.

quicker to optimal solutions. During the execution of exact constructive algorithms, the lower bound remains fixed while the upper bound progressively decreases with the application of *branch and bound*. This strategy includes bounding by the chromatic number of current *bestColoring* array and possibly by the clique number $\omega(G)$, explained in more details below.

3.2.2.3.1 Branch and Bound

Branch and bound is a specific type of backtracking procedure that applies a bound evaluation to partial assignments. The goal is to eliminate branches containing partial solutions that, although valid, present a new bound worse than the current one. In graph coloring algorithms, the upper bound can be set to the chromatic number of the last best solution found (usually denoted as k), thus discarding part of the search space that generates any coloring with more than k colors. Both IsConsistent and InferNextColors methods receive *bestColoring* and *assignment* and compare them. If the current assignment contains more colors than the current best coloring, both methods will return failure and stop further exploration of vertices, forcing an earlier backtracking action.

One can modify Algorithm 3 to implement branch and bound. A new variable k should be added to the Backtrack function parameters, initialized with value 0, and it should be mutated whenever a new solution with fewer colors is found (after *bestColoring* is updated). Then, the variable k should be passed to all Backtrack recursive calls, as well as IsConsistent and InferNextColors, which would compare k with the number of colors of the current explored solution, and fail earlier if the current number of colors is not smaller than k.

3.2.2.3.2 Clique Detection

The addition of *clique detection* identifies the size of the clique for the first explored vertices, and stops the execution once the algorithm backtracks to the last visited vertex in the clique. Since all of its vertices are connected, it does not make sense to allow further backward movements as each vertex in the clique will always need a unique color. In order to simplify our pseudo code, the clique detection feature was not added to Algorithm 3, but it could be implemented by keeping track of the first vertices assigned to v that make a clique, and then finishing the algorithm once the recursion reaches back to the last registered vertex. Note that the initial clique found by the backtracking algorithm will not necessarily be the maximum clique of the graph, since the Maximum Clique Problem is NP-complete [13], and the order of the chosen vertices depends directly on the heuristic used by the GetNextVertex function.

3.2.2.4 Combining Features

To summarize, a more complete exact graph coloring algorithm based on Algorithm 3 combined with the features we have mentioned so far will present the following characteristics:

- it uses backtracking with some heuristic to choose vertices wisely;
- it uses the concept of *tight ordering* to explore only a subset of the search space and avoid exploring permutations of previous branch assignments;
- it enforces a decreasing dynamic upper bound for $\chi(G)$ by using branch and bound for the k variable;
- it enforces a fixed lower bound for $\chi(G)$ with some initial clique, and it only colors such clique once.

Based on this new improved algorithm, we will now present some exact heuristics that can be used for vertex selection.

3.2.2.5 Heuristics for Exact Algorithms

The efficiency of backtracking algorithms compared to basic enumeration approaches varies based on the quality of the heuristics used for deciding the next branch assignment. In Algorithm 3, for example, a heuristic called by the GetNextVertex function should ideally choose vertices in a way that generates an inference failure quickly, forcing the algorithm to backtrack and to prune the search space early on. For this reason, our work will study the following exact heuristics, which are based on the non-exact heuristics previously mentioned in Section 3.2.1.

3.2.2.5.1 Arbitrary Ordering

The arbitrary ordering heuristic behaves similarly to the CGCH algorithm, as vertices will be traversed in the same order in which they are declared in the graph. It demonstrates the least predictable behavior, since its efficiency will vary with the input. There are also no guarantees about the size of the initial clique to be found by the algorithm.

3.2.2.5.2 Exact DSATUR

The exact version of DSATUR uses DSATUR as the chosen heuristic with the enumeration approach proposed by Korman [11], where vertices are dynamically reordered both during forward and backward movements. This occurs because new assignments can alter the ordering of vertices based on the degree of saturation. It can be implemented in Algorithm 3 by adding the specific features from Algorithm 2, such as the *dsaturOrdering* variable

and the ReorderDSATUR function call. The DSATUR heuristic finds a large maximal clique initially, although it is not guaranteed to be the maximum clique in the graph [20].

3.2.2.5.3 Sewell Improvements

Sewell [20] proposed three improvements to the original Exact DSATUR algorithm:

- Rearrange vertices to find the maximum clique first: the original Exact DSATUR already finds a clique for the first explored vertices, but it is not guaranteed to be the maximum one, so Sewell proposed to use different algorithms for the task. It is important to note, however, that the problem of finding a maximum clique is also NP-Hard.
- Find better upper bounds: most of the execution runtime improvement happens when reducing the upper bound to match the graph's chromatic number. To achieve this, Sewell's implementation suggested the use of RLF heuristic with TABUCOL before the maximum clique is found, followed by TABUCOL again [13, 20]. An exception to this rule are random geometric graphs, where the suggestion is to use the DSATUR heuristic with tree search depth limited by a constant.
- Different tie-breaking rule: instead of breaking ties by choosing the vertex with the largest degree, Sewell proposes to use the *CELIM* heuristic, defined as follows [20]: let T be the set of tied vertices; for each vertex $v \in T$, for each color c that is still available to v, count how many uncolored neighbors of v also have the color c available; sum all of the counters for each vertex v; the vertex v with the largest sum will be chosen as the winner.

3.2.2.5.4 Pass Improvements

San Segundo [16] extended Sewell's work by modifying the CELIM rule by restricting its application only for candidate vertices. The new rule is named *PASS*, and it is defined as follows: let T be the set of tied vertices; for each vertex $v \in T$, for each color c that is still available to v, count how many uncolored neighbors of v *which are elements of* T ($v \in T$) also have the color c available; sum all of the counters for each vertex v; the vertex v with the largest sum will be chosen as the winner.

The author also noted that the new rule should be applied selectively, because during the initial steps of the search many uncolored neighbor vertices of $v \in T$ might also be elements of T, thus avoiding the desired effect of reducing the total number of comparisons. To fix this, he recommends to apply the modified CELIM depending on the value of a parameter μ, which is defined as:

$$\mu := colors(\Pi) - \rho_{\Pi}(v \in T) \quad \mu \geq 0,$$

where $colors(\Pi)$ is the number of colors in the current partial coloring Π, and ρ_Π is the saturation degree (the number of colors not available for the vertices), which is always the same value for any $v \in T$ (although their colors could differ). San Segundo [16] recommends to apply the PASS rule when $\mu \leq 3$, and to apply the original DSATUR heuristic otherwise.

3.3 Constraint Programming

In addition to constructive graph coloring algorithms, the graph coloring problem can be modeled as a constraint satisfaction problem (CSP). In this section, we will present some concepts related to the CSP approach [15], which can be used with backtracking to further prune the search space.

Definition 21. *A* constraint satisfaction problem (CSP) instance *is a triple* $\langle V, D, C \rangle$*, where* V *is a finite set of variables,* D *is a function from variables to their domains, such that for all* $v \in V$*,* $D(v) \subset \mathbb{Z}$ *and* $D(v)$ *is finite, and* C *is a set of constraints.*

The constraints for CSPs of graph coloring problems are represented by binary inequalities involving adjacent vertices. Since the relationship is mutual, the order of the variables does not matter.

Definition 22. *A* literal *of a CSP is a pair* $\langle v, d \rangle$*, where* $v \in V$ *and* $d \in D(v)$*. An* assignment *to any subset* $X \subseteq V$ *is a set that contains a single literal for each variable in* X*. A* solution *to a CSP instance is an assignment to* V *where all of the instance constraints are satisfied.*

The formulation of domains and constraints in the CSP will influence the overall efficiency of pruning algorithms. The following example intentionally allows for symmetric color assignments.

Example 4. *Instance definition for a CSP named* $P_0 = \langle V, D, C \rangle$ *for Example 1. Each edge from the graph has been converted into a constraint for the CSP and all vertex variables contain initial domains from 1 to 8:*

$$V = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$$
$$D(x_1) = D(x_2) = ... = D(x_8) = \{1, 2, ..., 8\}, C = \{c_1, c_2, ..., c_{11}\}$$
$$c_1 := x_1 \neq x_2, c_2 := x_1 \neq x_3, c_3 := x_1 \neq x_8, c_4 := x_2 \neq x_3,$$
$$c_5 := x_2 \neq x_8, c_6 := x_3 \neq x_4, c_7 := x_3 \neq x_8, c_8 := x_4 \neq x_6,$$
$$c_9 := x_4 \neq x_7, c_{10} := x_5 \neq x_6, c_{11} := x_6 \neq x_7$$

Assuming x_1 is the first vertex to be colored, it could accept eight different labels in the obtained solutions, although in practice only one would be necessary. An obvious

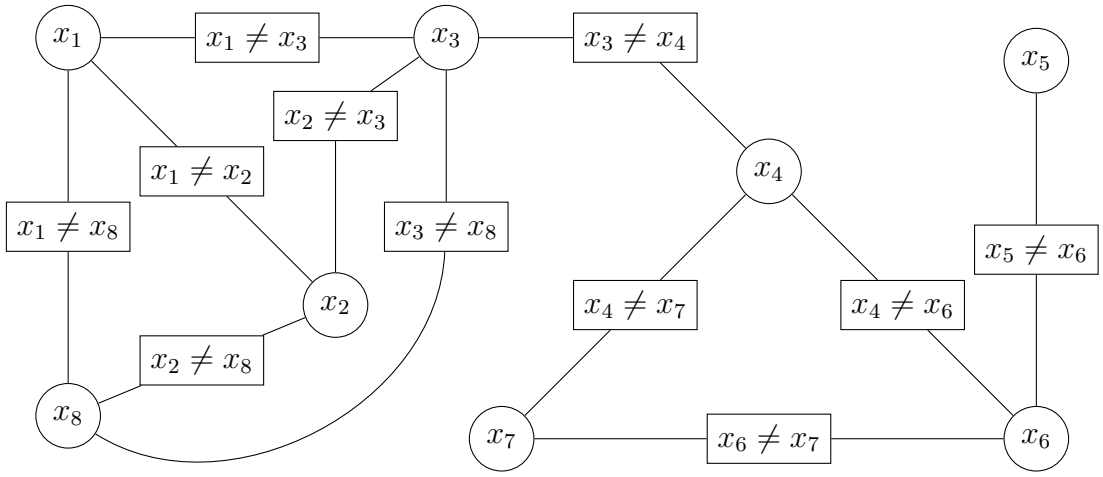

Figure 10 – Graphic representation of the constraint network for the graph of Example 1.

improvement to Example 4 would be to include the constraints presented by the definition of tight coloring created by Brown (see Section 3.2.2.1). Instead, our work will limit the study of CSPs only to the concepts necessary for obtaining arc consistency.

3.3.1 Arc Consistency

Arc consistency allows CSPs to be *pruned*, that is, to have some values removed from its domains before the enumeration of the solutions begins. This potentially reduces the search space given to the constructive algorithm. In order to explain it, we will introduce the following concepts.

Definition 23. *A variable v in a CSP instance is said to be* domain consistent *if all its domain values satisfy all of the constraints defined for v.*

Definition 24. *A* constraint network *for a CSP instance consists of a graph containing a node for each variable, a node for each constraint, and edges that connect each constraint to its variables. Such edges are also called* arcs.

Figure 10 contains a graphical representation for the constraint network for Example 4. Vertices are depicted as circles and constraints as rectangles.

Definition 25. *An* arc $\langle x_i, c_{i,j} \rangle$ *is* consistent *if and only if for all $x \in D(x_i)$ there exists $y \in D(x_j)$ such that $c_{i,j}$ is satisfied by the assignment $\{\langle x, x_i \rangle, \langle y, x_j \rangle\}$.*

An arc that is not consistent can be made consistent by removing unsatisfiable values from the variable domain, since by induction the base case is an empty domain, which is arc consistent. The removal of a value on the variable domain will never remove any general solutions, since a general solution depends on all variable assignments to be valid. If all arcs

of a given network are arc consistent, then the *network* is also *arc consistent*. A *Generalized Arc Consistency Algorithm* receives a CSP, possibly with non-binary constraints, and returns a new equivalent CSP with an arc consistent constraint network.

3.3.1.1 AC-3

The AC-3 algorithm is capable of turning an inconsistent CSP instance into consistent by achieving GAC (Generalized Arc Consistency) on the constraint network. AC-3 requires the conversion of any non-binary constraints into binary ones, but specifically for graph coloring there is no need to since all constraints will only contain two variables. A pseudocode is illustrated in Algorithm 4. First, the algorithm duplicates and inverts each constraint of the given CSP to a list by calling the GetConstraintArcs function. The inversion is required because the left and right side variables of some constraint will be treated differently by the algorithm. To emphasize this, our pseudocode uses a variable named *arcConstraints*, where each original constraint generates two constraints based on the direction of its arcs. AC-3 will then iterate *constraintList* and check if the domain of the left-side variable on each constraint c is value consistent. Based on Definition 25, all values on the left side must contain at least one value in the domain of the right-side variable that satisfies the constraint requirements. The algorithm checks each value in the left domain of the constraint by calling GetLeftDomain, and if some value on the left side is not consistent (LeftValueNotConsistent), then the left side variable will be marked dirty by setting *leftVarDirty* = True, and the RemoveLeftValue function call will remove the inconsistent value from the domain, thus requiring all constraints that reference the targeted variable on the right-hand side to be rechecked. The procedure of adding such constraints to to the *constraintList* will be performed later by the MarkLeftVarDirty function. After some value removal, in case one or more of the analyzed left domains become empty, the algorithm will be halted by the ThrowEmptyLeftDomainError call, as there are no remaining solutions for the CSP. Once each iteration is complete, the analyzed constraint is removed from the list by the RemoveConstraint function call. The algorithm finishes once *constraintList* is empty.

In graph coloring problems, the constraints are defined using the inequality operator, and this allows for an optimization to be made: in case the right hand variable's domain contains at least two values, then all values in the domain of the left side variable are consistent. If the right hand side contains a single value, one may check for its existence in the left side and remove it if positive, otherwise it is consistent. If the left side does not contain it, then nothing should change.

Algorithm 4 AC3(*csp*)

```
1:  arcConstraints ← GetConstraintArcs(csp)
2:  constraintList ← arcConstraints
3:  for each c in constraintList do
4:      leftVarDirty ← False
5:      for each value in GetLeftDomain(csp, c) do
6:          if LeftValueNotConsistent(csp, c, value) then
7:              leftVarDirty ← True
8:              RemoveLeftValue(csp, value)
9:              if LeftDomainEmpty(c) then
10:                 ThrowEmptyLeftDomainError(c)
11:             end if
12:         end if
13:     end for
14:     if leftVarDirty then
15:         constraintList ← MarkLeftVarDirty(constraintList, c, arcConstraints)
16:     end if
17:     constraintList ← RemoveConstraint(constraintList, c)
18: end for
```

3.4 Summary

We have observed in this chapter the complexity of the graph coloring problem, as well as why obtaining optimal solutions are considered to be difficult. The division of constructive algorithms into heuristics and exact algorithms were introduced to separate two common use cases, one in which heuristics trade optimality with faster execution time. We have defined both CGCH and DSATUR variations, with DSATUR usually obtaining better solutions than CGCH [13]. With relation to exact algorithms, we explained the concept of backtracking and how it is related to reducing the search space, along with other techniques such as using branch and bound for reducing upper bounds, finding a clique to reduce lower bounds, and applying tight coloring restrictions to avoid exploring permutations. Further improvements to the exact DSATUR algorithm were also presented, namely Sewell and Pass variations. Finally, we presented another proposed approach, which is to solve the graph coloring problem with the use of constraint satisfaction problems, where it is possible to prune the search space with the use of GAC (Generalized Arc Consistency) algorithms, such as the AC-3 algorithm. Our implementations are explained in the next chapter.

4 Methodology

The present work has the goal of implementing graph coloring algorithms and study their performances on a set of diverse graph problem instances found at the DIMACS Challenges dataset [1]. We will implement and benchmark both approximate (heuristics) and exact algorithm versions: heuristics will be primarily analyzed by how close their solutions are to the graph's chromatic number, while exact algorithms will be analyzed by the time taken to return optimal solutions and the size of their generated backtracking trees. Our algorithm implementations will use the same structural foundation and return only the minimum required data upon process completion, unless specified otherwise on the environment settings.

4.1 Graph Types

The provided DIMACS dataset includes the following graph types and quantities:

- 5 Mycielski graphs: triangle-free graphs where $\omega(G) = 2$;
- 14 Register Allocation graphs: graphs that resemble the allocation of variables during real code execution;
- 12 Leighton graphs: special graphs with guaranteed coloring size;
- 2 Class Scheduling graphs;
- 5 Book graphs: vertices are characters from a book and edges are created for encounters between them;
- 1 Game graph: each vertex is a football team and edges represent matches between teams;
- 5 Miles graphs: created by USA road relationships;
- 13 Queens graphs: related to the n-queens problem;
- 1 Latin Square graph.

Table 1 presents more detailed data about each graph, such as size, order, chromatic number and minimum and maximum degrees.

[1] Challenge information and datasets are available at https://dimacs.rutgers.edu/programs/challenge/ and https://mat.tepper.cmu.edu/COLOR/instances.html

4.1.1 DIMACS Files

The *dimacs* folder in the project source contains a collection of problem instances as DIMACS files from the chosen graph dataset. A DIMACS file commonly uses the *.col* extension, and follows a particular structure where each line may begin with c, p or v:

- c represents lines of comments and therefore are ignored by the graph parser.
- p represents the problem definition, which is expressed on a single line beginning with p, followed by the *edge* keyword and two numbers n and m, where n is the quantity of vertices and m is the quantity of edges in the graph.
- e represents the definition of an edge, followed by two indexes of vertices to be declared as adjacent. Each vertex index must be an integer between 1 and n.

Table 1 – Graphs from the DIMACS dataset to be analyzed by our graph coloring experiments. Chromatic number values are given by the DIMACS source website while order, size, density (D) and minimum and maximum degrees were calculated from file definitions. Column |E| ignores loops and duplicated edges.

Graph Name	**Type**	**|V|**	**|E|**	χ	**D**	δ	Δ
myciel3.col	mycielski	11	20	4	0.36	3	5
myciel4.col	mycielski	23	71	5	0.28	4	11
myciel5.col	mycielski	47	236	6	0.22	5	23
myciel6.col	mycielski	95	755	7	0.17	6	47
myciel7.col	mycielski	191	2,360	8	0.13	7	95
anna.col	book	138	493	11	0.05	1	71
david.col	book	87	406	11	0.11	1	82
huck.col	book	74	301	11	0.11	1	53
homer.col	book	561	1,628	13	0.01	0	99
jean.col	book	80	254	10	0.08	0	36
games120.col	game	120	638	9	0.09	7	13
queen5_5.col	queen	25	160	5	0.53	12	16
queen6_6.col	queen	36	290	7	0.46	15	19
queen7_7.col	queen	49	476	7	0.40	18	24
queen8_8.col	queen	64	728	9	0.36	21	27
queen8_12.col	queen	96	1,368	12	0.30	25	32
queen9_9.col	queen	81	1,056	10	0.33	24	32
queen10_10.col	queen	100	1,470	?	0.30	27	35
queen11_11.col	queen	121	1,980	11	0.27	30	40
queen12_12.col	queen	144	2,596	?	0.25	33	43

Graph Name	**Type**	**|V|**	**|E|**	χ	**D**	δ	Δ
queen13_13.col	queen	169	3,328	13	0.23	36	48
queen14_14.col	queen	196	4,186	?	0.22	39	51
queen15_15.col	queen	225	5,180	?	0.21	42	56
queen16_16.col	queen	256	6,320	?	0.19	45	59
miles250.col	miles	128	387	8	0.05	0	16
miles500.col	miles	128	1,170	20	0.14	3	38
miles750.col	miles	128	2,113	31	0.26	6	64
miles1000.col	miles	128	3,216	42	0.40	13	86
miles1500.col	miles	128	5,198	73	0.64	28	106
zeroin.i.1.col	register	211	4,100	49	0.19	0	111
zeroin.i.2.col	register	211	3,541	30	0.16	0	140
zeroin.i.3.col	register	206	3,540	30	0.17	0	140
mulsol.i.1.col	register	197	3,925	49	0.20	0	121
mulsol.i.2.col	register	188	3,885	31	0.22	0	156
mulsol.i.3.col	register	184	3,916	31	0.23	0	157
mulsol.i.4.col	register	185	3,946	31	0.23	0	158
mulsol.i.5.col	register	186	3,973	31	0.23	0	159
le450_5a.col	leighton	450	5,714	5	0.06	13	42
le450_5b.col	leighton	450	5,734	5	0.06	12	42
le450_15a.col	leighton	450	8,168	15	0.08	2	99
le450_15b.col	leighton	450	8,169	15	0.08	1	94
le450_25a.col	leighton	450	8,260	25	0.08	2	128
le450_25b.col	leighton	450	8,263	25	0.08	2	111
le450_5d.col	leighton	450	9,757	5	0.10	29	68
le450_5c.col	leighton	450	9,803	5	0.10	27	66
fpsol2.i.1.col	register	496	11,654	65	0.09	0	252
fpsol2.i.2.col	register	451	8,691	30	0.09	0	346
fpsol2.i.3.col	register	425	8,688	30	0.10	0	346
inithx.i.1.col	register	864	18,707	54	0.05	0	502
inithx.i.2.col	register	645	13,979	31	0.07	0	541
inithx.i.3.col	register	621	13,969	31	0.07	0	542
le450_15c.col	leighton	450	16,680	15	0.17	18	139
le450_15d.col	leighton	450	16,750	15	0.17	18	138
le450_25c.col	leighton	450	17,343	25	0.17	7	179
le450_25d.col	leighton	450	17,425	25	0.17	11	157
school1.col	school	385	19,095	?	0.26	1	282
school1_nsh.col	school	352	14,612	?	0.24	1	232

Graph Name	**Type**	**|V|**	**|E|**	χ	**D**	δ	Δ
latin_square_10.col	latin	900	307,350	?	0.76	683	683

4.1.1.1 Reading and Parsing Graphs

A *readFile* method was created to parse graph coloring instances. It receives a graph file in the *.col* DIMACS format and creates a graph structure from the notation consisting of vertices and edges. Some of the DIMACS files require the removal of duplicate edge definitions, since some graphs contain edges being defined twice by two edges in opposite directions. Once instantiated, the graph structure contains the following properties:

- n - the total number of vertices;
- m - the total number of edges;
- the *maximum degree* in the graph;
- the *adjacency list*;
- an auxiliary vector that contains the *degree* of each vertex.

4.2 Algorithms

The following graph algorithms will be analyzed by our tests:

- Classic Greedy Coloring Heuristic
- DSATUR Heuristic
- Exact with Arbitrary Order
- Exact DSATUR
- Exact DSATUR with Sewell rule
- Exact DSATUR with Pass rule with and without μ parameter selection
- Exact DSATUR with AC-3 for GAC with $\delta = 0, 1, 2$ parameter

The algorithms were selected for our study because the Pass algorithm is based on the DSATUR and Sewell algorithms, and the Pass algorithm is expected to show performance improvements when compared to the Sewell algorithm, according to the original article [16].

Our custom implementation of DSATUR with AC-3 uses a delta parameter δ to conditionally apply the propagation algorithm during different stages of the backtracking

exploration. Initially, AC-3 pruning is disabled until some feasible coloring is found by the DSATUR heuristic. During each backward or forward step, the AC-3 pruning algorithm will run if $\delta = k - totalColors$. This was required because solving a CSP instance is costly and thus it is desired to limit its number of calls. Through prior analysis, we noticed that $\delta = 0$ was found very often, $\delta = 1$ was common and $\delta = 2$ was rare.

4.2.1 Implementation

The *src* folder of our project contains implementations in C++ for both heuristics and exact algorithms introduced in the previous section [2]. Different algorithms were created in different C++ namespaces and a base structure for enumerating solutions was shared across the exact algorithms, starting from Arbitrary Backtracking and later being copied into the DSATUR variations. Once the program terminates, the optimal solution will be output to a JSON object that contains the algorithm's name along with the following data:

- *colors*: if the algorithm is exact, this property will represent the chromatic number of the graph, otherwise it will represent the number of colors found in the solution for the selected heuristic;
- *time*: time taken from start to end of the execution (in milliseconds);
- *backtrackingVertices*: this is an optional property only defined for exact algorithms. It contains the number of explored vertices in the backtracking tree generated by the selected implementation;

4.2.1.1 Verifying Correctness

When solving problems, verifying the correctness of implementations can be very challenging. In our case specifically, possible faulty scenarios for graph coloring algorithm implementations are divided into the categories below:

1. *The chromatic number returned by the program is given by an invalid coloring.* This implies that at least one color clash between two adjacent vertices has gone unnoticed in the construction of our solution. It can be fixed by creating a function that will verify – in polynomial time – that the provided coloring vector and the graph's adjacency list produce a valid coloring. Our code automatically performs this verification before returning any answer.

[2] Source code for our algorithm implementations can be accessed at https://github.com/rafaelcalpena/graph-coloring

2. *The program did not perform search on branches that were meant to be explored.* Without predefined external information, this scenario is more difficult to detect and address. It can be sub-divided in two possibilities:

    - *The resulting chromatic number is greater than the real chromatic number.* It is possible to detect this bug in case we are provided with the correct result by some external source. The DIMACS dataset used in our tests contains the correct chromatic number for every instance, so these values were used in our Test Benchmark Configuration to catch for any divergences in the final answer.
    - *The resulting chromatic number is still the same as the real chromatic number.* This case is harder to detect, because one key feature of a good heuristic is to reduce the search space so that the optimal result is returned faster. The final answer would still be correct for the analyzed graph, but the program would not execute properly for other graphs that contain the optimal solution in branches that were incorrectly skipped. A way to mitigate such possibility is to test the algorithm with as many graphs as possible. Passing all of the DIMACS dataset instances does not guarantee the correctness of our algorithms, but it does reduce the likelyhood of a bug not being caught in any of the instances. Other mitigation strategies include the implementation of logging functionality for manual inspection and step-by-step debugging of targeted graph instances. We have explored such possibilities and will introduce our created tools next.

4.2.1.2 Logging and Debugging

In order to improve understanding of how our implementations behave for some particular graphs it was convenient to analyze their progress with the help of logging artifacts. We have achieved this by breaking each algorithm into smaller units of work called *log statements*, which include actions such as variable assignments and internal state update calls as well as forward and backward movements during the exploration of the search space. Each log statement is created by calling the *debug* method that receives two parameters: the message and an *fstream* reference for some desired log file as the write target.

The message to be appended takes the form of an action object, declared as JSON document on a stringified format, and contains an *action* key and optionally extra properties depending on its type. Since actions usually contain information about integer vectors and sets, we have created utility functions to serialize such structures into the payload.

4.2.1.2.1 Common Log Actions

Below we have a list of actions that are commonly used throughout our coloring algorithm implementations:

- *set*: defines the update or registration of a new variable where the *key* property corresponds to the identifier of the variable that is being updated and the *value* property represents the updated value. Usage examples of this action are: setting the current coloring vector, current best coloring vector, upper and lower bounds, ordering of vertices for some heuristic, and setting the current targeted vertex;
- *finishInitialSetup*: this action happens exactly once and is triggered before the beginning of the iteration section. During initial setup, variables are initialized and some vertex will be selected to start the exploration of the backtracking tree;
- *iteration*: this action is emitted in the beginning of every *while* iteration step. Our algorithms use an i index for the analyzed vertex position. During each iteration it is possible for i to change into a higher value (move forward into some deeper branch), a lower value (backtrack) or stay the same (on a new worse coloring found, where the chromatic number is greater than the current upper bound);
- *moveForward* and *moveBackwards*: these actions happen when the i index increases or decreases, respectively. They represent branching and backtracking movements in the search space tree;
- *preventSearchInSubBranches*: triggered when some valid coloring has been found but does not satisfy the upper bound limit *(coloringWorseThanLimit)*, or when Generalized Arc Consistency algorithms have detected an empty domain to one of the variables *(gacEmptyDomain)*. Both cases will prevent the exploration of unnecessary sub-branches;
- *foundColoring*: activated when a new coloring has been found. Contains vertex colors as a serialized integer vector for the *value* property. This event is guaranteed to happen at least once during the program execution;
- *stop*: declared when the program backtracks to the first vertex of the chosen ordering and stops. Only happens once during program execution;
- *finalResult*: guaranteed to happen exactly once when final result is obtained. Contains the assigned colors represented in a serialized integer vector for the *value* property;
- *getMaximumDegree*: activated after the program calculates the result of the most recent maximum degree in the graph;
- *getColoringNumber*: activated after the program calculates the most recent number of colors in the graph.

Example 5. *Final output for* `myciel4.col` *graph with* arbitrary-backtracking *algorithm*

```
{
    "arbitrary-backtracking": {
        "colors": 5,
        "time": 22565,
        "backtrackingVertices": 220476
    }
}
```

4.2.1.2.2 Flexibility and Performance Impact

The *debug* method checks for the *DEBUG* environment variable and behaves differently depending on its assigned value:

- when *DEBUG=0*, logging will be skipped;
- when *DEBUG=1*, the logger will append the message to the log file. This is the preferred configuration for analyzing the execution output;
- when *DEBUG=2*, the debugger will append the message to the log file and output it to the terminal. In general, this option is discouraged as it greatly increases the time taken to finish execution and prevents communication between the algorithm's implementation and the test bench, since the latter requires obtaining the final output JSON object (from Example 5).

During development it was observed that using *DEBUG=0* still had a noticeable impact in the overall execution time, therefore a compiler flag *-DDEBUG* and a *DEBUG* C++ macro were also created to remove the performance issue entirely:

- when the *-DDEBUG* flag is *absent* on compile time, the *DEBUG* macros will not be expanded into *debug* calls, thus producing faster benchmarks;
- when the *-DDEBUG* flag is *present* on compile time, the *DEBUG* macro will be expanded into a *debug* method call.

Table 2 displays a comparison of all possibilities for an example of `myciel4.col` graph and *arbitrary-backtracking* algorithm, colored optimally with 5 colors and containing 220,476 vertices in the search space tree.

In the next chapter, we will see more details about the environment in which our tests are executed.

Table 2 – Comparison between -DDEBUG build option and DEBUG environment variable values and their impact on the execution time of coloring for myciel4.col graph with arbitrary-backtracking algorithm.

Configuration	Execution Time (in *ms*)	Recommended For
with -DDEBUG and DEBUG=2	111,592	–
with -DDEBUG and DEBUG=1	22,565	Logging
with -DDEBUG and DEBUG=0	4,097	–
without -DDEBUG	1,724	Benchmarking

5 Test Bench

A test bench setup was created to help analyze, coordinate and verify the results of our graph coloring algorithm implementations [1]. By default, the test bench will instantiate all possible combinations of algorithms and graph files described on Table 1, and will save reported results into a concatenated JSON file. When a solution is obtained, it should verify that optimal colorings for exact algorithms also match the chromatic numbers from the solutions provided by the DIMACS dataset [2]. Since there are many combinations of DIMACS graphs and coloring algorithms, the test bench may distribute them across different cores when possible. Finally, it should also allow for a maximum timeout setting to finish the algorithm's execution in case the program's execution does not return an optimal answer within the desired time.

To make this reproducible, our test bench was designed to run as a Docker container, which encapsulates all required dependencies (such as the *gcc* compiler) on an immutable image composed by our software stack along with a specific system kernel version for Linux. We will see later that this architecture also allows the test bench to run on cloud service providers. Given that all our studied algorithms are of deterministic nature, only minor fluctuations (usually in milliseconds) are expected to occur between multiple runs with the same configuration.

5.1 Implementation

The test bench was written in JavaScript and runs on Node.js. During each test instance, it will invoke the desired C++ algorithm implementation by calling *execFile* from Node's *child_process* native module and await for the returned JSON summary (see example 5). Our algorithms are always compiled *without* the *-DDEBUG* flag when running within the test bench to remove any performance penalties caused by logging statements (see Table 2 for an example comparison).

5.1.1 Environment Variables

It is possible to customize test suite settings by configuring the following environment variables:

[1] Source code for the test bench can be acessed at https://github.com/rafaelcalpena/graph-coloring

[2] Challenge information and datasets are available at https://dimacs.rutgers.edu/programs/challenge/ and https://mat.tepper.cmu.edu/COLOR/instances.html

- *TIMEOUT* (in milliseconds): Defines the timeout limit for algorithm execution. Default value is 10 minutes (600,000 ms).
- *FILES*: Defines a list of comma-separated graph filenames to be colored in the test suite. By default, all graph instances from the DIMACS Challenge Dataset will be used.
- *ALGORITHMS*: Defines a list of algorithms to be included in the test suite. By default, all of our algorithm implementations will be analyzed.

5.1.2 Test Modes

The testing framework was written to support two modes: *local* and *remote* (cloud-based).

5.1.2.1 Local Mode

With local mode, one can run test suites on a local machine by installing Docker and running the *docker-compose.yml* file located at the *graph-coloring* folder of the project's source. A summary-like JSON file will be generated to the *output* folder and will be updated when new results are emitted. Each graph file will be analyzed sequentially by spawning different algorithm implementations at the same time. Once all algorithms are complete, the test suite moves on to the next graph instance.

Since obtaining optimal coloring solutions for graph instances is a slow procedure especially in larger and/or denser graphs, using the local mode is only recommended in specific cases such as when running on a powerful machine, using a lower value for the timeout configuration or running to assert that the test bench setup itself is working properly. It is also not ideal because the number of algorithms running in parallel may not be the same as the existing number of cores in the machine, and thus concurrent processes may affect overall results.

5.1.2.2 Remote Mode (Cloud-Based)

With remote mode, test scripts are located in the *cloud* folder of the source project, and the workflow is composed by semi-automated actions. This mode requires a cloud provider (AWS in our case) and obtains results by distributing the computation across multiple containers on self-managed virtual machines.

5.1.2.3 Pros and Cons of Cloud Testing

In comparison with local mode, cloud mode presents the following advantages:

- It is only necessary to leave the local machine working during the initial and final testing stages, such as during resource deployment, triggering batch job starts and performing data synchronization.
- It reduces the chance of unpredictable events like power failures halting the test bench execution, since cloud providers are usually redundant against such scenarios.
- Resource utilization for the local machine is significantly reduced.
- The jobs run fully in parallel as each docker container instance behaves as a separate test bench. All results get synchronized to the local machine and concatenated into a single file after test completion.
- The computing power in the cloud is elastic and is limited mostly by account limits. For example, AWS accounts have a limit of 100 simultaneously running containers.

In contrast, the cloud environment also presents disadvantages:

- A provider's user account is necessary for creating and utilizing resources. The *root* user must also have assigned permissions for all cloud services used in the test bench execution.
- Depending on the allocated quantity of computing resources, provider billing can be significant. It is also difficult to predict the total charged amount upfront and it may not be possible to revert billing in case extra resources are utilized unintentionally.
- Cloud providers host different services with different usage limits, causing potential vendor lock-in. Although our tests run inside a container to reduce complexity, pushing and running such containers to different providers require different cloud test bench implementations, and therefore we have limited our testing to a single provider.

5.1.2.4 Cloud Workflow

Running the test bench on a cloud provider can be achieved with the following steps, which are illustrated in Figure 11:

1. A user account is created in the cloud provider's website and assigned the necessary permissions to run all services.
2. User builds and runs the *Dockerfile* located at the *cloud* folder with *docker build* and *docker run* commands, provides necessary volume mounts and cloud login credentials, and accesses the container from the terminal.

3. User runs *start.sh* script, which will automate resources provisioning with *Pulumi* library, and will trigger a job submission via the AWS CLI once the resources are up.

4. AWS Batch will run a job array, that is, a collection of test suites each with a unique combination of an algorithm and a graph instance to be analyzed. Each container instance will be running on a virtual machine that is self managed by the computing provider. Overall progress of the execution can be tracked via the provider's website dashboard.

5. Once all container instances have returned results, user runs the *datasync.sh* script which will transfer all JSON files from the EFS volume shared between containers to an S3 Bucket (storage service).

6. User calls the *concatenate.sh* script to download JSON files from S3 Bucket to local computer and combine them into a single JSON file.

7. User may optionally call *convert-to-csv.js* script to migrate results from JSON file into a *csv spreadsheet*.

5.1.2.5 Job Array Combinations

In addition to previously defined variables, AWS Batch will inject two more environment values during runtime:

- *AWS_BATCH_JOB_ID*: Used to identify and concatenate jobs from the same test suite into a single file.
- *AWS_BATCH_JOB_ARRAY_INDEX*: Used to assign unique combinations of algorithms and graph files into different containers. When the job array index is present the test bench will filter a single combination of an algorithm and a graph file. The corresponding *graph index* (g) is given by the formula i **div** $|A|$ and the *algorithm index* (a) is given by i **rem** $|A|$, where i is the job array index and $|A|$ is the size of the set containing all algorithms to be tested.

Table 3 depicts an example of a matrix of 15 container instances ($max(i) = 14$) distributed over 5 algorithms ($|A| = 5$).

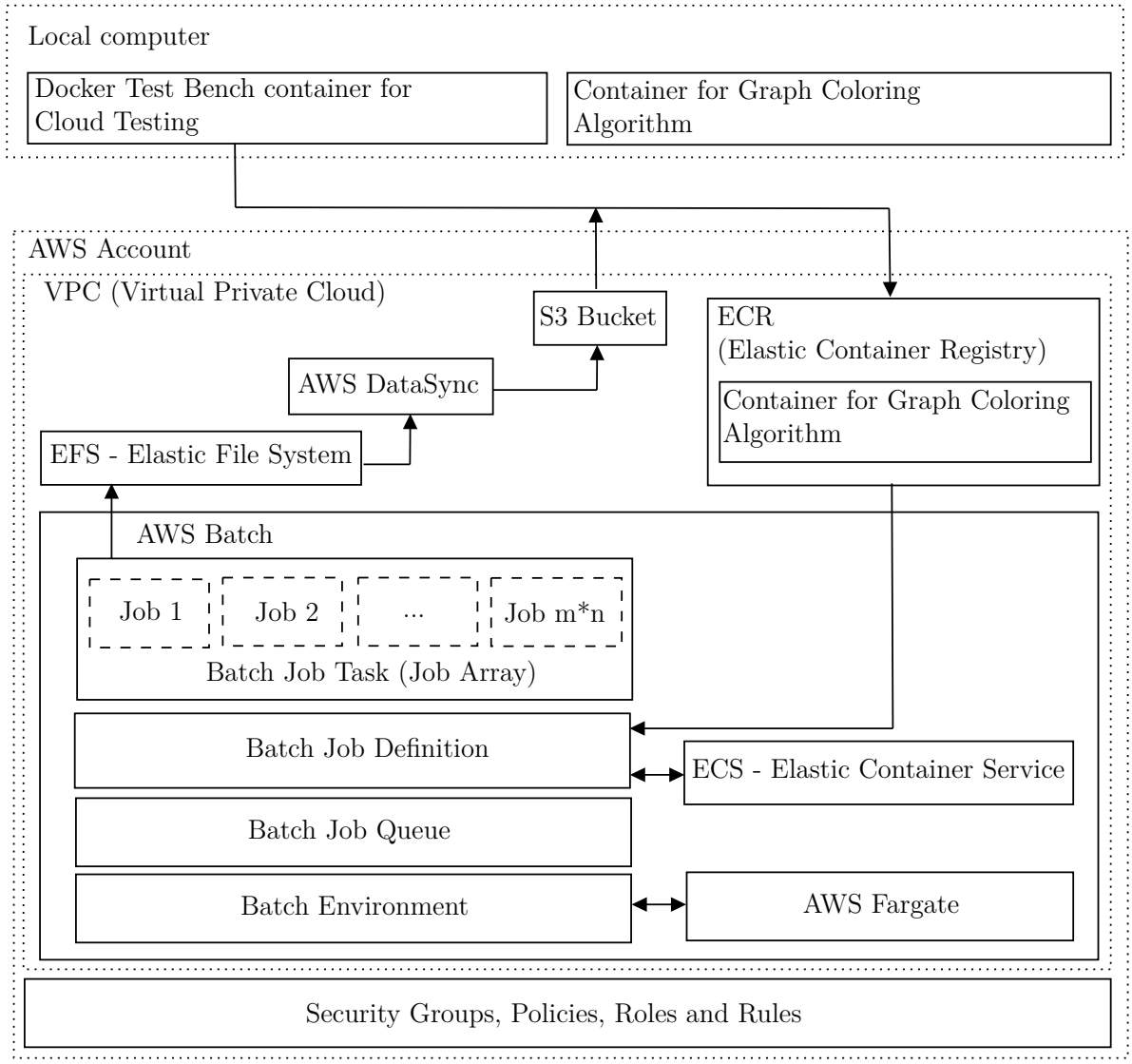

Figure 11 – Cloud Resources Diagram - The testing procedure begins with the upload of resources to ECR, and finishes with the retrieval of the generated data hosted at an S3 bucket.

Table 3 – Example of graph coloring algorithms and file combinations. Each algorithm is represented by a value of a and a graph file by a value of g. Container instances are therefore identified by values of i, which combine the file from the row with the algorithm from the column.

—	$a = 0$	$a = 1$	$a = 2$	$a = 3$	$a = 4$
$g = 0$	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$
$g = 1$	$i = 5$	$i = 6$	$i = 7$	$i = 8$	$i = 9$
$g = 2$	$i = 10$	$i = 11$	$i = 12$	$i = 13$	$i = 14$

In the next chapter, we will see another tool created to aid the development of coloring algorithms, especially for analyzing small graphs.

6 Visualizer

Along with the creation of heuristics and exact algorithms, a web visualizer was built to aid understanding of the generated final results and log files [1]. By reading *debug* statements from JSON output files mentioned in Section 4.2.1.2 (created when running the algorithms with *-DDEBUG* and *DEBUG=1* options), the visualizer is capable of replaying coloring operation steps that were registered during the algorithm's execution, thus allowing the user to specify a given point in time to review the internal state for the current graph coloring and for the search space backtracking tree. The visualizer's user interface contains information such as the algorithm used, number of vertices and edges, maximum and minimum degree of the graph, graph density, adjacency list, chromatic number, and colorings found. It also displays visualizations for the graph and the backtracking tree.

6.1 Implementation

The web visualizer was built using *VueJs* for the UI framework along with *CytoscapeJs* for the graph viewers. It works exclusively for exact algorithms; CGCH and DSATUR heuristics can be visualized by inspecting the the generated backtracking tree for the equivalent exact implementation up until the first solution is found. Figure 12 gives us an overview of the web visualizer interface, which will be explained with more details next.

6.1.1 Features

The user interface for the visualizer is divided in three sections: *left menu*, *graph* and *backtracking tree*. The *left menu* contains the following components, shown in Figures 13 and 14:

- *Info*: Presents useful information about the targeted graph file, such as *file name*, *algorithm* used during coloring procedure, vertex count, edge count, maximum and minimum vertex degrees and graph density (for simple graph types).
- *Adjacency List*: Contains a list of numbers and corresponding vectors where each number represents an adjacency relationship between both vertices.
- *Chromatic Number*: The chromatic number given by the algorithm's result with relation to the analyzed graph.

[1] The web visualizer can be accessed at https://rafaelcalpena.github.io/graph-coloring

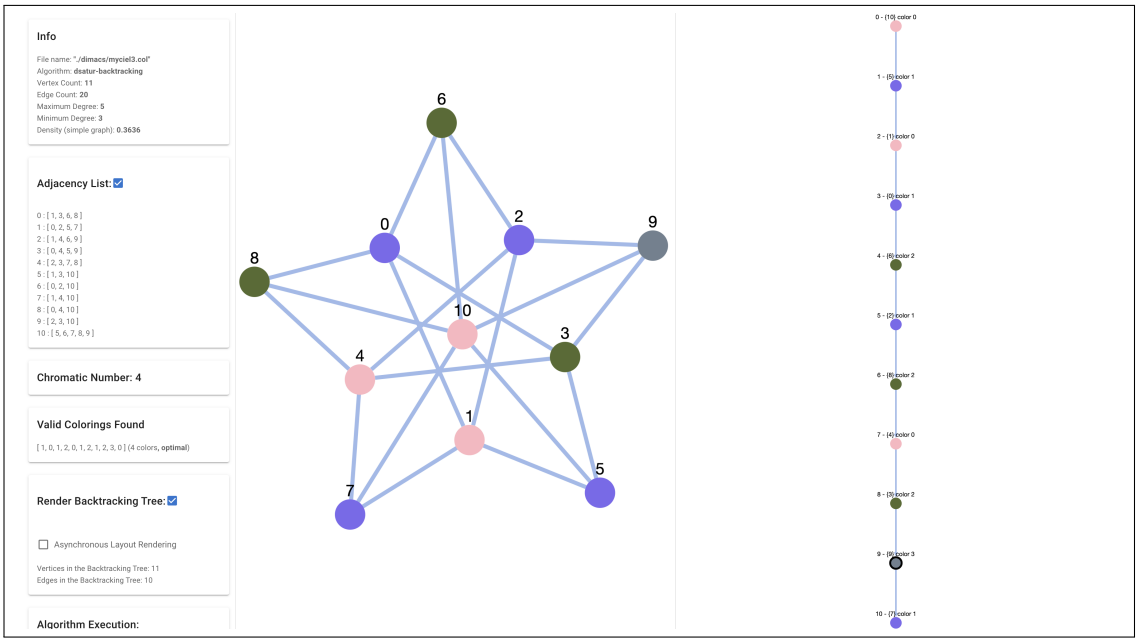

Figure 12 – Web visualizer interface, divided into left menu, graph and backtracking tree sections.

- *Valid Colorings Found*: Contains all colorings that have been found during execution of the algorithm sorted in descending order. This a characteristic of the branch-and bound-methodology, which will lower the upper bound incrementally throughout search.

- *Render Backtracking Tree*: Enable or disable the rendering of the backtracking tree. This option is present because the backtracking tree becomes the primary performance bottleneck on large graphs, and it could potentially freeze the user interface. It is also possible to use asynchronous layout rendering for a small performance improvement. It is possible to disable the backtracking tree rendering.

- *Algorithm Execution*: Allows the user to control replaying of the results in a step-by-step manner. Contains the *current step* (begins at 0) and the *final step* indicators. There are previous and next buttons for navigating steps, and an *operations* slider allows the user to skip to the action in the desired step. The *speed* slider will fast-forward actions automatically. This is useful for replaying the algorithm's execution in slow motion. The *maximum speed* input adjusts the sensitivity of the speed slider so that larger and smaller values are possible. The *real speed* reports the actual rendered speed in operations per second.

- *Logs*: Contains a list of actions that have been performed during execution. Each action contains a key and optionally extra values. Only a subset of all actions is rendered relative to the current selected step to avoid making the user interface unresponsive. Selecting an action skips to the specific step.

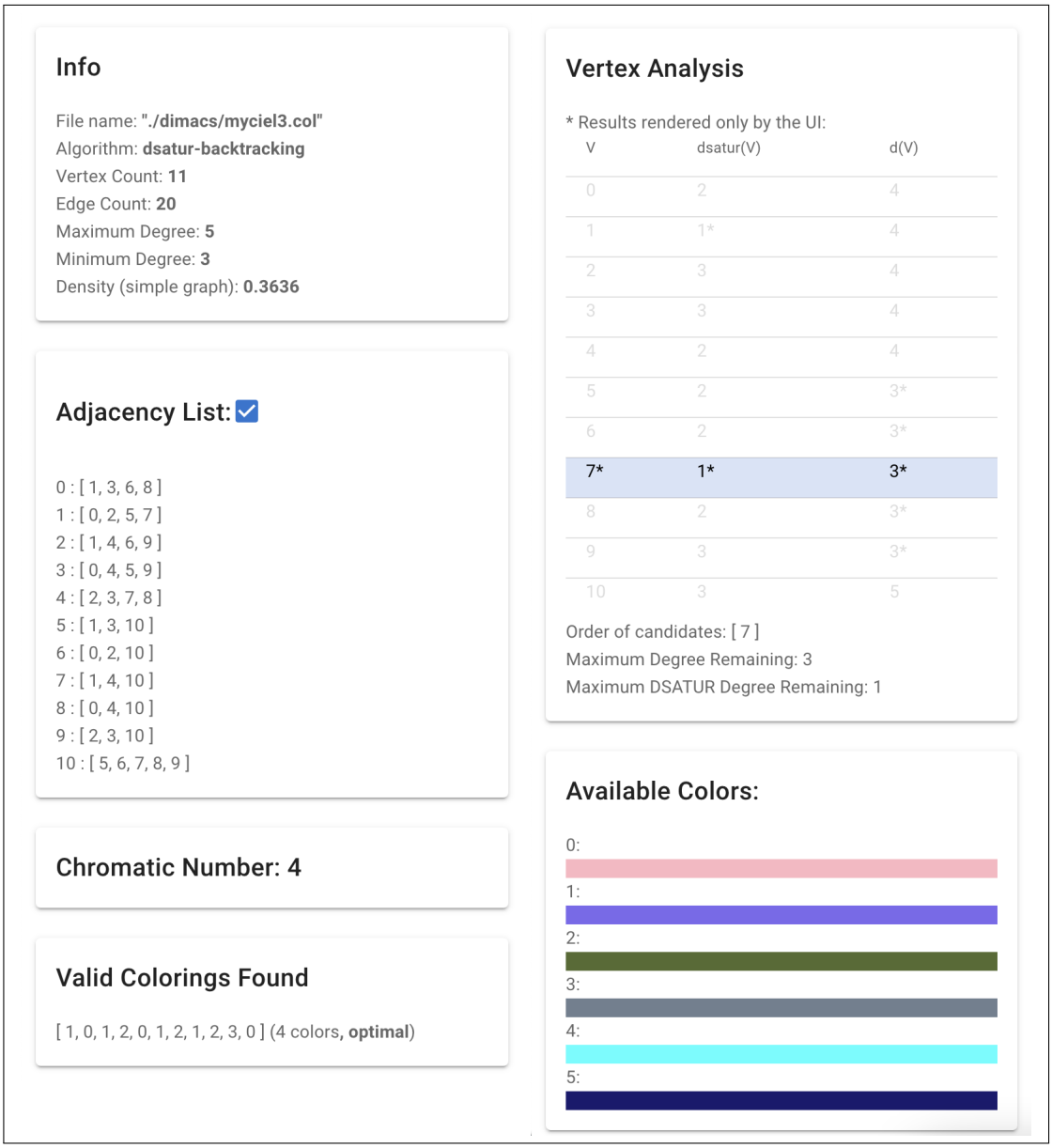

Figure 13 – Left menu components, including Info, Adjacency List, Vertex Analysis, Chromatic Number, Valid Colorings Found and Available Colors panels.

- *Problem Variables*: Contains the current internal state of the execution in the selected step, and usually includes values for the current coloring vector, best coloring vector, upper bound, vertex ordering, current analyzed vertex and current color.
- *Vertex Analysis*: Displays a table with the vertex index and degree and DSATUR degree information, as well as order of candidate vertices.
- *Available Colors*: Displays a list of available colors and respective color indexes. This list is based on the current value for the upper bound defined in the analyzed step.

The *graph section* contains a visual representation of the analyzed graph, where assignments for the current step are translated into background colors. Each vertex can be identified by

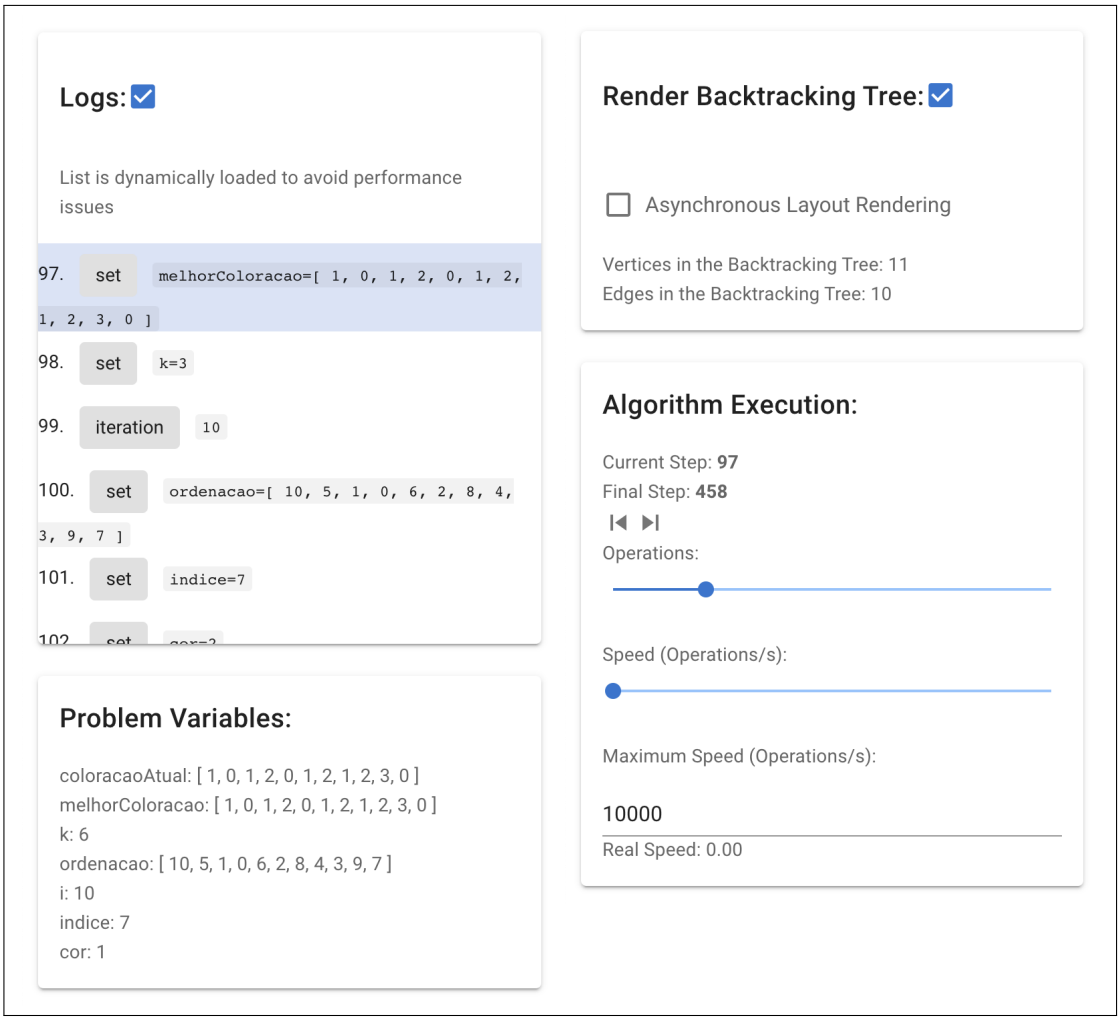

Figure 14 – Left menu components, including Logs, Render Backtracking Tree, Problem Variables and Algorithm Execution panels.

its index located on the top label region. The light grey color implies that the respective vertex lacks a color assignment for the current step. The *backtracking tree* renders visually all assignments that have been tried until the current step. The topmost vertex represents the initially chosen vertex and each branch represents choices for next vertices. Labels contain both the *ordering index* as well as the *selected vertex index* (located inside curly brackets). The current analyzed vertex is covered by a black border.

Figures 15, 16, and 17 show both *graph* and *backtracking tree* sections during different stages of the DSATUR exact algorithm execution for `myciel3.col` graph. At first, the graph to be colored has not been assigned any colors and the backtracking tree is empty (Figure 15); after the first solution is found (Figure 16), the algorithm backtracks several times to possibly find another solution with a lower chromatic number (Figure 17). The algorithm finishes once all possible branches in the backtracking tree have been explored.

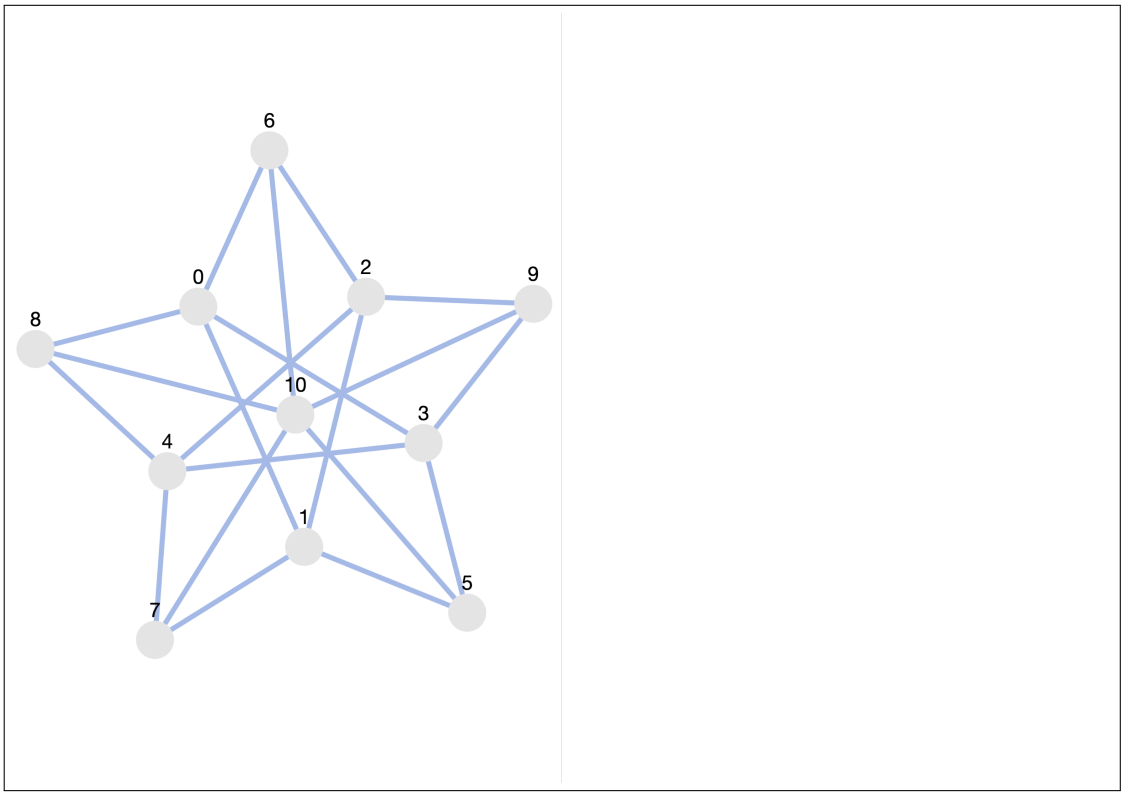

Figure 15 – Graph myciel3.col and empty backtracking tree on initial step.

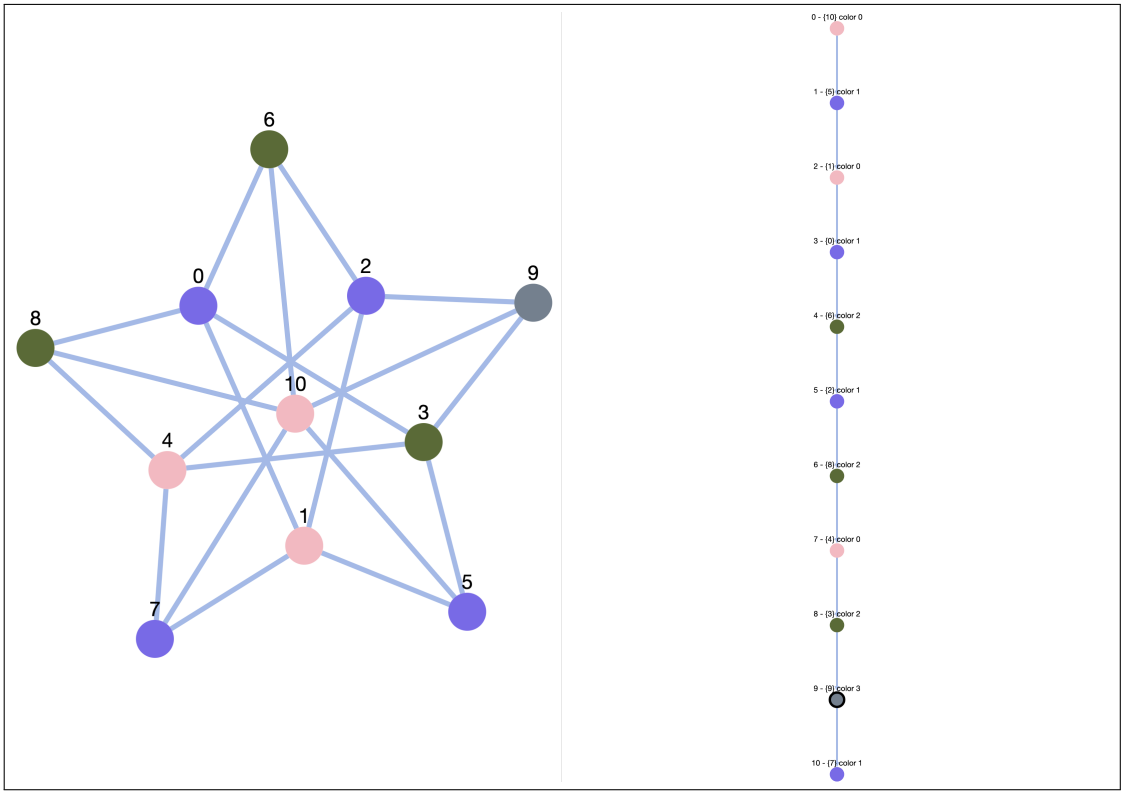

Figure 16 – Graph myciel3.col and backtracking tree on first solution found. The tree displays the colors assigned to each vertex in the solution found.

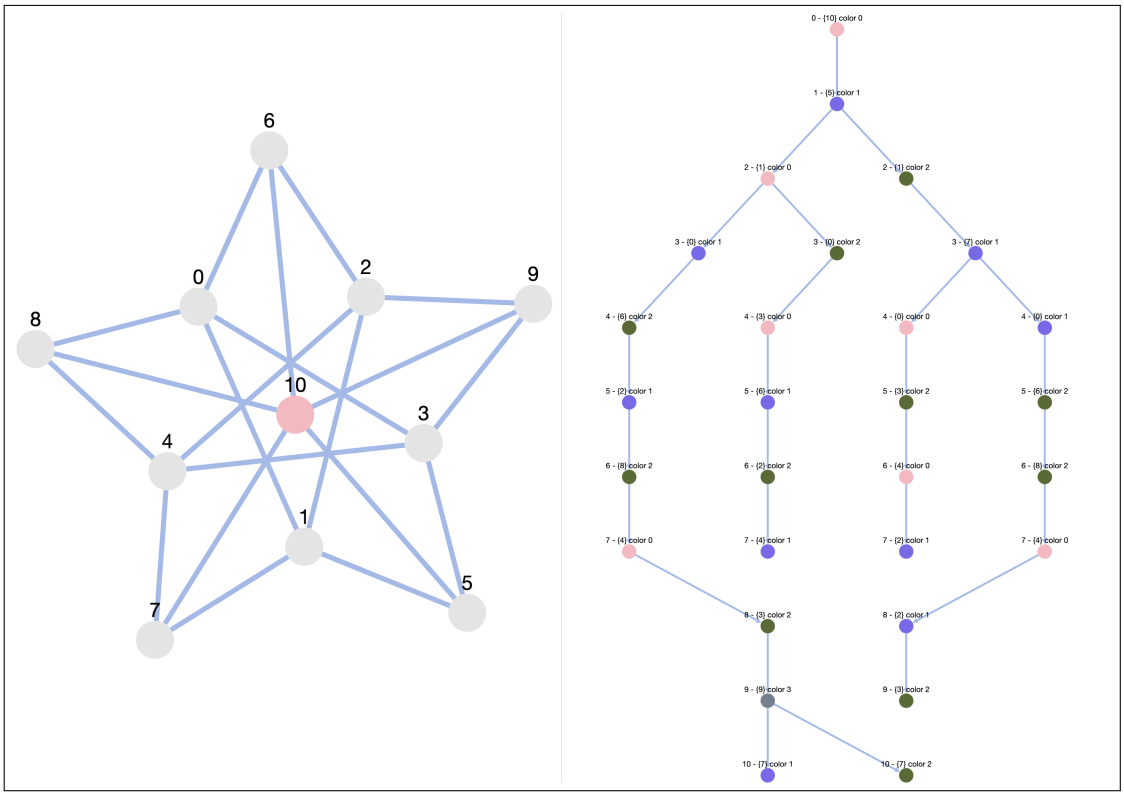

Figure 17 – Graph myciel3.col and explored backtracking tree after executing the last step of the coloring procedure. Some branches have diverged into two color choices for the same vertex.

6.1.2 Building and Running

The recommended approach for running the visualizer is to make use of its *Docker* containers. On a computer with *Docker* installed, the *visualizer* folder has to be built as a container image and initialized with *FILE* and *ALG* environment variables provided by the user. Our docker compose file internally uses *-DDEBUG* compilation during build step and *DEBUG=1* for executing algorithm implementations. The command *docker compose up* will spawn the respective containers: the desired graph coloring algorithm implementation with logging enabled, and a web server on port 8080 by default. Once the *graph-coloring* process is complete, the resulting logs will be stored on a JSON file and the visualizer user interface will become accessible through a local web interface at `http://localhost:8080/visualizer/?algorithmType={ALG}`, where the *ALG* environment variable must match one of the following values:

- dsatur-backtracking;
- dsatur-sewell;
- dsatur-pass-always;
- dsatur-pass-conditional;

- dsatur-gac-0;
- dsatur-gac-1;
- dsatur-gac-2;
- arbitrary-backtracking.

6.1.3 Limitations

Although the web visualizer interface can be helpful to understand the coloring of small to medium-sized graphs, the complexity of rendering the backtracking tree tends to grow exponentially for larger graphs, thus requiring the user to disable the backtracking tree view either temporarily or permanently. In the graph section viewer, edges and vertices also become more overlapped in denser graphs, making the identification of adjacent vertices more difficult. Finally, the *-DDEBUG* with *DEBUG=1* environment options will generate a large quantity log statements to be saved in storage, therefore drastically slowing down the execution of the algorithms prior to the visualizer's execution (see Table 2 for an example comparison). For these reasons, the web visualizer is mostly limited to a debugging and/or learning tool for graph coloring. Our test results presented next in Chapter 7 do not store any log statements (no *-DDEBUG* option) and use only the JSON data provided by the algorithm in the standard output (see Example 5).

7 Results and Discussion

In this chapter, we present the results obtained by our test suite to measure the overall performance of our developed algorithm implementations.

7.1 Test Configuration

Our experiments ran a test suite using the remote (cloud-based) mode of the test bench. The setup was composed of 58 DIMACS files from the dataset, generating a total of 580 combinations (each algorithm was tested against each graph). Instances were configured with a *TIMEOUT* value of *1h (3600s)*, and the cloud provider dynamically managed the execution of each instance based on resource availability and limits. Default cloud quotas limited the execution to a maximum of 100 simultaneous instances.

Figure 18 illustrates resource utilization in the compute environment during testing. Upward spikes in the graph represent the initialization of new instances while downward spikes represent their termination, caused either by a successful exit or a timeout exit. In the beginning of the test execution, 100 of the 580 instances were initialized simultaneously, and many heuristic instances quickly finished causing downward spikes in the utilization percentage until more complex exact instances were automatically spawned by AWS Fargate. The job continued with occasional downward larger spikes representing multiple instances often timing out during similar periods. In the final stage, the last batch of spawned instances did not need 100% resource allocation since at that point there were fewer instances remaining in the queue. The entire test job array finished after a total run time of approximately 4h.

Figure 18 – AWS Fargate Resource Count Utilization During Benchmarking (in %).

7.2 Results

Table 4 shows obtained results for non-exact algorithms with optimal solutions highlighted by green background cells. As initially expected, colorings given by the DSATUR heuristic

Table 4 – Elapsed run time (in ms*) for each heuristic, followed by $\chi(G)$, and the number of colors assigned obtained by each heuristic. Results highlighted in green are equal to $\chi(G)$.*

Graph Name	|V|	|E|	CGCH Time	DSATUR Time	Optimal Colors	CGCH Colors	DSATUR Colors
myciel3.col	11	20	0	0	4	4	4
myciel4.col	23	71	0	0	5	5	5
myciel5.col	47	236	0	77	6	6	6
myciel6.col	95	755	0	6	7	7	7
myciel7.col	191	2,360	0	100	8	8	8
anna.col	138	493	0	7	11	12	11
david.col	87	406	0	3	11	12	11
huck.col	74	301	0	83	11	11	11
homer.col	561	1,628	0	489	13	15	13
jean.col	80	254	0	79	10	10	10
games120.col	120	638	0	86	9	9	9
queen5_5.col	25	160	0	0	5	8	5
queen6_6.col	36	290	0	0	7	11	9
queen7_7.col	49	476	0	1	7	10	11
queen8_8.col	64	728	0	2	9	13	12
queen8_12.col	96	1,368	0	10	12	15	14
queen9_9.col	81	1,056	0	80	10	16	13
queen10_10.col	100	1,470	0	85	?	16	14
queen11_11.col	121	1,980	0	73	11	17	15
queen12_12.col	144	2,596	0	94	?	20	16
queen13_13.col	169	3,328	0	186	13	21	17
queen14_14.col	196	4,186	0	115	?	23	19
queen15_15.col	225	5,180	0	206	?	25	21
queen16_16.col	256	6,320	0	293	?	25	23
miles250.col	128	387	0	82	8	9	8
miles500.col	128	1,170	0	7	20	22	20
miles750.col	128	2,113	0	87	31	34	31
miles1000.col	128	3,216	0	93	42	44	42
miles1500.col	128	5,198	0	102	73	76	73
zeroin.i.1.col	211	4,100	0	120	49	49	49
zeroin.i.2.col	211	3,541	0	183	30	30	30
zeroin.i.3.col	206	3,540	0	190	30	30	30
mulsol.i.1.col	197	3,925	0	105	49	49	49
mulsol.i.2.col	188	3,885	0	106	31	31	31
mulsol.i.3.col	184	3,916	0	102	31	31	31
mulsol.i.4.col	185	3,946	0	107	31	31	31
mulsol.i.5.col	186	3,973	0	186	31	31	31
le450_5a.col	450	5,714	0	515	5	14	10
le450_5b.col	450	5,734	0	593	5	13	9
le450_15a.col	450	8,168	0	688	15	22	17
le450_15b.col	450	8,169	1	691	15	22	16
le450_25a.col	450	8,260	1	608	25	28	25
le450_25b.col	450	8,263	0	606	25	27	25
le450_5d.col	450	9,757	0	801	5	18	12
le450_5c.col	450	9,803	0	801	5	17	10
fpsol2.i.1.col	496	11,654	1	690	65	65	65
fpsol2.i.2.col	451	8,691	1	618	30	30	30
fpsol2.i.3.col	425	8,688	83	607	30	30	30
inithx.i.1.col	864	18,707	2	1911	54	54	54
inithx.i.2.col	645	13,979	1	1402	31	31	31
inithx.i.3.col	621	13,969	1	1409	31	31	31
le450_15c.col	450	16,680	1	1196	15	30	23
le450_15d.col	450	16,750	1	1192	15	31	24
le450_25c.col	450	17,343	1	1201	25	37	29
le450_25d.col	450	17,425	81	1122	25	35	28
school1.col	385	19,095	1	904	?	42	17
school1_nsh.col	352	14,612	1	698	?	39	27
latin_square_10.col	900	307,350	92	30191	?	213	132

were optimal for more instances than CGCH solutions: out of 50 instances with known chromatic numbers, DSATUR matched 33 optimal solutions while CGCH only matched 22. When the optimal coloring was not achieved by any of the heuristics, DSATUR also performed better than CGCH overall by assigning at least one fewer color to the solution, with the exception of a single graph, `queen7_7.col` (11 and 10 colors respectively). The difference in coloring efficiency of both heuristics became more evident in larger graphs, with `school1_nsh.col`, `school1.col` and `latin_square_10.col` displaying a disparity of 12, 25 and 81 colors between DSATUR and CGCH solutions, respectively.

In terms of performance, CGCH was the fastest heuristic in all instances, with its slowest execution taking 92ms for `latin_square_10.col`. On the other hand, for the DSATUR algorithm, it took 30,191ms to color the same graph. The difference in DSATUR and CGCH running times grew as the size of graphs increased, as we can especially see on `queen` instances results. Possible further improvements to the DSATUR implementation layer would impact time benchmark results for both exact and heuristic versions.

Below we compare the optimality of heuristic results separated by the type of the analyzed graphs:

- *Mycielski* graphs: Both CGCH and DSATUR found optimal solutions for all analyzed graphs (from `myciel3.col` to `myciel7.col`).
- *Book* graphs (Anna, David, Huck, Homer, Jean): Only DSATUR found optimal solutions for all graphs, while CGCH colored optimally 2 of 5 graphs.
- *Games* graphs: Both heuristics optimally colored the graph.
- *Queen* graphs: Except for `queen5_5.col` (colored optimally by DSATUR), none of the heuristics found optimal solutions for any of the 12 remaining graphs (from `queen6_6.col` to `queen16_16.col`). This implies that backtracking versions of the heuristics will not find the optimal solution initially for graphs of queen type, and thus they can be used to clarify how exact algorithms behave progressively during lower bound updates.
- *Miles* graphs: Only the DSATUR heuristic found optimal colorings for this graph type.
- *Zeroin* and *Mulsol* graphs: Both heuristics colored all graphs optimally.
- *Leighton* graphs: DSATUR only colored 2 graphs optimally (`le450_25a.col` and `le450_25b.col`), while CGCH did not color any of the graphs optimally.
- *Fpsol* and *Inithx* graphs: Both heuristics were optimal for all instances.

- *School* graphs: Although the optimal chromatic number was not provided by the dataset, we have noticed a significant difference in the number of colors for both heuristics for `school1.col` and `school1_nsh.col`, respectively 12 and 25 between CGCH and DSATUR solutions.

- *Latin Square*: This graph presented the largest difference in colorings from both heuristics (81 colors). The optimal solution was not originally given by the dataset.

7.2.1 Exact Algorithms

Table 5 displays the time taken by each test instance to finish execution and return the optimal solution. Cells highlighted by the green background help identify the fastest algorithm for each graph, while the dash symbol (–) represents a timeout exit (no solution after 1h). Table 6 contains the backtracking tree vertex count, with highlighted cells representing the smallest value obtained for the graph instance.

An analysis of the results for exact algorithms indicates that 34 of 58 graphs were optimally colored by at least one algorithm while the remaining 24 graphs terminated with timeouts. Overall, original DSATUR Backtracking had the best performance in terms of fastest solved instances compared to other algorithms on the same graph: original DSATUR was the fastest on 20 graphs, followed by both DSATUR Pass variations on 7 unique graphs each. DSATUR Sewell Backtracking obtained the fastest solution for `le450_5b.col` graph which was solved only by Sewell algorithm itself. GAC variations and Arbitrary Backtracking did not find solutions first in any of the graphs.

Performance was also measured by sorting algorithms in descending order by quantity of generated solutions for all graphs in the dataset. This is a better metric for checking how their execution handles different graph types on average cases, such as when the provided graph type is unknown. DSATUR Pass Always Backtracking was the algorithm that solved most of the instances in 1h (32 out of 58), followed by DSATUR Pass Conditional Backtracking (31), original DSATUR Backtracking and DSATUR Sewell Backtracking (25 each), DSATUR GAC-2 (17), DSATUR GAC-1 (14), DSATUR GAC-0 Backtracking (8) and Arbitrary Backtracking (6).

7.2.1.1 GAC Variations

None of the DSATUR Backtracking variations with GAC achieved good run time results. This is probably attributed to a slow formulation of CSP instances, as our algorithm had to take into account the k value (upper bound) which is dynamic and decreases over time. For larger graphs, it appears that the pruning benefit was reduced, in some instances being of no help at all. The most aggressive version of our pruning algorithm, GAC-0,

reduced the search tree size to approximately 1/3 of the size in one of the instances, but the algorithm only solved 8 of the 58 graphs in 1h.

Overall, the GAC-2 algorithm only solved the graphs that original DSATUR Backtracking also solved, but GAC-2 always added extra overhead to the execution time. The same happened with GAC-1 in relation to GAC-2 and GAC-0 with relation to GAC-1, meaning that the creation of CSP instances for performing pruning operations was too costly and never repaid any benefits.

7.2.1.2 Factors of Influence

Size, order and density of graphs are important metrics when analyzing graph coloring, but the graph structure itself can impact execution time significantly. For example, it is worth noting that `huck.col` has the same chromatic number as `anna.col` and `david.col` and all three were colored optimally by DSATUR heuristic, but `huck.col` was not colored by our exact algorithms even though `anna.col` contains more edges and vertices. In another example, `myciel6.col` has a similar number of edges and fewer vertices than `miles250.col` but it did not have an optimal coloring given in our tests while the latter did.

The results have demonstrated the Backtracking Tree size was also inversely correlated to performance in some cases, such as DSATUR Pass Backtracking improvements for *queen* and *zeroin* graph types, as we can see in more details by grouping graphs by their types.

7.2.1.3 Results by Graph Types

- *Mycielski* graphs: Arbitrary Backtracking only solved 2 of the instances, while DSATUR backtracking variations except GAC-0 solved 3 instances. Original DSATUR backtracking was the fastest algorithm for this graph type. DSATUR GAC-0 Backtracking had the smallest backtracking tree size for `myciel3.col` and `myciel4.col` instances, while DSATUR Sewell Backtracking pruned `myciel5.col` the most.

- *Book* graphs: Arbitrary Backtracking did not solve any of the instances but all DSATUR Backtracking algorithms were able to solve `anna.col` and `david.col` graphs. Original DSATUR Backtracking was the fastest for all solved graphs, and `jean.col` graph was only solved by original DSATUR Backtracking, Sewell and GAC-2 variations. GAC-2 reduced `jean.col` backtracking tree size by about 500 vertices, a small number compared to the tree order (approximately 2.2 million vertices).

- *Games* graphs: Only DSATUR Backtracking Pass Always was able to color the `games120.col` graph.

- *Queen* graphs: None of the algorithms were able to solve instances from `queen9_9.col` to `queen16_16.col`. From `queen5_5.col` to `queen8_12.col`, DSATUR Pass Always Backtracking presented a clear advantage being the algorithm which solved the most instances (5) and also the fastest in most of them (3), followed by DSATUR Pass Conditional Backtracking and original DSATUR with 4 solved instances and 1 fastest instance each. Both Pass variations (especially Pass Always) had excellent results in reducing the size of the Backtracking tree for this graph type. Surprisingly, Arbitrary Backtracking solved 4 *queen* instances, including `queen8_12.col` which was only solved again by DSATUR Pass Always Backtracking.

- *Miles* graphs: The original DSATUR Backtracking algorithm was the fastest in all instances it solved (3), but Pass Always Backtracking and Pass Conditional Backtracking both solved all instances (5) and were the fastest for 1 instance each. Pass variations also reduced the size of the backtracking tree compared to the original DSATUR Backtracking algorithm, although not as significantly and consistently as their previous achievements for *queen* graphs. DSATUR Sewell had an unusually high backtracking tree vertex count for the `miles750.col` graph relative to other algorithms.

- *Zeroin* graphs: Original DSATUR Backtracking and both Pass Backtracking variations solved all 3 instances, while DSATUR Sewell Backtracking only solved one instance. Fastest solutions presented mixed results, since `zeroin.i.1.col` was quickly colored by the original DSATUR algorithm with relation to DSATUR Pass conditional, but the latter was much faster for `zeroin.i.2.col` and `zeroin.i.3.col` graphs due to its excellent reduction in the backtracking tree size for both instances.

- *Mulsol* graphs: `mulsol.i.1.col` was not optimally colored by any algorithm, but original DSATUR backtracking presented a clear advantage for graphs of same type, solving all remaining instances and being the fastest for all of them by a significant margin. Arbitrary backtracking on the other hand timed out for all graphs. Each *mulsol* instance contained the same backtracking tree vertex count for different algorithms, suggesting that there are not enough search space optimization techniques available for this type of graph.

- *Leighton* graphs: Instances `le450_15c.col`, `le450_15d.col`, `le450_25c.col` and `le450_25d.col` were not optimally colored by any of the algorithms. For remaining instances, DSATUR Sewell Backtracking and DSATUR Pass Conditional Backtracking both solved 5 instances, and Sewell exclusively solved `le450_5b.col`. This was also the only graph in the dataset where Sewell was the fastest. Original DSATUR backtracking and DSATUR Pass Always presented the largest count of fastest solutions (2 each). Arbitrary Backtracking did not color instances of *Leighton* type.

DSATUR Pass variations had a small reduction in the number of explored vertices for the backtracking tree, and Sewell had an unusually high backtracking vertex count for the `le450_15b.col` graph.

- *Fpsol* graphs: `fpsol2.i.1.col` was not solved by any of the algorithms, and `fpsol2.i.2.col` and `fpsol2.i.3.col` were solved by all DSATUR variations except GAC ones. Original DSATUR backtracking was the fastest to obtain a solution for both graphs. Sewell had a small decrease in the backtracking vertex count while Pass variations had a significant increase for graphs of this type.

- *Inithx* graphs: Only the `inithx.i.1.col` graph was solved by DSATUR Pass Backtracking variations, with Conditional being the fastest even when both algorithms explored the same number of backtracking tree vertices.

- *School* graphs: `school1.col` graph was solved by all DSATUR Backtracking variations except GAC ones. `school1_nsh.col` graph on the hand was only solved by DSATUR Pass variations, with DSATUR Pass Conditional Backtracking being approximately 2.4 seconds faster than DSATUR Pass Always Backtracking. Although not specified in the initial dataset, the obtained chromatic numbers were 14, differing from approximate solutions obtained by initial heuristics and matching results obtained by other experiments [21]. Pass variations had excellent reductions in the number of backtracking vertices for `school1.col` compared to original DSATUR and Sewell Backtracking.

- *Latin Square* graph: All algorithms timed out when attempting to solve the *Latin Square* graph instance.

Table 5 – $\chi(G)$, and elapsed run time (in ms*) for each exact algorithm. The fastest algorithm for each graph file is highlighted in green. Timeouts after 1h are represented by dashes. The last row displays the quantity of graphs that were solved by each algorithm before timing out.*

Graph Name	χ(G)	Arbitrary Bkt Time	DSATUR Bkt Time	DSATUR Sewell Bkt Time	DSATUR Pass Always Bkt Time	DSATUR Pass Conditional Bkt Time	DSATUR GAC-0 Bkt Time	DSATUR GAC-1 Bkt Time	DSATUR GAC-2 Bkt Time
myciel3.col	4	76	0	2	1	1	15	5	2
myciel4.col	5	8296	106	304	298	294	21090	2397	216
myciel5.col	6	-	121198	191121	299905	271884	-	1699102	146712
myciel6.col	7	-	-	-	-	-	-	-	-
myciel7.col	8	-	-	-	-	-	-	-	-
anna.col	11	-	213	1701	1318	1016	31097	32900	13604
david.col	11	-	98	1117	811	790	2392	2494	2495
huck.col	11	-	-	-	-	-	-	-	-
homer.col	13	-	-	-	-	-	-	-	-
jean.col	10	-	1045891	2109206	-	-	-	-	1834187
games120.col	9	-	-	-	77610	-	-	-	-
queen5_5.col	5	1	1	183	14	90	102	280	109
queen6_6.col	7	1506	1095	4793	498	418	2430318	230802	30121
queen7_7.col	7	7513	7188	23192	591	4790	-	1416924	99717
queen8_8.col	9	-	3269886	-	794112	988191	-	-	-
queen8_12.col	12	1536801	-	-	1807	-	-	-	-
queen9_9.col	10	-	-	-	-	-	-	-	-
queen10_10.col	?	-	-	-	-	-	-	-	-
queen11_11.col	11	-	-	-	-	-	-	-	-
queen13_13.col	13	-	-	-	-	-	-	-	-
queen14_14.col	?	-	-	-	-	-	-	-	-
queen15_15.col	?	-	-	-	-	-	-	-	-
queen16_16.col	?	-	-	-	-	-	-	-	-
miles250.col	8	-	190	610	336	308	7695	1016	813
miles500.col	20	-	380	3018	1697	1510	185494	199602	78791
miles750.col	31	-	2107	752696	5988	6087	-	-	-
miles1000.col	42	-	-	-	18995	18709	-	-	-
miles1500.col	73	-	-	-	83202	84089	-	-	-
zeroin.i.1.col	49	-	1804	73501	65500	49492	-	-	-
zeroin.i.2.col	30	-	2218195	-	32909	28503	-	-	2812521

Graph Name	χ(G)	Arbitrary Bkt Time	DSATUR Bkt Time	DSATUR Sewell Bkt Time	DSATUR Pass Always Bkt Time	DSATUR Pass Conditional Bkt Time	DSATUR GAC-0 Bkt Time	DSATUR GAC-1 Bkt Time	DSATUR GAC-2 Bkt Time
zeroin.i.3.col	30	-	2215410	-	31306	29200	-	-	2742210
mulsol.i.1.col	49	-	-	-	-	-	-	-	-
mulsol.i.2.col	31	-	893	50616	28405	23789	-	428702	431998
mulsol.i.3.col	31	-	892	51314	28003	25409	-	431601	436610
mulsol.i.4.col	31	-	894	52620	28802	24099	-	449920	438003
mulsol.i.5.col	31	-	813	56293	28792	24673	-	453506	436298
queen12_12.col	?	-	-	-	-	-	-	-	-
le450_5a.col	5	-	-	-	141517	143104	-	-	-
le450_5b.col	5	-	-	246814	-	-	-	-	-
le450_15a.col	15	-	-	-	-	-	-	-	-
le450_15b.col	15	-	-	886303	-	69102	-	-	-
le450_25a.col	25	-	5793	56207	24394	23305	-	-	-
le450_25b.col	25	-	5816	44810	23499	21795	-	-	-
le450_5d.col	5	-	-	-	-	-	-	-	-
le450_5c.col	5	-	-	377606	56996	58365	-	-	-
fpsol2.i.1.col	65	-	-	-	-	-	-	-	-
fpsol2.i.2.col	30	-	34006	423793	204602	176716	-	-	-
fpsol2.i.3.col	30		32912	423713	194802	174712	-	-	-
inithx.i.1.col	54	-	-	-	2413091	1940781	-	-	-
inithx.i.2.col	31	-	-	-	-	-	-	-	-
inithx.i.3.col	31	-	-	-	-	-	-	-	-
le450_15c.col	15	-	-	-	-	-	-	-	-
le450_15d.col	15	-	-	-	-	-	-	-	-
le450_25c.col	25	-	-	-	-	-	-	-	-
le450_25d.col	25	-	-	-	-	-	-	-	-
school1.col	?	-	8695	635988	665700	634198	-	-	-
school1_nsh.col	?	-	-	-	99192	96807	-	-	-
latin_square_10.col	?	-	-	-	-	-	-	-	-
Qty. Solved (1h)	-	6	25	25	**32**	**31**	8	14	17

Table 6 – Number of backtracking vertices explored by each algorithm to find the optimal solution. For each graph, the algorithm with the lowest quantity of backtracking vertices is highlighted in green. Timeouts after 1h are represented by dashes.

Graph Name	Arbitrary Bkt Vtcs	DSATUR Bkt Vtcs	DSATUR Sewell Bkt Vtcs	DSATUR Pass Always Bkt Vtcs	DSATUR Pass Conditional Bkt Vtcs	DSATUR GAC-0 Bkt Vtcs	DSATUR GAC-1 Bkt Vtcs	DSATUR GAC-2 Bkt Vtcs
myciel3.col	80	29	24	25	25	17	29	29
myciel4.col	220476	906	614	755	755	327	892	906
myciel5.col	-	448650	181474	470021	439559	-	447611	448026
myciel6.col	-	-	-	-	-	-	-	-
myciel7.col	-	-	-	-	-	-	-	-
anna.col	-	873	873	870	871	868	871	868
david.col	-	432	432	437	439	432	432	432
huck.col	-	-	-	-	-	-	-	-
homer.col	-	-	-	-	-	-	-	-
jean.col	-	2261607	2261607	-	-	-	-	2261123
games120.col	-	-	-	55830	-	-	-	-
queen5_5.col	114	25	28	26	26	25	25	25
queen6_6.col	29325	4987	3488	580	580	1647	4942	4957
queen7_7.col	119682	22130	11490	449	5236	-	22052	24306
queen8_8.col	-	7161560	-	633809	818489	-	-	-
queen8_12.col	13395034	-	-	726	-	-	-	-
queen9_9.col	-	-	-	-	-	-	-	-
queen10_10.col	-	-	-	-	-	-	-	-
queen11_11.col	-	-	-	-	-	-	-	-
queen13_13.col	-	-	-	-	-	-	-	-
queen14_14.col	-	-	-	-	-	-	-	-
queen15_15.col	-	-	-	-	-	-	-	-
queen16_16.col	-	-	-	-	-	-	-	-
miles250.col	-	503	504	505	503	502	503	503
miles500.col	-	1171	1168	1159	1151	1166	1169	1166
miles750.col	-	2740	261094	1392	1412	-	-	-
miles1000.col	-	-	-	1375	1385	-	-	-
miles1500.col	-	-	-	1595	1593	-	-	-
zeroin.i.1.col	-	5234	5234	5104	5104	-	-	-
zeroin.i.2.col	-	904621	-	2525	2521	-	-	904621

Graph Name	Arbitrary Bkt Vtcs	DSATUR Bkt Vtcs	DSATUR Sewell Bkt Vtcs	DSATUR Pass Always Bkt Vtcs	DSATUR Pass Conditional Bkt Vtcs	DSATUR GAC-0 Bkt Vtcs	DSATUR GAC-1 Bkt Vtcs	DSATUR GAC-2 Bkt Vtcs
zeroin.i.3.col	-	904477	-	2381	2377	-	-	904477
mulsol.i.1.col	-	-	-	-	-	-	-	-
mulsol.i.2.col	-	1635	1635	1635	1635	-	1635	1635
mulsol.i.3.col	-	1486	1486	1486	1486	-	1486	1486
mulsol.i.4.col	-	1488	1488	1488	1488	-	1488	1488
mulsol.i.5.col	-	1492	1492	1492	1492	-	1492	1492
queen12_12.col	-	-	-	-	-	-	-	-
le450_5a.col	-	-	-	17320	17320	-	-	-
le450_5b.col	-	-	13795	-	-	-	-	-
le450_15a.col	-	-	-	-	-	-	-	-
le450_15b.col	-	-	93453	-	8839	-	-	-
le450_25a.col	-	3905	3909	3836	3929	-	-	-
le450_25b.col	-	3811	3817	3828	3792	-	-	-
le450_5d.col	-	-	-	-	-	-	-	-
le450_5c.col	-	-	5668	3056	3056	-	-	-
fpsol2.i.1.col	-	-	-	-	-	-	-	-
fpsol2.i.2.col	-	9196	9180	13024	13024	-	-	-
fpsol2.i.3.col	-	8445	8429	12273	12273	-	-	-
inithx.i.1.col	-	-	-	80606	80606	-	-	-
inithx.i.2.col	-	-	-	-	-	-	-	-
inithx.i.3.col	-	-	-	-	-	-	-	-
le450_15c.col	-	-	-	-	-	-	-	-
le450_15d.col	-	-	-	-	-	-	-	-
le450_25c.col	-	-	-	-	-	-	-	-
le450_25d.col	-	-	-	-	-	-	-	-
school1.col	-	2765	3082	610	610	-	-	-
school1_nsh.col	-	-	-	1102	1102	-	-	-
latin_square_10.col	-	-	-	-	-	-	-	-

7.3 Further Improvements

In this section, we discuss some further improvements which could be implemented in our work.

7.3.1 Algorithms

Our implementations for Sewell and Pass could become faster by following all suggestions from Sewell's original article [20]. It states that the initial coloring found should be an optimal coloring, otherwise the original version of DSATUR could still be faster. The author therefore suggested the use of TABUCOL (or non-exact DSATUR for geometric graphs) with RLF during the beginning of execution.

For further speed improvements on exact algorithms we suggest making heavier use of caching where possible to avoid recalculations such as saturation degree and heuristic dependent information, where upon backward movements the cached information could be restored more quickly. Other data structures such as heaps, linked lists and binary trees could be added, and measured against the existing implementation that uses arrays and sets. The current implementation contained in this work should not, however, interfere significantly with the overall results of the analysis, since all algorithm heuristics were built on the same foundation layer. This was an important requirement to compare different heuristics and reduce differences to the minimum surface possible.

7.3.2 Test Bench

The code for the Test bench feature could be abstracted into scripts to allow for testing of different programs both in local and remote environments. Our approach guarantees modularity through container-based development, and uses job array execution for tasks that are resource-intensive. Another useful addition, specifically for the graph coloring program, would be to add more metrics, such as standard deviation and average of the obtained results.

7.3.3 Web Visualizer

The web visualizer could be published on a website, and contain our developed C++ algorithms compiled with the *WebAssembly* technology, allowing for inspection and real-time execution of graph coloring instances via a simple website. It could also lazy-load log files to avoid freezing the UI when loading larger graphs.

7.4 Conclusion

In this work, we measured the performance of multiple constructive graph coloring algorithms on a set of graph files from the DIMACS challenges, and we developed a web visualizer to inspect actions performed by the studied algorithms in small graphs. We began by presenting a brief history about the origins of the graph coloring problem and its applications, and introduced the definition of the problem itself, along with some formal concepts required to understand our developed coloring algorithms.

Due to the intractability of the graph coloring problem, the study of coloring algorithms was divided into two categories: heuristics and exact algorithms. We selected the Classical Greedy Coloring Heuristic (CGCH) and the DSATUR Heuristic for the approximation algorithms. Extra concepts necessary for understanding exact algorithms were presented, such as the search space, backtracking, tight ordering, branch-and-bound, clique detection, the existence of upper and lower bounds for the chromatic number, the formulation of the problem as a constraint satisfaction problem, constraint networks and arc consistency. The exact algorithms studied in our work were: the Arbitrary Ordering Algorithm and four exact DSATUR algorithms (Original, Sewell, Pass and GAC). We suggested the creation of the GAC algorithm as a new DSATUR variant, which obtains generalized arc consistency with AC-3 in specific steps during backtracking to perform pruning of the search space.

Our graph coloring algorithms were implemented in C++, and the code was encapsulated in Docker containers to improve portability and to allow for more predictable behavior across different environments. The test bench, which runs on Node.js, supports both local and cloud modes. It was responsible for managing the execution of multiple coloring instances and for concatenating their results into a single file. We also created a web visualizer to inspect previous algorithm outputs in the browser. It was made using VueJs and CytoscapeJs, and it contains a left menu which displays information about the algorithm, the colored graph, intermediate colorings found and the action logs. It depicts the graph visualization and the backtracking tree visualization in the center and right panels, whose current state can be set by the controls in the left menu.

In order to measure the performance of the studied algorithms, we executed the test bench on AWS Batch and AWS Fargate cloud services. It generated a job array containing each combination of DIMACS graphs and studied algorithms, with a timeout of 1h for each instance. The algorithms were tested on *.col* files containing the definition of the graph to be colored. We partially verified the correctness of our algorithms by ensuring that their solutions did not contain any color clashes, and that the chromatic numbers matched those provided by the DIMACS dataset.

The cloud benchmark test took a total time of about 4h. For heuristics, we evaluated the elapsed time and the quantity of colors used by each solution; for exact algorithms,

we evaluated both the elapsed time and the backtracking tree vertex count. Our results have validated that the DSATUR heuristic is often better than the CGCH heuristic, as it obtained more optimal colorings, but the CGCH was the fastest in all tested graphs. Possible improvements for the DSATUR algorithm implementation may reduce the elapsed time difference between both heuristics. The DSATUR heuristic colored books and miles graphs optimally, but queens, leighton, school and latin square graphs were not optimally colored by any heuristic. Exact algorithms had multiple outcomes when analyzing different graph types, although this may have been related to our algorithm implementations. Approximately 59% (34 of 58) graphs were colored by at least one exact algorithm, and the DSATUR-based algorithms were often faster, with original DSATUR being the fastest one in the greatest quantity of graphs. DSATUR GAC-0,1,2 significantly reduced the quantity of backtracking vertices for some graphs, but its running time was too long. The original exact DSATUR algorithm showed good results for books and mulsol graph types. In general, DSATUR Pass showed good performance by being the exact algorithm to solve the greatest quantity of graphs in 1h, followed by DSATUR Sewell and original DSATUR tied for second place, followed by DSATUR GAC variants, and finally by the arbitrary order backtracking algorithm. For the DSATUR Pass algorithm, some correlation between the low vertex count in the backtracking tree and the elapsed run time was also noticed.

The general performance of exact algorithms for the DIMACS dataset obtained (from best to worst) was:

1. DSATUR Pass Always Backtracking
2. DSATUR Pass Conditional Backtracking
3. DSATUR Sewell Backtracking and original DSATUR Backtracking
4. DSATUR GAC-2 Backtracking
5. DSATUR GAC-1 Backtracking
6. DSATUR GAC-0 Backtracking
7. Arbitrary Backtracking.

For future works, we suggest improving our Sewell and Pass algorithm implementations to follow all modifications proposed by Sewell, and to target our algorithm tests to specific types of graphs, such as planar graphs. Obtaining more information about the current state of instances that have timed out may also be particularly useful. The code for the test bench may be reused for other projects which require tests to run on cloud environments, and the web visualizer may be refactored to support graph coloring in real-time instead of only replaying previous logs.

Bibliography

[1] Rudolf Fritsch and Gerda Fritsch. *Four-Color Theorem*. Springer, 1998. page 1.

[2] Kenneth Ownsworth May. The origin of the four-color conjecture. *Isis*, 56(3):346–348, 1965. page 1.

[3] Carlos Laercio Gomes de Lima. *Um estudo sobre teoria dos grafos e o teorema das quatro cores*. Master's dissertation, Universidade de Sao Paulo, 2016. page 1.

[4] Kenneth Ira Appel and Wolfgang Haken. *Every planar map is four colorable*, volume 98. American Mathematical Soc., 1989. pages 1 and 3.

[5] Mate Glaurdić, Jelena Beban-Brkić, and Dražen Tutić. Graph colouring and its application within cartography. *KoG*, 20(20):99–114, 2016. page 1.

[6] Shamim Ahmed. Applications of graph coloring in modern computer science. *International Journal of Computer and Information Technology*, 3(2):1–7, 2012. page 1.

[7] Michael Paleczny, Christopher Vick, and Cliff Click. The java {HotSpot™} server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001. page 1.

[8] Sevin Shamizi and Shahriar Lotfi. Register allocation via graph coloring using an evolutionary algorithm. In *International Conference on Swarm, Evolutionary, and Memetic Computing*, pages 1–8. Springer, 2011. page 1.

[9] Armen Asratian, Tristan Mark Joseph Denley, and Roland Häggkvist. *Bipartite Graphs and Their Applications*. Cambridge Tracts in Mathematics. Cambridge University Press, 1998. page 2.

[10] William John Cook, William Cunningham, William Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 2011. page 2.

[11] Alane de Lima and Renato Carmo. Exact algorithms for the graph coloring problem. *Revista de Informática Teórica e Aplicada*, 25(4):57–73, 2018. pages 2, 11, and 19.

[12] Stephen Arthur Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. pages 2 and 10.

[13] Rhyd Lewis. *A guide to graph colouring: algorithms and applications*. Springer, Cham, 2016. pages 2, 3, 10, 11, 12, 14, 15, 18, 20, and 24.

[14] Christian Bessiere. Chapter 3 - constraint propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 29–83. Elsevier, 2006. page 3.

[15] Ian Philip Gent, Christopher Jefferson, Steve Linton, Ian Miguel, and Peter Nightingale. Generating custom propagators for arbitrary constraints. *Artificial Intelligence*, 211:1–33, 2014. pages 3 and 21.

[16] Pablo San Segundo. A new dsatur-based algorithm for exact vertex coloring. *Comput. Oper. Res.*, 39(7):1724–1733, jul 2012. pages 3, 16, 20, 21, and 28.

[17] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph Theory with Applications*. Elsevier, New York, 1976. page 10.

[18] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979. page 14.

[19] Donald Kreher and Douglas Robert Stinson. Combinatorial algorithms: Generation, enumeration, and search. *SIGACT News*, 30(1):33–35, mar 1999. page 16.

[20] David Johnson and Michael Alan Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, USA, 1996. pages 20 and 57.

[21] Emmanuel Hébrard and George Katsirelos. Constraint and satisfiability reasoning for graph coloring. *Journal of Artificial Intelligence Research*, 69:33–65, 2020. page 52.