Graph coloring

Group 1:
Jonathan Geurts, Arsenijs Hutornovs, Konduru Hemachandra,
Bas Laarakker, Alisa Todorova, Kelso Wilkin

Maastricht University
Department of Data Science and Knowledge Engineering
Project 1.1

Abstract

The Graph Coloring Problem (GCP) deals with an assignment of labels (most commonly called colors) to a given graph's elements - edges or vertices (also referred to as nodes), such that no adjacent edges or vertices can have the same color. Here we focus on the assignment of colors to vertices. The GCP's goal is to minimize the number of colors used to color the complete graph. The optimal (minimal) number of colors is called the chromatic number of the graph. The main goal of this project is to find the chromatic number of simple and complex, seen (from already given test suite) and unseen (given during examination) graphs. However, the GCP is a NP-hard problem, meaning that there is no known algorithm to compute the chromatic number of all graphs in reasonable time. That is why this paper examines 4 different GCP algorithms: a version of the DSatur Backtracking algorithm, a version of the MCQ algorithm, an improved Recursive-Largest-First (RLF) algorithm and the Welsh-Powell (WP) algorithm, in order to further examine which of those algorithms produce the tightest bounds on the chromatic number, or perhaps the chromatic number itself, as well as whether the results of separate algorithms can be combined to produce more meaningful results.

The mentioned algorithms are implemented in Java, and are tested on graphs of varying sizes. The results make clear that the improved RLF algorithm with more advanced heuristics outperforms the simple RLF and the WP algorithms by a large margin, coming at the cost of a higher time complexity. The DSatur Backtracking algorithm is often able to improve on these bounds and provide exact results for smaller graphs, but rarely even comes close to the result of the improved RLF algorithm on larger graphs. Finally, the MCQ algorithm provides a lower bound close to the chromatic number for small graphs, but its results become insignificant on larger graphs. We conclude that a combination of algorithms producing upper and lower bounds, and perhaps even exact results, works well for smaller graphs. For larger graphs, it is more valuable to focus only on a state-of-the-art upper bound approximation algorithm, and disregard the possibility of getting an exact result.

Keywords: Graph coloring, Lower bound, Upper bound, Chromatic number, Heuristics, DSatur algorithm, Backtracking algorithm, MCQ, Maximum Clique Problem, Recursive-Largest-First algorithm, RLF algorithm, Welsh-Powell algorithm, Graphs, Bipartite, Adjacency Matrix, Planar Graphs

Table of Contents

1. Introduction

The Graph Coloring Problem (GCP) is a famous NP-hard problem in computer science. Originally, a similar problem was proposed as the Four Color Problem in 1852 by the mathematician Francis Guthrie, who noted that four colors were sufficient to color the map of the countries of England such that no regions that share a common border, have the same coloring (Aslan, & Baykan, 2018). In essence, the graph coloring problem deals with an assignment of labels (most commonly called colors) to a given graph's elements - edges or vertices (also referred to as nodes), such that no adjacent edges or vertices can have the same color. For this project, and hence, in this paper, we are focusing on coloring the vertices of the graph, which is simply referred to as vertex coloring. The optimal (minimum) number of colors is called the chromatic number of the graph.

The complexity of a graph mostly increases when the number of nodes and edges increase. This then leads to an increase of its chromatic number and thus, the computational effort of solving a graph. Speed and accuracy of solutions are important for the real-world applications of GCP, such as register allocation, circuit testing, scheduling, exam timetabling, map coloring, pattern matching, designing seating plans and solving Sudoku puzzles (Musliu, & Schwengerer, 2013). That is why this paper examines which algorithms produce the tightest bounds on the chromatic number, or perhaps the chromatic number itself, as well as whether the results of separate algorithms can be combined to produce more meaningful results.

The Graph Coloring Problem is an NP-hard problem, and as such there is no known algorithm which can consistently find the chromatic number of a random graph in reasonable time. However, a lot of research has been done on approximation algorithms to find reasonable bounds for the chromatic number. This research started off mostly with algorithms using heuristics for the ordering of the vertices to be colored. Common examples of these algorithms are the RLF algorithm (Adegbindin et al., 2014), the DSatur algorithm (Brélaz, 1979) and the Welsh-Powell algorithm (Welsh, & Powell, 1967). In recent years, interest has shifted towards algorithms based on meta-heuristics, and a lot of the current state-of-the-art algorithms are based on a combination of these meta-heuristics. Some of these high-performance algorithms for GCP are the TabuCol algorithm, the PartialCol algorithm, the Hybrid Evolutionary Algorithm (HEA), the AntCol algorithm, the Hill-Climbing (HC) algorithm, and the Backtracking DSatur algorithm (Lewis, 2016). However, there remains a trade-off with these high-performance algorithms in terms of their execution time compared to the simpler algorithms. This should be taken into account when measuring the performance of algorithms (Aslan, & Baykan, 2018). Moreover, the benefit of these algorithms is often only noticeable on larger graphs. That is why both those algorithms based on heuristics and those based on meta-heuristics were considered for this project.

Four GCP algorithms have been implemented in this project. They are the following: a version of the Backtracking DSatur algorithm - for possibly providing an exact solution; both a simple and an improved version of the Recursive-Largest-First (RLF) algorithm and the Welsh-Powell (WP) algorithm - both for providing upper bounds; and a version of MCQ for solving the maximal clique problem (MCP) - for providing a lower bound. Two separate upper bound algorithms were chosen as it provided an interesting comparison measure, and their runtime is nearly negligible when compared to the other algorithms. Then, the backtracking DSatur algorithm was selected as it improves significantly on the simple backtracking approach.

Furthermore, although the problem is considered NP-hard, it is worthwhile to have an algorithm for providing exact results, especially in the case of smaller graphs. Finally, the MCQ algorithm was selected to solve the maximal clique problem: an NP-complete problem often used to provide lower bounds on the chromatic number. It was selected due to the fact that it produces far better results than its simpler counterparts while not being overly complex.

The approach taken in this experiment involves first calculating a lower bound using the MCQ algorithm and calculating separate upper bounds with RLF and WP algorithms. The best of the upper bounds k is chosen as a starting point for the backtracking DSatur algorithm, which will try to solve the graph for $k-1$ colors and continuously solve it for less colors. If it reaches the lower bound or is not able to solve it for any k, the chromatic number has been found. The novelty factor lies in the fact that multiple upper bound algorithms are used to significantly limit the search space of the exact algorithm later in the process. Furthermore, the performance of the advanced version of the RLF algorithm has been improved significantly, based on new heuristics proposed in 2014 (Adegbindin et al., 2014). These change the results drastically compared to the basic RLF algorithm, especially on larger graphs.

For this project the algorithms were implemented into a single-player game with a Graphical User Interface (GUI). It was created with the JavaFX library as it has been proven to be more versatile for modern applications. The game consists of 3 game modes: To the Bitter End, Best Upper Bound, and Random Order, and the main goal is to use as few colors as possible to color the graph. Before playing the modes, the user has to configure the graph either by manually inputting the number of nodes and vertices, by reading in a graph from file (by writing out the file path), or by choosing the graph to be randomly generated by the game itself. In each mode there is a hint button, which gives the user some computer-assisted help when pressed. Furthermore, the computer warns the player if their current coloring is not valid, i.e. two adjacent vertices have the same coloring.

To examine and discuss all aspects of this project, this paper is divided into seven main sections. First is this introduction of the project and paper, which introduces the Graph Coloring Problem, different GCP algorithms, the problem statement of the project and thus, the focus of this paper. Then, in Methods, the notation used throughout this paper is given. Furthermore, this section describes in detail the implemented algorithms developed for this project. The data sets used for the experiments are also explained there. In Implementation, the basic structure of the implementation is presented through UML diagrams. There is also a user guide of the game with GUI. In Experiments, the conducted experiments are discussed. In Results all results from the experiments are presented, which are then interpreted and explained in the Discussion section. In Conclusion a summary of this project and paper is provided.

2. Methods

This section includes a subsection on the notation used throughout this paper, and subsections discussing the implemented algorithms by explaining the underlying theory and exploring how they might produce better results than their simpler counterparts. These explanations are supported by pseudocode. The improvement in most algorithms comes from ordering vertices in beneficial ways.

2.1 Notation

In this paper, and specifically this section, a standard notation is used in most cases. A graph is represented as $G(V, E)$, where G is the graph with a set of vertices V and a set of edges E. Furthermore, the number of vertices in a graph is often referred to simply as n. A complete graph is referred to as K_n, where n is again the number of vertices. For a vertex v, a vertex w is adjacent to v if there is an edge from v to w. For a vertex v, the set of vertices adjacent to v is the neighbouring set of v, often denoted as *neighbours(v)* or $\Gamma(v)$. The degree of a vertex v is equivalent to the size of *neighbours(v)*, and is referred to as *deg(v)* or *degree(v)*. The number of colors used to color a graph is often called k, and so a k-coloring of a graph means the graph is colored with k colors. Lastly, the saturation degree of v, explained in 2.2.3, is denoted as *sat(v)*. Figure 1 and 2 show examples of graphs.

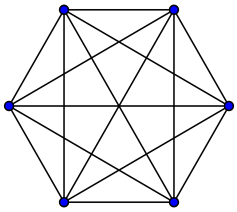


Figure 1: Graph K_6

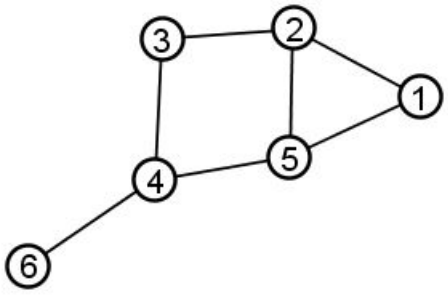


Figure 2: Graph with deg(1) = 3, deg(3) = 2

2.2 Algorithms

2.2.1 Lower Bound

For the lower bound of a graph G, the key observation to be made is that it may contain a complete subgraph K_k, also called a clique of G. A complete graph of size k will need exactly k colors to be properly colored. Therefore, a graph G containing subgraph K_k will need at least k colors for a proper coloring. From this fact we may induce that if we are able to find the largest

clique in a graph, it will often provide a reasonable lower bound on the chromatic number. Finding the largest clique in a graph is a well-known problem in computing known simply as the "Maximum clique problem" (MCP).

Here, an algorithm very similar to Tomita's MCQ algorithm is implemented. It is an extension of a simpler algorithm that will be explained first (Tomita, & Kameda, 2007). In the algorithm, we keep track of two sets: set *C*, initially empty, containing the vertices in the clique currently being created, and randomly ordered set *P* containing vertices that are possible candidates to be added to C, initially all vertices. A vertex *v* in *P* is selected as the first vertex to be added to *C*. *P* is then updated by removing all the non-neighbours of *v.* So the updated set $P' = P \cap \{neighbours(v)\}$. Then, a new vertex *v'* is chosen and the process is repeated. This ensures that all vertices in set P will be adjacent to all vertices in set C, which ensures that it remains a complete subgraph. Once P is empty, we have found the maximal (local maximum) clique. If the clique found is larger than the current largest clique, we store this as the new largest clique. We then repeat this process with a different vertex chosen as starting *v*. To limit the search space of the algorithm, we abandon the current search if the combined size of C and P is smaller than the current largest clique. This makes sense because the growing clique will never exceed the current largest.

The main improvement to be made is in the ordering of set P to limit the search space. By finding a feasible coloring for the vertices in set P, the vertices are split into separate color classes. This means that, in essence, a greedy coloring algorithm on the vertices is run first, to then order them by their assigned color. The advantage comes from the fact that by taking a vertex *i* from *P*, it is clear that all entries in *P* till *i* can be colored with *c* colors, where *c* is the number assigned to vertex *i* as a color. This means that the subgraph of these entries can have a clique of size no more than *c* (Tomita, & Kameda, 2007). If this size combined with the growing clique cannot beat the "champion" clique, the search can be abandoned. This limits the search space significantly, as unproductive searches may be abandoned much earlier than with a random ordering of *P*.

```
Algorithm 1: Expand clique
  Result: Maximal Clique
  set of vertices in clique C (empty initially;
  set of candidate vertices P (all vertices initially);
  maxClique ← 0;
  sortByColor(P);
  for i = 0; i < P.size(); i++ do
      if P.size() + color(i) < maxClique then
          return;
      end
      v ← P.get(i);
      P.add(v);
      P ← neighbours(v);
      if P.isEmpty() then
          if C.size() maxClique then
              maxClique ← C
          end
      end
      if !P.isEmpty() then
          ExpandClique(C, P)
      end
      P.remove(i);
      C.remove(i);
  end
  return maxClique;
```

Figure 1: Pseudocode of the Maximal Clique Problem (MCP) algorithm

2.2.2 Upper bound

The algorithms implemented to compute a reasonable upper bound for the coloring of a graph are both improved versions of the greedy algorithm. Greedy coloring uses a set of vertices $V = \{v_1, v_2, ..., v_n\}$ and a set of color classes $C = \{c_1, c_2, ..., c_k\}$, to which the vertices may be assigned. A vertex is chosen to be the first vertex and is assigned to the first color class. Then, all the other vertices are assigned to the first color class where that assignment does not cause a clash. A clash occurs when two vertices in the same color class are adjacent. The improvement with both of these algorithms comes from their ordering of vertices.

The first of the algorithms is often called the Welsh-Powell (WP) algorithm, after the authors of the original paper (Welsh, & Powell, 1967). It involves ordering vertices by their degree and keeping track of two sets, V, which includes all uncolored vertices at the start, and V', which is initially empty. The vertices are ordered by degree, where the vertex with the largest degree has the highest rank. The vertex *v* with the highest degree is selected to be colored, and is assigned to the lowest color class possible, which is the first color class at the start. The color to which vertices are being assigned in this iteration is the active color, *c*. The vertices non-adjacent to *v* are added to V', which becomes the set of candidates for the next selection. The vertex in V' with the highest degree is colored next with color *c*, and all vertices adjacent to it are removed from V'. This process is repeated until V' is empty, as there are no candidate vertices left. If there are uncolored vertices left, these are added to V again and the next iteration starts with the next color *c+1* as the active color, until there are no uncolored vertices left. The benefit of ordering the vertices this way is that the most constrained vertices, the ones with the most neighbours, are colored first. This leaves the less constrained vertices for later, as they are often easier to color.

```
Algorithm 1: Welsh-Powell
  Result: k-coloring
  Set of vertices V = {v1, v2, ..., vn};
  set of colours C = {c1, c2, ..., cn};
  Set V' = ∅;
  active color c = 1;
  while V ≠ ∅ do
      choose v ∈ V with highest deg(v);
      color v with active color;
      V' ← neighbours(v);
      while V' ≠ ∅ do
          choose v' ∈ V' with highest deg(v');
          color v' with active color;
          V' ← neighbours(v');
      end
      active color++;
  end
```

Figure 1: Pseudocode of the Welsh-Powell (WP) algorithm

The second algorithm is the Recursive-Largest-First (RLF) algorithm. Like the WP algorithm, it optimizes the greedy algorithm by ordering the vertices in a specific way. However, unlike the WP algorithm, the RLF algorithm also dynamically reorders the vertices throughout the iterations of the algorithm. The initial ordering of vertices is determined by the degrees of the vertices, where the vertex with the largest degree is ranked first. Thereafter, the goal is to

construct color classes iteratively, starting with the first color class for the first color. This is achieved by keeping track of three sets: set U, the set of uncolored vertices; set C_v, the set of vertices being assigned to color class C with first vertex v; and lastly set W, the set of uncolored vertices with a neighbour vertex in C_v. For each vertex $v \in U$, $A_U(v)$ and $A_W(v)$ are the number of neighbours it has in set U and W. Set U initially contains all vertices, whereas C_v and W are empty (Adegbindin et al., 2014).

The highest ranked vertex is chosen as vertex v and is the first vertex in the new color class C_v. For the first iteration, the highest ranked vertex is simply the vertex with the highest degree. The further construction of the color class works as follows: W is initialized as the set of vertices adjacent to v, and v is moved from U to C_v. All vertices adjacent to v are moved from U to W. The next vertex $u \in U$ to be colored is the vertex with the highest value $A_W(u)$, with ties being broken by selecting the vertex with lowest value $A_U(v)$. Once again, u is moved from U to C_v, and all vertices adjacent to v are also moved from U to W. This process is repeated until set U is empty. At this point, the sets are reset and the next color class is constructed. The first vertex for constructing the next color class should be the vertex with the highest value $A_U(v)$. The construction of new color classes is continued until all vertices have been colored.

The improved heuristic proposed in 2014 and implemented in the second version of the RLF algorithm introduces a different way of selecting the first vertex for a color class (Adegbindin et al., 2014). Instead of creating a single set C_v, we construct a set C_v for a constant number M starting vertices v, having the highest value $A_U(v)$. The set C_v that leaves a residual graph of uncolored vertices with the least number of edges is then chosen to be the definite color class C_v. The constant M can be decided on based on how the performance is measured. A larger M will might produce better upper bounds as more options are considered, but increases time complexity. Here, we let $M = p \cdot n$, where n is the number of nodes and $0 < p < 1$. The number p may then be varied based on the results of experiments. The algorithm can be noted as RLF-p, where p is the assigned value of p.

Algorithm 2: Advanced RLF algorithm

Result: k-coloring
Set of uncolored vertices U = $\{v_1, v_2, ..., v_n\}$;
$A_U(v)$: the number of neighbours ∈ U of v ;
$A_W(v)$: the number of neighbours ∈ W of v ;
graph G;
k = 0;
$0 < p < 1$;
while *G has uncolored vertices* **do**
  k ← k+1;
  select $p \cdot n$ vertices v ∈ U with highest $A_U(v)$;
  construct color class C_{vi} for each vertex v_i;
  choose the color class C_v leaving
    the least residual edges in remaining subgraph G'
end

Algorithm 3: Construction of color class

Result: color class C_v
Set W = ∅;
Set of uncolored vertices U = $\{u_1, u_2, ..., u_n\}$;
root vertex v;
move v from U to C_v and move neighbours(u) ∈ U to W;
while $U \neq \emptyset$ **do**
  $W \leftarrow W + neighbours(v)$;
  Select $p \cdot n$ vertex $u \in$ U with highest $A_W(v)$;
  Move u from U to C_v and move neighbours(u) ∈ U to W
end

Figure 2: Pseudocodes of the advanced Recursive-Largest-First (RLF) algorithm

2.2.3 Exact algorithm

As we have already established, the problem of deciding whether a graph is *k*-colorable is NP-complete; whereas computing the chromatic number $\chi(G)$ of a graph, is considered NP-hard. Therefore, algorithms guaranteeing exact results are often disregarded, as excess time is not available to many. However, by ordering and reordering uncolored vertices, we may see significant improvements in time complexity.

The most researched algorithms for computing the exact chromatic number are often variants of the basic backtracking algorithm, which is guaranteed to find the optimal result given infinite time. A simple backtracking algorithm for graph coloring involves setting a maximum number *k* of colors to be used. The vertices are then ordered, either randomly or by some heuristic, and are assigned the first possible color, from *1* to *k,* without causing a clash. When a vertex *v* cannot be assigned any of the *k* colors without introducing a clash, the algorithm backtracks to the last vertex *v-1* to assign it the next possible color, and then retries to assign *v* a color. The algorithm may backtrack multiple steps to if it cannot assign *v* a color by changing the color of *v-1*. Once a proper *k*-coloring has been found, the algorithm may be run for *k-1* colors. If the algorithm can find a solution for *k* colors, but not for *k-1* colors, meaning it has traversed the total search space and not found a coloring, the chromatic number must be *k* (Lewis, 2016).

The algorithm implemented here was originally proposed in 1979 by Korman and is a backtracking version of the DSatur algorithm. The DSatur algorithm (meaning "degree of saturation") was originally proposed by the mathematician Daniel Brélaz in 1979. It involves ordering the vertices by increasing saturation degree based on the current partial coloring of the graph. The saturation degree of a vertex is the number of different colors assigned to its neighbouring vertices. More formally, $sat(v) = |\{c(u) : u \in \Gamma(v) \wedge c(u) \neq 0\}|$, where $c(u)$ is the color of vertex *u* and $\Gamma(v)$ is the set of neighbours of vertex *v* (Korman, 1979). When the next vertex is colored, the ordering of vertices by saturation degree is recalculated based on the new partial coloring. When two vertices have the same saturation degree, the degree of the vertices is used to break the tie, so the vertex with the largest degree comes first.

For the backtracking version of DSatur, the initial ordering of the vertices is determined by the DSatur algorithm. The vertices are then colored by the backtracking algorithm, with one meaningful change. Whenever the algorithm performs a backtrack, the vertices are reordered again based on their saturation degree. This ensures a more valid ordering based on the current state of the graph.

Algorithm 1: DSatur Backtracking

```
Result: Boolean of coloring
maxColor ← k;
set of vertices V = {v_1, v_2, ..., v_n};
set of colors C = {c_1, c_2, ..., c_k};
c = 1;
v ← maxSat(V);
while c <= k do
    if validColor (v, c) then
        updateSat();
        v' ← maxSat(V);
        if color(v') then
            return true;
        end
    end
    color of v ← 0;
    updateSat();
    c++;
end
return false;
```

Figure 1: Pseudocodes of the DSatur algorithm

2.2.4 Special Cases

A complete graph is a graph that has an edge between any two vertices, so it has the largest number of edges possible, disregarding self loops. A complete graph K_n with n vertices has a set of edges E with size $|E| = \frac{n(n-1)}{2}$ (Lewis, 2016). It is clear that every vertex in a complete graph must be assigned its own color, as it is adjacent to every other vertex. Therefore, the chromatic number of K_n must simply be n. A method to test if a graph is complete is to simply check if the number of edges is equal to $\frac{n(n-1)}{2}$.

The detection of a bipartite graph is the same as the problem of 2-colorability, as every 2-colorable graph is bipartite. The problem of 2-colorability is a problem that may be solved in polynomial time, making it worthwhile to test for this type of graph. In fact, the algorithm for detecting bipartite graphs is rather simple. A node is chosen randomly to be the root node and colored with color A. Using breadth-first search, all neighbours of the current node are colored with the color not assigned to the current node. Then, these newly colored vertices are added to a queue. For all these vertices, the process is repeated. If this fails at any point, meaning that a node being colored has already been assigned the color of its neighbour, the graph is not bipartite.

Cycle graphs, often noted as C_n are a type of graph that consist of a single cycle. In other words, graph C_n consists of set of edges $V = \{v_1, v_2, ..., v_n\}$ and set of vertices $\{(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n), (v_n, v_1)\}$. Cycle graphs can be separated into even cycles ($n = even$) and odd cycles ($n = odd$). Any even cycle is bipartite, and can thus be colored with two colors. These cycles will already be detected when testing whether a graph is bipartite. Any odd cycle, however, can be colored with a minimum of three colors. Instead of manually detecting odd cycle graphs, the fact that the implemented RLF algorithm always produces an exact result for odd cycle graphs is used, a theorem shown by Lewis (Lewis, 2016, Theorem 2.2.1). This ensures we do not need to separately detect cycle graphs, regardless of whether they are even or odd.

Algorithm 1: Bipartite detection

```
Result: Boolean of bipartiteness
root ← v';
set of vertices V = {v_1, v_2, ..., v_n};
empty queue q;
enqueue(root);
color(root) ← 0;
while queue ! empty do
    v ← queue.pop();
    W ← neighbours(v);
    for w ∈ W do
        if color(w) = color(v) then
            return false;
        end
        if color(w) = NIL then
            color(w) = 1 − color(v);
            enqueue(w);
        end
    end
end
return true;
```

Figure 1: Pseudocode of bipartite graph detection

2.3 Data sets

The data sets used in this experiment include the graphs made available for phase 1 and phase 3. These graphs vary significantly in their size, ranging from under 100 vertices to the largest having 4007 vertices, although most graphs have a vertex count in the range of 100 to 500. The density of these graphs varies as well, although in all but a few cases, the density is considerably lower than the density of the standard DIMACS graphs often used to test graph coloring algorithms.

To provide more variability, a number of standard DIMACS graphs were also used to test the implemented algorithms. As these graphs were selected to represent both many different sizes and densities, and different types of graphs, they provide a suitable test suite for analyzing the performance of these algorithms. These graphs are far larger and more dense than the phase 3 graphs.

3. Implementation

In this section, the implementation of the separate classes is briefly explained and supported by UML diagrams. For the implementation in Java of the main class bringing these separate classes together, see **Appendix B: Main class of implementation with JavaDoc**.

3.1 Representation

For the algorithms implemented in this project, it is often beneficial to represent the graph in different ways depending on what was important: accessing elements, searching for elements or updating elements. The graphs are given as an array of edges, but the main way the graphs are represented for all algorithms is as an adjacency matrix G[][], where G[v][u] = 1 if the vertices *v* and *u* are adjacent. Therefore, a separate method creates an adjacency matrix from an array of edges. Furthermore, an adjacency list, where element G[v] is a list of vertices adjacent to *v* is used for the DSatur backtracking algorithm and the RLF algorithm. Besides this, more information about the graph was stored while reading in a graph from a file. An array containing the degree of each vertex was created, being useful for all algorithms. The number of nodes, the vertex with the highest degree and the highest degree are also stored in a class called ReadData.

3.2 Algorithms

3.2.1 Lower bound

For the lower bound algorithm, there are two parameters to create an instance of the class. These are, respectively, the graph represented as an adjacency matrix and an array of the degrees of each vertex. Besides these parameters, there are no parameters that may be varied to

alter the performance. However, the allowed time of calculation may be changed inside the class based on what aspect of performance is important. The main method of this class is the search() method, requiring no parameters, as the important parameters were given to the constructor.

MCP
- colorClasses: ArrayList<Integer> - cpuTime: long - degrees: int[] - graph: int[][] - maxSize: int - n: int
+ MCP (graph: int[][], degrees: int[]) + allowedcolor(v: int, cc: ArrayList<Integer>): boolean + build (C: ArrayList<Integer>, P: ArrayList<Integer>): void + findOrdering(P: ArrayList<Integer>): void + search(): int + sortBycolor(C: ArrayList<Integer>, Ord: ArrayList<Integer>, P: ArrayList<Integer>, color: int[]): void

Diagram 1: A UML diagram of our version of MCQ for solving the maximal clique problem (MCP) and for providing a lower bound.

3.2.2 Upper bound

For the improved RLF-*p* algorithm, there are three parameters to create an instance of the class. These are, respectively, the graph represented as an adjacency matrix, an array of the degrees of each vertex and lastly an array of edges. The array of degrees is used for initial ordering, while the array of edges is used for creating an adjacency list. When solving a graph using the algorithm, the parameter *P*, corresponding to the percentage of vertices for which a trial class is created, must be specified as a parameter in the solve() method. This parameter impacts performance significantly.

RLF
+ colors: int[] + Cv: HashSet<Integer> + degreesTot: int[] + degreesU: int[] + degreesW: int[] + edges: ColEdge[] + graph: int[][] + n: int + neighboursSet: HashMap<Integer, HashSet<Integer>> + U: HashSet<Integer> + W: HashSet<Integer>

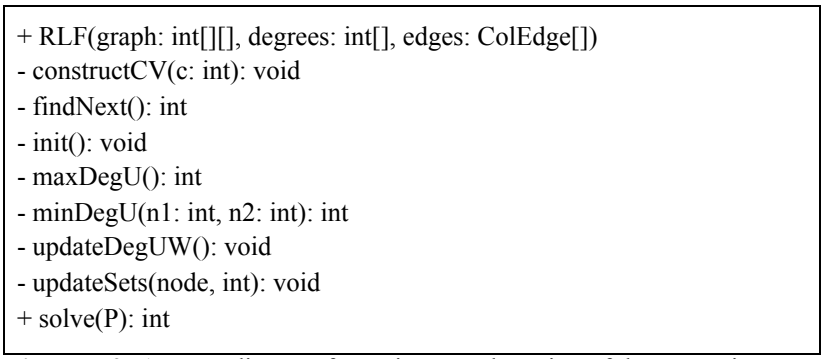


Diagram 2: A UML diagram for an improved version of the Recursive-Largest-First (RLF) algorithm for providing upper bound.

For the Welsh-Powell algorithm, there are two parameters to create an instance of the class. These are, respectively, the graph represented as an adjacency matrix and an array of the degrees of each vertex. The array of degrees is used for the dynamic ordering of the vertices, while the array of edges is used for creating an adjacency list. When solving a graph using the algorithm, the solve() method is run, which requires no further parameters. Therefore, there are no varying parameters that may impact the performance of the algorithm.

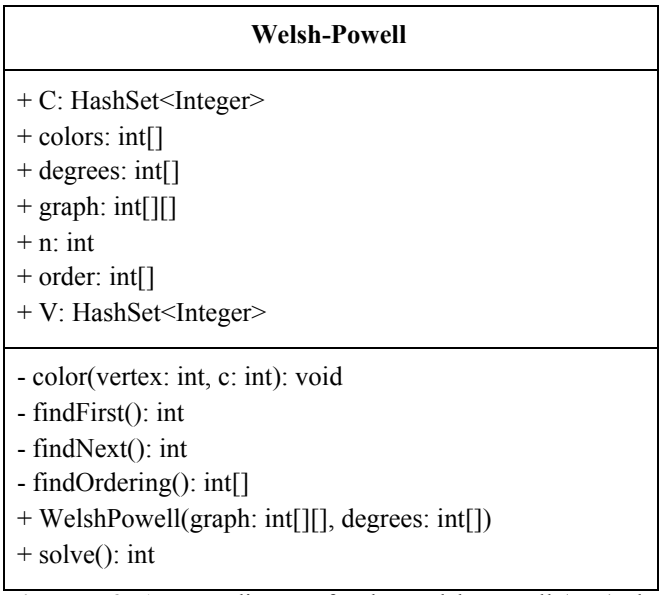


Diagram 3: A UML diagram for the Welsh-Powell (WP) algorithm for providing upper bound.

3.2.3 Exact algorithm

For the DSatur backtracking algorithm, there are two parameters to create an instance of the class. These are, respectively, the graph represented as an adjacency matrix and an array of the degrees of each vertex. The array of degrees is used for initial ordering. When solving a graph using this algorithm, the solve() method is run with the maximum number of colors available as a parameter. This parameter impacts the performance of the algorithm, as it is far easier to find a solution with more colors than when using less colors. In practice, of course, the

algorithm is run for *k* colors to see if it is solvable. If it is, the algorithm is run for *k-1* colors, and so on, until no solution can be found.

DSaturBacktrack
+ adjColors: ArrayList<Integer> + colorsArray: int[] + coloured: int + degrees: int[] + finalColoring: int[] + G: int[][] + GList: ArrayList<Integer> + max_color: int + n: int + saturation: int[]
- assignColor(currV: int): boolean - getHighestRankVertex(): int - updateSat(v: int, c: int): void - updateSat(v: int, originalC: int, remove: int): void + DSaturBacktrack (G: int[][], degrees: int[]) + solve(max: int): boolean

Diagram 4: A UML diagram for a version of the Backtracking DSatur algorithm for providing an exact solution.

3.2.4 Special cases

The SpecialGraphs class consists of two methods: one testing for bipartiteness, the other for completeness. Creating an instance of the class requires no parameters. Running the test for a complete graph requires the number of nodes and number of edges as parameters. Running the test for a bipartite graph requires two parameters, namely the graph represented as an adjacency matrix and a root vertex from which to start searching. The root vertex may be set to the vertex with the highest degree, which can be obtained from the ReadData instance. It can also simply be set to the first vertex, as the impact on performance is minimal.

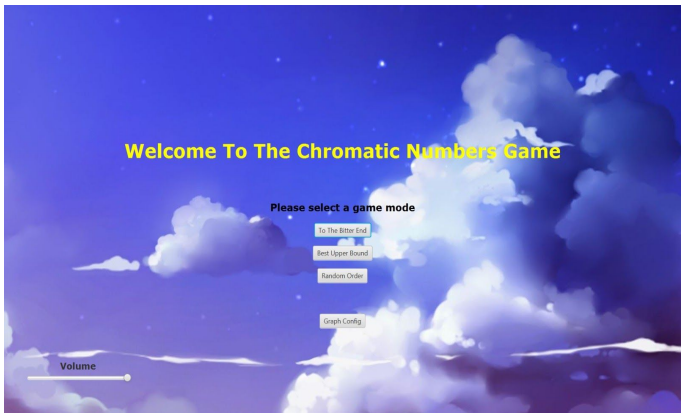
SpecialGraphs
+ isComplete(n: int, m: int): boolean + isBipartite (graph: int[][], root: int): boolean

Diagram 5: A UML diagram for the special case graphs.

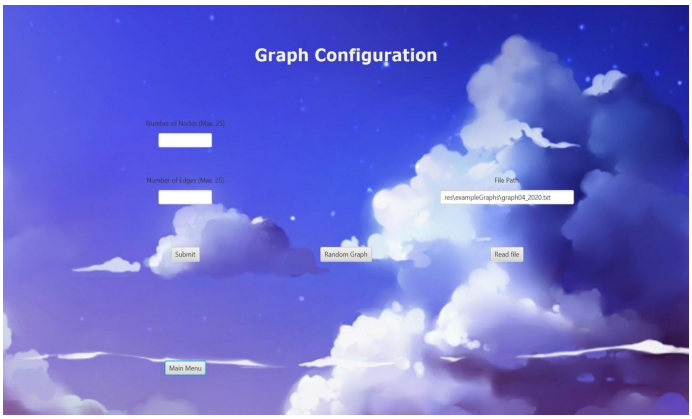
3.3 Graphical User Interface (GUI)

For this project the algorithms were implemented into a single-player game with a Graphical User Interface (GUI). It was created with the JavaFX library as it has been proven to be versatile for modern applications. The game consists of 3 game modes: To the Bitter End, Best Upper Bound, and Random Order, and the main goal is to use as few colors as possible to color the graph. (See picture 1 and 3 for reference)

Before playing the modes, the user has to configure the graph by clicking on the "Graph Config" button (see picture 2). The user then has 3 options for configuring the graph: by manually inputting the number of nodes and vertices, by reading in a graph from file (by writing out the file path), or by choosing the graph to be randomly generated by the game itself. If no specifications are given before starting one of the game modes, the graph will be generated randomly. The graph is displayed in the modes by an assortment of buttons, each representing a vertex. Depending on the graph generation, any combination of vertices and edges is possible, as long as the number of vertices and the number of edges both do not exceed 25. This limit was chosen in order to prevent larger graphs from not appearing fully on the screen. The vertices are always arranged in a circular formation, so no vertex will intersect with another.



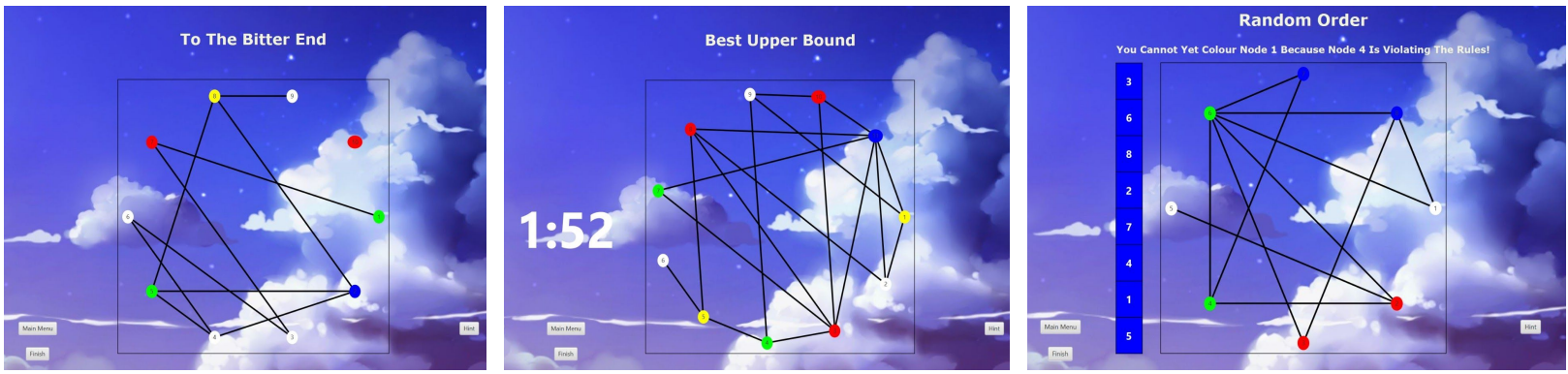
Picture 1: The main menu of the game.



Picture 2: Graph configuration scene.

In the "To the Bitter End" game mode the player should color the graph with the minimum amount of colors needed. The computer does not allow the player to finish until the chromatic number has been found. In the "Best Upper Bound" game mode the player is given a fixed amount of time to color the given graph. In this mode, however, it is not necessary for the user to find the minimum number of colors. In the "Random Order" game mode the computer generates a random ordering of the vertices and the player has to pick the colors of the vertices in exactly that order. Once the color of a vertex has been chosen, it cannot be changed again. The player can never get stuck as they are always allowed to introduce a new color. In all game modes, in order for the nodes to be colored, the player must click on each node until it has reached the desired color. The number of colors available increases every time a new color is needed, in order to prevent too many unnecessary clicks on the node. In each mode there is a hint button, which gives the user some computer-assisted help when pressed. Furthermore, the computer warns the player if their current coloring is not valid, i.e. two adjacent vertices have the same coloring. Once the player believes they have colored the graph accordingly, they can press

the finish button. Then, if they were wrong, they will receive a hint. If they colored correctly, a winning message is displayed, and then the player can either replay the same mode or go to the main menu and choose another. The chromatic number is generated as soon as the graph is created, so the program has a reference for the “To The Bitter End” mode.



Picture 3: All 3 game modes.

4. Experiments

The algorithms were tested on the provided phase 3 graphs as well as a number of graphs selected from the standard DIMACS test graphs for graph coloring. The phase 3 graphs were chosen simply because they were provided as a part of the project. Several DIMACS graphs of varying sizes and densities were chosen to showcase how the algorithms compare on graphs with much larger complexity (Trick, 1994). These larger graphs are often better able to provide insights on the significant differences between the algorithms. As these graphs are often used as test graphs, we were able to compare our results to the current known best bounds.

The algorithms (MCQ, WP, RLF, RLF-p, DSatur backtracking (DSat)) were each given a separate runtime of 2 minutes for each graph, based on the fact that the tournament allows the same time. Here, RLF-*p* references the RLF algorithm with the advanced heuristic of color class construction with a value *p*, while RLF references the basic algorithm. The RLF-*p* algorithm and the DSat were the only algorithms with varying parameters, so the other algorithms were simply run in the same state for every graph, being given the parameters mentioned in the Implementation section.

The varying parameter for the DSat algorithm, the number of allowed colors *k*, was initially set to the best result found by the basic RLF algorithm for both the phase 3 graphs and the DIMACS graphs. Once a result was found, *k* was decremented for the next iteration. The varying parameter for the RLF-*p* algorithm, *p,* was decided to be *1.0* for the phase 3 graphs, because the low complexity of these graphs meant a high value of *p* would not cause excessive computational effort. For the DIMACS graphs, the RLF-*p* algorithm was tested for several of *p*, namely $p = 0.1, 0.2, 0.4, 0.6, 0.8, 1.0$. The result of the RLF-*0.1* algorithm was then used as a

comparison to the other algorithms, and the RLF-*0.2, -0.4, -0.6, -0.8 and -1.0* were compared to each other, to see if varying the value of *p* caused significant changes in results.

If an algorithm managed to finish before the 2-minute cut-off, the time was noted down. Furthermore for each algorithm, the best result after either the cut-off or finishing the run was noted down. For all algorithms, besides the MCQ algorithm, this was an upper bound. For the MCQ algorithm, this was a lower bound.

5. Results

Tables 1 and 2 show the results of the algorithms on eight separate DIMACS graphs. For results, computed from the graphs in the given test suite for Phase 3 of the project, turn to **Appendix A: Results of Phase 3 test suite**.

Table 1 shows the best found results of all the algorithms, for these results a value of $p = 0,1$ was decided for the RLF-*p* algorithm. The graphs are noted as: name (best known upper bound); the results are noted as: result (time in ms). If no time is mentioned, the maximum allowed time was taken.

Graph	E	WP	RLF	RLF-0.1	MCP	DSAT
Dsjc125.1 (5)	736	9 (2)	9 (3)	**8 (14)**	4 (2)	8
Dsjc125.9 (44)	6961	57 (4)	57 (20)	**48 (123)**	34 (25907)	49
Dsjc250.9 (72)	27897	103 (8)	97 (84)	**80 (766)**	42	90
Dsjc500.1 (12)	12458	21 (7)	20 (34)	**17 (403)**	5 (25)	17
Dsjc500.5 (48)	62624	75 (13)	72 (154)	**59 (3226)**	13 (10934)	59
Dsjc1000.1 (20)	49629	33 (21)	31 (120)	**28 (2843)**	6 (43)	28
Dsjc1000.5 (83)	249826	131 (33)	125 (1244)	**101 (44804)**	15	101
Dsjc1000.9 (222)	449449	331 (39)	321 (2242)	**252**	60	306

Table 1: The results of the respective algorithms on 8 DIMACS graphs.
(WP = Welsh-Powell; RLF = Recursive-Largest-First; RLF-0.1 = advanced Recursive-Largest-First with p = 0,1; DSAT = DSatur backtracking).

Table 2 shows the results of the RLF-*p* algorithm being run for varying values of *p*. The results are noted as result (time in ms).

Graph	E	RLF-0.2	RLF-0.4	RLF-0.6	RLF-0.8	RLF-1.0
dsjc125.1	736	**7 (17)**	7 (80)	7 (89)	7 (72)	7 (81)
dsjc125.9	6961	48 (167)	48 (441)	**47 (520)**	47 (626)	47 (603)
dsjc250.9	27897	**77 (1173)**	80 (3789)	79 (5070)	79 (6063)	79 (6131)
dsjc500.1	12458	17 (490)	17 (1625)	**16 (2003)**	16 (2375)	16 (2455)
dsjc500.5	62624	57 (5618)	58 (18129)	**56 (24813)**	56 (29251)	57 (31469)
dsjc1000.1	49629	27 (4795)	28 (17661)	**26 (22064)**	26 (25781)	26 (27455)

Table 2: The results of the algorithm with varying values for p on 6 DIMACS graphs.
(WP = Welsh-Powell; RLF = Recursive-Largest-First; RLF-0.1 = advanced Recursive-Largest-First with p = 0,1; DSAT = DSatur backtracking).

6. Discussion

From the results shown in table 1, it is clear that the WP algorithm is consistently outperformed by the other upper bound algorithms. This is consistent with the expectations, as it has the simplest heuristics and the smallest computational effort. The gap between the amount of time taken for the WP and the basic RLF algorithm widens as the number of edges increases, but the RLF algorithm is always fast enough to make it a better option than WP in all but the largest graphs. The improvement of the RLF-*0.1* algorithm compared to basic RLF ranges from *10%* to *20%*, consistent with previous research done on this heuristic (Adegbindin et al., 2014). The improvement seems to be larger as the size of the graphs grow. This gain is significant, but comes with a large increase in effort. The difference in computational effort becomes greater as the size of the graph grows. This is sensible, as the number of trial color classes created for each iteration is dependent on *n* (10% of *n* for RLF-*0.1*). With the cut-off time for the tournament being 2 minutes, the gain in results from RLF-*0.1* is clearly worth the extra time needed for solving a graph.

The DSatur backtracking algorithm (DSat), with an initial *k* set to the result of RLF, is never able to improve on the result of RLF-*0.1*. This is due to the fact that it becomes harder to find a coloring for less colors, and the fact that these graphs are simply too large. For the phase 3 graphs (see Appendix A), however, it was able to improve on RLF-*0.1* in some cases. Perhaps more importantly, it was able to show that certain graphs were not solvable for a certain *k*, but solvable for *k+1*, therefore guaranteeing the chromatic number. This makes it worthwhile to include the algorithm. The MCQ algorithm was able to run fully on a number of graphs, but was often cut-off. Interestingly, the time taken for MCQ did not seem so dependent on the size of the graph. An example of this, seen in table 1, is it taking 43 ms to solve dsjc1000.1, with 49629 nodes, and it taking 10934 ms for dsjc500.5, with 62624 nodes. Instead, the size of the largest

clique had a stronger influence on the complexity for MCQ. For most DIMACS graphs, the result of MCQ was not close to the chromatic number of the graph, thus providing little useful information. This likely comes from the fact that for larger graphs, the maximum clique is often not a reliable lower bound anymore. On the other hand, for the phase 3 graphs, it was often able to come close to, or even match, the chromatic number, making it a useful indicator.

From table 2 it is clear that it is hard to predict which value of *p* for RLF-*p* will produce the best results. In most cases, increasing the value of *p* does not give significant improvement, the results are very similar for all values of *p*. This suggests that the best trial color class is often created with a vertex with one of the most neighbours in set U, often one in the top 10-20%. This is not in line with the original paper proposing this heuristic, where an increase in *p* often gave an increase in results (Adegbindin et al., 2014). This might be explained with the fact that the algorithms were given more time and larger graphs in that paper. On much larger graphs there might be more of a randomness factor, where the best trial color classes are not necessarily created with a vertex in the top 10-20%. The difference in time complexity for the values of *p* is consistent with the expected increase in operations, as an increase of *0.1* for *p* means that trial color classes are constructed for an extra 10% of the vertices.

With these algorithms, the chromatic numbers for 15 out of 20 phase 3 graphs were found (see Appendix A for results). This often involved DSat building on the result of the RLF algorithm to the point where it could not find a solution for a certain k. In other cases, it was the fact that the best upper bound matched the lower bound, giving the chromatic number. This shows that these algorithms can build on the results of other algorithms, and that these results can be combined to produce more certainty. However, for the larger DIMACS graphs the lower and upper bounds were never able to coincide, and DSat rarely improved significantly on the found upper bounds. Therefore, this strategy works well mostly for simpler, smaller graphs. In general, the best bounds produced by the algorithms were significantly closer to the chromatic number for smaller graphs than for larger graphs. Especially the results of the MCQ and DSat and algorithms became less relevant on larger graphs.

7. Conclusion

The aim of this project was to examine which of the developed algorithms (MCQ, WP, RLF, RLF-*p*, DSatur backtracking (DSat)) produce the tightest bounds on the chromatic number, or perhaps the chromatic number itself, as well as whether the results of separate algorithms can be combined to produce more meaningful results. To derive answers, the algorithms were tested on graphs of varying sizes and densities, and thus, different complexity. These graphs were from phase 3 test suite, as well as DIMACS graphs. The algorithms were each given a separate runtime of 2 minutes for each graph. If an algorithm managed to finish before the 2-minute cut-off, the time was noted down. Furthermore for each algorithm, the best result after either the cut-off or finishing the run was noted down. For all algorithms, besides the MCQ algorithm, this was an upper bound. For the MCQ algorithm, this was a lower bound.

The chromatic numbers for 15 out of 20 phase 3 graphs were found with these algorithms. The results make clear that the RLF algorithm, with more advanced heuristics, outperforms the simple RLF and the WP algorithms by a large margin, coming at the cost of a higher time complexity. The DSatur Backtracking algorithm is often able to improve on these

bounds and provide exact results for smaller graphs, but rarely even comes close to the result of the improved RLF algorithm on larger graphs. Finally, the MCQ algorithm provides a lower bound close to the chromatic number for small graphs, but its results become insignificant on larger graphs. We conclude that a combination of algorithms producing upper and lower bounds, and perhaps even exact results, works well for smaller graphs. However, it is clear that the DSat and MCQ algorithms suffer on larger graphs, often making their computational effort not worth the result. Therefore, in further research on these larger graphs, the focus might be on implementing and improving algorithms based on meta-heuristics. Besides that, lower bounds on the chromatic number based on other properties of the graph are worth investigating, which might provide more consistent results for graphs with a higher chromatic number.

8. References

Adegbindin, M., Hertz, A., & Bellaiche, M. (2014). A New Efficient Rlf-like Algorithm For The Vertex Coloring Problem. *Yugoslav Journal of Operations Research*, 26, 441-456. https://doi.org/10.2298/YJOR151102003A

Aslan, M., & Baykan, N. A. (2018). A Performance Comparison of Graph Coloring Algorithms. *International Journal of Intelligent Systems and Applications in Engineering*, 1-7. https://doi.org/10.18201/ijisae.273053

Brélaz, D. (1979). New methods to color the vertices of a graph. *Communications of the ACM,* 22(4), 251–6. https://doi.org/10.1145/359094.359101

Korman, S.M. (1979). The graph-colouring problem. In N. Christofides, A. Mingozzi, P. Toth, C. Sandi (Eds.), Combinatorial Optimization. (pp. 211–235). Wiley. https://doi.org/10.1007/978-0-387-74759-0_253

Lewis, R.M.R.. 2016. *A Guide to Graph coloring*. 1st ed. Springer International Publishing. https://doi.org/10.1007/978-3-319-25730-3

Musliu N., & Schwengerer, M. (2013). Algorithm Selection for the Graph Coloring Problem. In G. Nicosia, & P. Pardalos (Eds.), Learning and Intelligent Optimization. LION 2013. Lecture Notes in Computer Science, vol 7997. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-44973-4_42

Tomita, E., Kameda, T. (2007). An Efficient Branch-and-bound Algorithm for Finding a Maximum Clique with Computational Experiments. *J Glob Optim*, 37, 95–111. https://doi.org/10.1007/s10898-006-9039-7

Trick, M.A. (1994, October 26). Network Resources for Coloring a Graph. *Carnegie Mellon University.* http://mat.gsia.cmu.edu/COLOR/color.html

Welsh, D. J. A., & Powell, M. B. (1967). An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1), 85–86. https://doi.org/10.1093/comjnl/10.1.85

9. Appendices

Appendix A: Results of Phase 3 test suite

The following results are computed from the graphs in the given test suite for Phase 3 of the project.

Graph	V (vertices)	Edges (Edges)	MCQ (Lower bound)	WP (Upper bound)	RLF (Upper bound)	RLF-1.0 (Upper bound)	DSATUR (Upper bound)	X(G)
01	218	1267	6	8	8	7	6	6
02	529	271	2	3	3	3	3	3
03	206	961	3	9	7	7	5	-
04	744	744	2	3	3	3	3	2
05	215	1642	5	10	10	8	7	7
06	131	1116	10	14	14	12	10	10
07	212	252	3	4	4	4	3	3
08	107	516	4	9	7	6	5	5
09	43	529	8	12	12	12	12	-
10	387	2502	8	9	9	9	9	9
11	85	1060	9	15	14	12	11	-
12	164	323	2	5	5	4	4	-
13	143	498	11	13	11	11	11	11
14	456	1028	3	7	6	5	5	4
15	4007	1198933	2	3	6	5	2	2
16	107	4955	98	98	98	98	98	98
17	164	889	15	15	15	15	15	15
18	907	1808	3	5	5	4	4	-
19	106	196	8	8	8	8	8	8
20	166	197	2	6	4	3	3	3

Appendix B: Main class of implementation with JavaDoc

```
/**
 * requires 1 input file
 */

public class Tester {

    /**
     * The main method of the tournament code for solving a graph
     * Prints statements when new bounds have been found
     * If the chromatic number is found, it is printed and
     * the program is terminated
     * @param args exactly one input file,
     * in the format of the given test graphs
     */
    public static void main(String[] args) {
        FileRead reader = new FileRead();

        // DIMACSRead reader = new DIMACSRead();

        //an object for storing all the read data from the file
        ReadData data = new ReadData();
        data = reader.read(args[0]);

        //creates an adjacency matrix of the graph
        AdjMatrixCreator matrixCreator = new AdjMatrixCreator();
        int[][] graph = matrixCreator.create(data.edges,
data.nodes);

        /**
         * initialize all the used algorithms
         * note that the WP algorithm was not used for the
         * final tournament code
         */
        RLF rlf = new RLF(graph, data.degArray, data.edges);
        MCP lowerCalc = new MCP(graph, data.degArray);
```

```
            DSaturBacktrack dsat = new DSaturBacktrack(graph,
data.degArray);
            RLFn rlfn = new RLFn(graph, data.degArray, data.edges);
            SpecialGraphs specCase = new SpecialGraphs();

            int ub;
            int lb;

            int n = data.nodes; // |V|
            int m = data.edges.length; // |E|

            /**
             * Testing for special cases;
             * if a graph is bipartite, X(G) = 2
             * if it's complete, X(G) = n
             */
            if (specCase.isBipartite(graph, 0, data.edges)) {
                System.out.println("CHROMATIC NUMBER = 2");
                System.exit(0);
            }
            if (specCase.isComplete(n, m))
{System.out.println("CHROMATIC NUMBER = " + n); System.exit(0);}

            //runs the MCQ algorithm, the cap is 40 seconds
            lb = lowerCalc.search();
            System.out.println("NEW BEST LOWER BOUND = " + lb);

            //runs the basic RLF algorithm for a fast first upper boound
            ub = rlf.solve();
            System.out.println("NEW BEST UPPER BOUND = " + ub);

            //if the bounds match, it's the chromatic number
            if (ub == lb) {System.out.println("CHROMATIC NUMBER = " +
ub); System.exit(0);};

            int tempUB;
            double incr;
            double P;
            long allowed;
```

```
/**
 * These if-statements decide how much time
 * the RLF-n algorithm is assigned.
 * It also decides the first value of p for RLF-n.
 * If the graph is bigger,
 * more time is assigned to RLF-n, a
 * s it brings more value than the DSat,
 * which rarely gives improvements on large graphs.
 */
if (m < 100000) {
    P = 0.2;
    if (m < 10000) {allowed = 25000;}
    else if (m < 35000) {allowed = 40000;}
    else {
        allowed = 55000;
    }
}

else if (m < 200000) {
    P = 0.1;
    allowed = 70000;
}

else {
    P = 0.05;
    allowed = 70000;
}

/**
 * These statements decide the values by which p
 * is increased after a run of RLF-n.
 * If the graph is larger, RLF-n will take longer
 * and will thus be run less times.
 * Therefore, it must be incremented by a larger amount,
 * as RLF-n will only run if p < 1.0.
 * For smaller graphs, the increment can be smaller,
 * so RLF-n is run more often,
 * giving more possibilities for an improvement.
 */
if (m < 15000) {
```

```
                incr = 0.1;
            }
            else if (m < 25000) {
                incr = 0.15;
            }
            else if (m < 35000) {
                incr = 0.25;
            }
            else {
                incr = 0.25;
            }

            /**
             * RLF-n is run for increasing values of p,
             * until it exceeds the allowed time
             * or p has reached 1.0, at which point
             * it makes no sense to increase p
             * If the new bound improves on the best bound,
             * it is stored.
             */
            long start = System.currentTimeMillis();
            while (System.currentTimeMillis() - start < allowed && P <=
1.0) {
                tempUB = rlfn.solve(P);
                if (tempUB < ub) {
                    if (ub == lb) {System.out.println("CHROMATIC NUMBER
= " + ub); System.exit(0);};
                    System.out.println("NEW BEST UPPER BOUND = " +
tempUB);
                    ub = tempUB;
                }
                P = P + incr;
            }

            /**
             * The remaining time is spent running the DSat algorithm to
improve
             * on the current best upper bound, starting with ub-1.
             * If it cannot solve it for a certain k colors,
             * we know the chromatic number is k+1
```

```
                 */
                int max = ub-1;
                while (ub != lb) {
                    if (dsat.solve(max)) {
                        System.out.println("NEW BEST UPPER BOUND = " + max);
                        if (max == lb) {System.out.println("CHROMATIC NUMBER
= " + max); System.exit(0);};
                        ub = max;
                        max--;
                    }
                    else {
                        System.out.println("CHROMATIC NUMBER = " + (max+1));
                        System.exit(0);
                    }
                }

            }
        }
```