Group 18 – Project1.1 – Final Report

BERGDAL MJENGWA Manfred, BERSIA Adrien, CHAKRIAN Gevorg, PANGHE Ada, PICQUET Marie, REIS RIO Diogo, VERŞEBENIUC Dumitru

ABSTRACT

This report presents an algorithm for the graph coloring problem. The graph coloring problem is a well-known and extensively researched NP-hard problem. The goal of the presented algorithm is to compute, in the shortest time possible, an upper bound, a lower bound, and if possible, the exact chromatic number of any graph. The method used is a combination of pattern-recognition algorithms, a tabu search and a genetic local search algorithm using two different crossover methods, an algorithm to find the maximum clique number of a graph, and a backtracking algorithm. This approach allows us to reach the chromatic number of most of the graphs of our benchmark and get good boundaries on the others. Some ways of improving these results will also be discussed. We also present the specific implementation details of a graph coloring game able to use the previous algorithm to compute in-game hints to help the player to solve his coloring.

TABLE OF CONTENTS

Abstract 1

Introduction 2

1. Methods 2

1.1. Definitions 2

1.2. Graph Coloring Game 2

1.3. Chromatic number Calculation 6

2. Implementation 12

2.1. Storing methods 12

2.2. Graph Coloring Game 12

2.3. Chromatic number Calculation 14

3. Experiments 18

3.1. Benchmarks 18

3.2. Score and Timer Formula Generation 18

3.3 Upper bound 18

4. Results 19

4.1. Upper Bound 19

4.2. Final Program results 19

5. Discussion 20

5.1. Graph Coloring Game: Graph display 20

5.2. Chromatic number Calculation: Graph reduction 20

6. Conclusions 20

7. References 20

8. Appendice 22

INTRODUCTION

The graph coloring problem is a well-known and studied NP-hard problem[1, Ch. 14]. The goal of the problem is to find the minimal number of colors we can use to colorize every node of an undirected graph such that no adjacent nodes have the same color. This number is called the *chromatic number* of the graph. Given the NP-hardness of the coloring problem, this number is not always reachable in a reasonable time. In such a case we want to have the best lower and upper bounds of the chromatic number possible. This means we can use a range of different methods to reach these numbers faster. The goal of this report is to present a few of these methods and explore the situations where they can be used with the best efficiency.

Most of the following algorithms already exist and are often considered as part of the most efficient ones of their category as we write this report. We will try in the following to find the best combination of these algorithms to get suitable results. For that we are going to present the results of a program combining:

For the upper bound calculation:

- a tabu search algorithm called Tabucol [2]
- a genetic algorithm using a Tabucol variation as mutation and a UIS (Union of Independent Sets) crossover [3] and a second one using a self-developed variation of the UIS.

For the lower bound: a pruning based algorithm calculating the maximum clique number of a graph [4], for the exact chromatic number: a backtracking algorithm [5], that we tried to combine with a graph reduction method, and for the pattern recognition a program identifying known structures from graph theory.

The methods used to create an interactive game allowing the user to colorize a graph with the help of our algorithm are also presented.

1. METHODS

In the following section the global functioning of our program is presented.

1.1. Definitions

In this report, we will mostly follow the notation used in [6]:

Given a graph $= (\mathrm{V}, \mathrm{E})$ with vertex set V and edge set E, and given an integer k, a k-coloring of G is a function $c: V \to \{1, \dots, k\}$. The value $c(x)$ of a vertex x is called the color of x. The vertices with color i $(1 \leq i \leq k)$ define a *color class*, noted V_i. If two adjacent vertices x and y have the same color i, the edge $[\mathrm{x}, \mathrm{y}]$ is said to be conflicting. A k-coloring without any conflicting edges is considered legal. A *stable set* is a subset of vertices no two of which are adjacent. Hence, a k-coloring is legal if and only if its color classes are stable sets. The graph coloring problem (GCP for short) is to determine the smallest integer k, called chromatic number of G and denoted $\chi(G)$, such that there exists a legal k-coloring of G. Given a fixed integer k, the problem of determining whether there exists a legal k-coloring of G is called k-GCP.

1.2. Graph Coloring Game

Here the focus will be set on the created game.

1.2.1. Graph creation

The game implemented allows the user to open an existing graph or create a random one. To create a random graph *G* without subgraphs or non-connected vertices the graph creation algorithm

requires certain parameters as the number of edges is related to the number of vertices. To control this bond, we set boundaries for the number of edges:

The lower bound is $|\mathrm{V}| - 1$ which is the minimum number of edges to create a connected graph with $|V|$ vertices, see appendix B. The maximum number of edges is translated by the creation of a *complete* $|V|$, defined as a graph where every node is connected to all the others, see appendix C. This means that a graph of $|V|$ can have a maximum number of edges of $\frac{|V|\times(|V|-1)}{2}$. If these parameters are correct, the program creates a random graph such that all nodes are in a single graph.

1.2.2. Graph display

The main challenges to tackle in the display of a graph are to find a single method working for every type of graph, and to avoid having edges crossing non-connected vertices. These constraints guided us to choose to spread evenly all vertices on a circle, with their edges represented as straight lines between them. However, this condition means that in some cases (especially in graphs with a lot of vertices and edges between close vertices) an edge is crossing another vertex. This default was solved by choosing to make the edge go outside the circle to go around the vertices on its way. See appendix G.

The sizes of the vertices and circle on which they are located are defined, with D the diameter of the circle, dv the diameter of the vertices and θ the angle between two vertices (*Fig 1*), with the following criteria:

$$\begin{cases} D + dv = \frac{4}{5} \times Panel\ size \\ dv = distance\ between\ two\ vertices \end{cases} \Rightarrow \begin{cases} \theta = \frac{360}{Number\ of\ vertices} \\ D = \frac{\frac{4}{5} \times Panel\ size}{1 + \frac{1}{2} \times \sqrt{\frac{1}{2} \times (1 - \cos\theta)}} \\ dv = D \times \frac{1}{2} \times \sqrt{\frac{1}{2} \times (1 - \cos\theta)} \end{cases}$$

For the representation of the edges, we can notice that an edge can be considered as crossing a non-connected vertex when:

$$\theta_i \leq \theta_{lim} \qquad \text{with:} \qquad \begin{cases} \theta_{lim} = \sin^{-1}\frac{d}{b} \\ \theta_i = \cos^{-1}\left(\frac{c^2 - a^2 - b^2}{2ab}\right) \end{cases}$$

With a the distance between the two vertices to connect, b the distance between the first vertex and its neighbor, c the distance between this neighbor and the second vertex, and d the shortest distance between the center of the neighbor and the tangent of this vertex passing by the center of the first vertex. θ_i is the angle between a and b, and θ_{lim} the angle between b and the tangent aforementioned (*Fig 2*).

If $\theta_i \leq \theta_{lim}$ the edge can't be drawn straight. Then:

We take the middle of a, and use it to calculate the angle θ allowing us to determine the coordinates of M, the point where our edge is going to change direction, before drawing the line between the center of the vertex i and M, and between M and the center of the vertex j (*Fig 3*):

$$\begin{cases} x_M = D_x + \cos\left(\theta \times \frac{i+j}{2}\right) \times R \\ y_M = D_y + \sin\left(\theta \times \frac{i+j}{2}\right) \times R \end{cases}$$

R represents a repel factor: the further away the two vertices are the further away from the center M is (for example, on appendix *G,* vertices 1 and 3 are only separated by 1 vertex, so M will be closer to the center than if we wanted to connect 1 and 4). R is defined as:

$$R = \frac{D}{2} + d \times \frac{|i-j|}{2}$$

D_x and D_y represent the x and y components of the coordinates of the center of the graph.

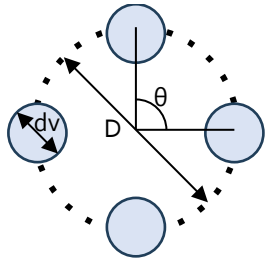


Fig 1: Vertices location parameters

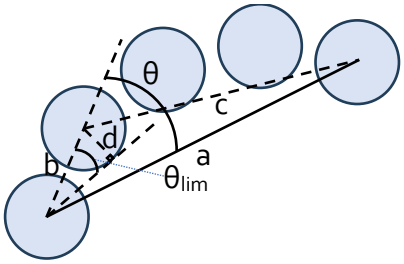


Fig 2: Edges representation

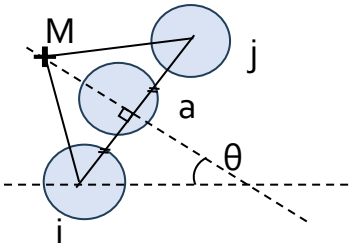


Fig 3: Bypassing edge representation

1.2.3. Score calculation and display

In each game mode, we implemented score formulas that aim to determine the player's performance for all completed graph colorings.

After each achieved score, if it is one of the top ten for its respective game mode, then that score gets stored accordingly to a file which is then displayable in a high scores table accessible from the main menu.

1.2.3.a. To the Bitter End and Random Order Game Modes

The score formula for these two game modes is the same, since they both depend on similar variables.

$$SCORE = 100.\log\left(10 + \frac{\left(\frac{numV + numE}{2}\right)}{numUsedColors + timeUsed}\right)$$

Going through each parameter, the reason the 100 multiplier is introduced is to give a more "arcade-like" score, so that the numbers used to evaluate the player's performance are more appealing and feel more rewarding. The logarithmic function is used to avoid dispersion when attributing the scores. Ideally, graphs with very different inherent difficulty and requiring more efforts than others should be scored accordingly. Even so, the span should not be too large. The initial addition of 10 inside the logarithmic function is simply to avoid any negative scores that would be generated whenever the second term of the addition results in a number smaller than 1, which is common. Opting for 10 instead of 1 allows us to have a good solid ground equal to the base of the logarithmic function. The second term of the addition inside the logarithmic function is used to determine the influence of each variable in the player's performance. The inherent difficulty of the graph is being equally dependent on the number of vertices and on the number of edges, therefore, those variables increase the score. The number of colors used to color the graph determines the ability of the player, thus, the

bigger that number, the worse the performance (this is graph dependent for the bitter end game mode, since the player needs to get the chromatic number right, but its use is still relevant, since it inherently influences the graph difficulty). The same reasoning applies for the time used. The faster the player is in coloring the graph, the smaller the discount – which leads to better scores, when compared to a player that required more time.

NOTE: For the Random Order game mode, the number of colors used to color the graph does not necessarily have to be equal to the chromatic number. Even so, if the player manages to get it right, then that performance will result in an associated score with a bonus of 5 points.

1.2.3.b. Best Upper Bound Game Mode

The score formula for this game mode has a few differences, which will be accordingly explained.

$$SCORE = \frac{1}{dif}.100.\log\left(10 + \frac{\left(\frac{numV + numE}{2}\right) + extraTime}{numUsedColors}\right)$$

In order to clearly explain the new variables' influence, it is relevant to introduce the difficulty (*dif*) variable and where it comes from. For that, a formula specific to this game mode only, needs to be mentioned as well. That formula is the timer formula, capable of estimating what time it should take the player to completely color each given graph.

$$TIME = dif.\left(3.7\left(\frac{numV + numE}{2}\right) + 1.7\right)$$

In the experiments section, the way this linear regression formula was generated will be clarified. For now, the main goal of showing it is to introduce the said *dif* variable. Before a game is started in the best upper bound game mode, the player is asked to pick the time difficulty for the already defined game graph, see appendix H.

Depending on which difficulty is selected, the *dif* variable is accordingly updated:
- EASY, *dif* = 3/2 (bonus of 50% of the original time)
- NORMAL, *dif* = 4/3 (bonus of 33% of the original time)
- HARD, *dif* = 5/4 (bonus of 25% of the original time)

NOTE: The above-mentioned time difficulty selection should not be confused with the "random" graph generation settings options. In the latter, the difficulty aspect comprises itself within the complexity of the graph in terms of its number of vertices and edges – not time wise. That selection defines the complexity of the graph the player will be playing with – and it is referent to all the game modes (since, again, it defines the graph itself), see appendix I. The time difficulty selection simply attributes more or less time – based on the fractions previously presented – from which the player is expected to completely color the graph; this is specific to the best upper bound game mode. Thus, these are two different types of difficulty.

Returning the focus back to the score formula, it should be easier to understand the *dif* variable's influence on evaluating the player's performance. The easier the time difficulty selected for one same graph, the bigger the deduction on the player's score. Only 66% of the rest of the formula will be counted for the score of a player that chose the EASY difficulty, since the player had more time in the first place. Therefore, 75% for NORMAL and 80% for HARD.

The last variable is the extra time variable. Since in this game mode the player has a limited time to complete the coloring of the graph – a timer is set. That timer follows the time obtained in the

time formula above, from the number it gets from that formula down to zero. The bigger the number of seconds that the player had yet to complete the graph coloring, in case of an early finish, the larger the extra time variable is (time that was still left for the timer to reach zero). Thus, the larger the extra time variable, the larger the increase on the score.

1.2.4. Hints

Two different hint functions were implemented in the Graph Coloring game logic.

1.2.4.a. The Regular Hint Function

The regular Hint function is displaying the color of the vertices connected to the selected one, see appendix D.

1.2.4.b. The Advanced Hint Function

The advanced Hint function is a more complex and informative hint, in case the regular advice is not enough for the user. The advanced Hint function is based on the backtracking algorithm, which suggests a color for the current vertex to be used. When the user presses the button “Adv. HINT”, the backtracking algorithm starts calculating the chromatic number or optimal color of the graph. If it takes more than one minute, the user will be shown a warning, see appendix E, telling him that the backtracking algorithm cannot calculate the chromatic number or optimal color of the graph in the given time, and it will offer two options: start the calculation again without a time limit or stop it and continue to play without an advanced hint for the moment.

If the backtracking algorithm successfully ends the calculation, the current selected vertex and its optimal color at this moment of the game are displayed, see appendix F.

1.3. Chromatic number Calculation

In this part the chromatic number calculation will be discussed.

1.3.1. General Overview

The global process of our calculation consists of reducing the graph, if possible, by calculating properties of its subgraphs (we define a subgraph as a part of a graph such that there are no edges between itself and the rest of the graph). In general, we try to make the boundaries of the chromatic number meet, and otherwise go through a brute force method. This can be resumed by the following flowchart (see (*Fig 4*)):

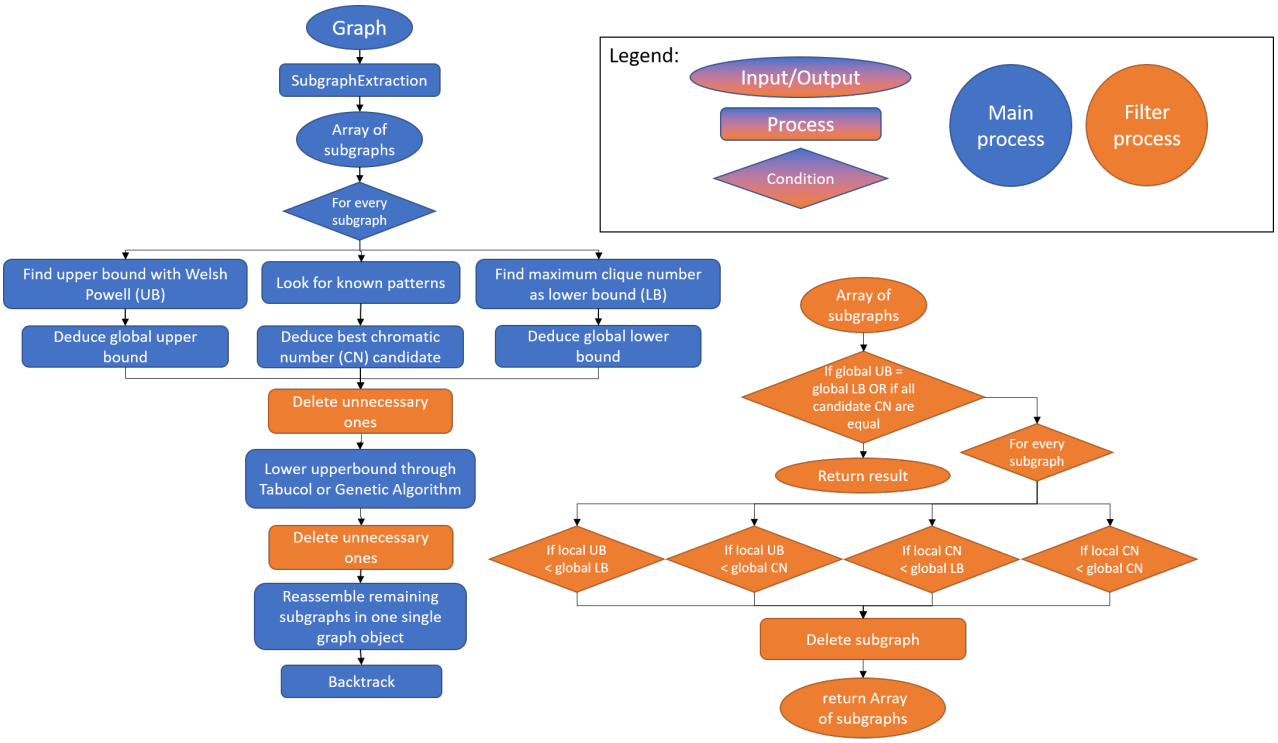


Fig 4: Global process overview: Chromatic number Calculation

1.3.2. Subgraph Extraction

Given the fact that there is no guarantee that every graph has all its vertices connected to each other (directly or indirectly), we opted to consider the possibility that an inputted graph can have more than one non-connected subgraph. A subgraph, on its essence, does not necessarily have to be disconnected from other subgraphs, however, by finding subgraphs isolated from each other, in one same inputted graph, we hope to optimize the overall procedure of our program.

Since our goal is to find the chromatic number of the overall inputted graph, by first **acknowledging the non-connected subgraphs**, we are able to quickly analyze them separately, using the upper and lower bound calculation algorithms (as well as the (sub)graph type recognition algorithm, described in the subsection 1.3.5). By performing such an analysis, we can swiftly ignore irrelevant subgraphs for the overall graph chromatic number calculation.

The way we extract and prepare the use of the non-connected subgraphs consists in two main parts. First, we start by finding them. This is done using a **Breadth-First Search** algorithm that goes through the inputted graph and continuously stores the vertices as they are found. We are then able to compute the adjacency matrix of our subgraph by comparing the list to the original graph and then by normalizing the denomination of every vertex from 1 to the number of nodes of the subgraph. This adjacency matrix is used to create a separate graph entity for each subgraph, that we can store in a list of graphs from which we can work with.

From this step, as working with subgraphs, we cannot stop as soon as we find a solution: the upper bound of one graph can be lower than the chromatic number of another graph, and so lower than the global chromatic number. This apparent default can be turned as an advantage as we can now discard the first subgraph from any further calculation, simplifying the global process. A few conditions can so help us reduce the size of the graph, and so, a graph can be discarded if its upper bound or chromatic number is lower than another known chromatic number or the global lower bound.

Once we have deleted all irrelevant subgraphs, and if this does not allow us to find the global chromatic number, our main program goes through a Backtracking algorithm (see Backtracking section). The backtrack process, by its nature can be a long process. For this reason, we don't want to spend time on a possibly local solution. To solve this problem, we need to reassemble our subgraphs in only one graph entity. We choose to do that instead of using the original graph as we have been able to delete non relevant graphs, and so lower the total number of vertices. For that we go through the opposite process by normalizing all subgraphs such that the vertex IDs go from 1 to the total number of vertices left. We then store them and their connections as a single graph.

1.3.3. Upper bound

To calculate the upper bound, we used 4 algorithms: the Welsh-Powell algorithm, the Tabucol, a genetic algorithm using a Tabucol variation as mutation and a UIS as crossover [3] and a second one using a self-developed variation of the UIS, which are explained below.

1.3.3.a. Welsh-Powell

The algorithm developed by D. J. A. Welsh and M. B. Powell in [7] allows us to compute on a very fast way a very simple upper bound of the chromatic number by colorizing the graph starting by the highest degree, and with the same color all non-adjacent vertices, then to add a new color and go again starting from the next non-colorized vertex with the highest degree and so on, until all the graph is colorized. The number of colors used is an upper bound of the chromatic number.

1.3.3.b. Tabucol

Tabucol is a tabu search algorithm for the k-GCP first developed by Hertz and de Werra [8] in 1987. We implemented a later version of this algorithm, developed by Galinier and Hao [9], and improved it following the data structure advised by Galinier and Hertz [2]. The following description is interpreted from [2]:

Tabucol first generates an initial k-coloring, allowing conflicting edges. Then will try to solve these conflicting edges by changing the color of one vertex at a time, until a legal k-coloring is reached. Every move is stored in a tabu list to avoid cycling, and so be able to escape from a local optimum. The number of conflicting edges for a coloring c is called cost-*function*, denoted f, such that $f(c) = \left|E_{conflicting}\right|$. A coloring is legal if and only if $f(c) = 0$.

The transformation of the present coloring is done by a *1-move* transformation, denoted (v, i), for a vertex v and a color $i \neq c(v)$, and assigning the color i to v. The result on the global coloring is noted c' and is defined as:

$$\forall x \in V: c'(x) = \big(c + (v,i)\big)(x) = \begin{cases} i & , x = v \\ c(x) & , \forall x \neq v \end{cases}$$

At every iteration of the Tabucol, there exist $|V| \times (k-1)$ different possible 1-moves. The idea is to select and perform the 1-move that will reduce the most the value of f (ties are broken randomly). This reduction is called the *performance* of the 1-move, denoted and defined as: $\delta_{(v,i)} = f(c') - f(c)$. If $\delta_{(v,i)} < 0$, (v, i) leads to a decrease of the cost function. This way of computing however leads to a complexity of $O(kF(c)|V|)$ for every iteration ([2]). A solution to reduce this complexity is to compute the γ-matrix, of size $|V| \times k$ and storing the $\gamma(v, i)$s of the graph denoting the number of vertices adjacent to v and of color i in the current coloring c. This γ-matrix is initialized at the beginning of the process and updated at every performed 1-move. The performance can now be computed as $\delta(v, i) = \gamma(v, i) - \gamma(v, c(v))$, in a constant time. To update the γ-matrix, it is sufficient for a 1-move (v, i), for every vertex w connected to v, to increase by one the element (w, i) and decrease by one the element $(w, c(v))$. This data structure allows us to

reach a complexity of $O(\min\{kF(c), |\Gamma(v)|\}$ by iteration, with $\Gamma(v)$ the set of vertices connected to v.

A 1-move is said to be *critical* if involving a conflicting vertex (i.e.. a vertex involved in a conflicting edge). For efficiency reasons, Tabucol only performs critical moves.

Every time a 1-move (v, i) is performed on c, we store the pair (v, i) in a tabu list for n iterations. (This is a slight variation as the original code stores the previous color of v, but our experiments show that this method works better on numerous benchmark graphs) This means that for n iterations the 1-move (v, i) will be forbidden (an exception is made if the tabu move allows to reach $f(c) = 0$). The number of iterations n is defined as follows:

$$n = L + \lambda \times F(c)$$

With L a parameter chosen randomly between 0 and 10 [2], λ another parameter, and $F(c)$ the number of conflicting vertices.

The algorithm stops as soon as $f(c) = 0$ or if the time limit (maximum number of iterations in the genetic algorithm variation) is reached.

1.3.3.c. Genetic Algorithm - UIS

The range of search processed by Tabucol on its own can be broadened by a population-based algorithm. We can moreover try to combine interesting features of two different colorings to try to reach a lower upper bound in a most efficient way. For this, we need to implement a genetic algorithm like the one created by Dorne and Hao in 1998 [10] and using a UIS (Union of Independent Sets) crossover. This algorithm is in the following shortened to GA.

On the same principle as Tabucol, GA tries to minimize the cost function f and stops if $f = 0$ or if the time limit is reached. For this, and at every iteration, we create random pairs of individuals within our population (every individual is paired with another one, avoiding repetition). Then, and for all pairs, we cross the two individuals with a probability that we call *cross ratio*. The UIS crossover gives us two offspring, that we use to replace the two parents, regardless of their fitness. We then improve all individuals by mutating them with a number of iterations of Tabucol. The Tabucol however is slightly modified to that purpose, in his inputs for example, but also by adding to it a *random walk* feature: with a probability of *random walk ratio* at every iteration we are going to choose a random 1-move to make instead of the best one. This is done to avoid a local optimum and discover new solutions, as we might not do as many iterations as a regular Tabucol for efficiency reasons.

Except for the abovementioned modifications, we use Tabucol as described previously. The fitness function of the GA, represented by the cost function, is also identical to the Tabucol cost function.

As a reminder, for a k-coloring of G, we have a function $c: V \to \{1, \dots, k\}$ with $c(x)$ representing the color of the vertex x. We can now define a *color class* V_i as the set of all vertices of color i. A *stable* set can be defined as a set of vertices no two of which are adjacent (i.e.. no conflicting edges within the set). For each color class of a coloring c, we can extract the largest stable subset, called *independent color* class, that we denote $I_{c,i}$. The UIS crossover presented by [10] is based on the unification of independent color classes of the two individuals of a pair (in the following Ind_1 and Ind_2) and gives as output two new individuals called $Child_1$ and $Child_2$: If we call the coloring of Ind_1 c and of Ind_2 d, we take, for every color i of $Child_1$, the union of $I_{c,i}$ and $I_{d,j}$, the independent color class of Ind_2 that has the most common vertices with $I_{c,i}$. The vertices l that aren't colorized after this process keep the color of Ind_1, $c(l)$. We then repeat the same process inverting Ind_1 and Ind_2 for $Child_2$. See appendix J.

We also implemented a variation of the UIS crossover, were instead of doing the unification of the sets with the most common vertices we decide arbitrarily which sets to unit (for example $I_{c,i}$ with $I_{d,i+1}$ if $i < k$ and with $I_{d,0}$ otherwise). These two variations can be compared in the experiments section.

1.3.4. Lower bound: Maximum clique problem

The maximum clique problem is to find the biggest complete subgraph of G, where a complete subgraph of G is a subgraph whose vertices are pairwise adjacent. By deduction, the chromatic number of a complete subgraph of G is always the number of vertices of the subgraph. And since the chromatic number of a subgraph of G is always smaller or equal to the chromatic number of the initial graph G, the maximum clique size is a lower bound of G.

To find the maximum clique a partially enumerative algorithm developed by Carraghan and M. Pardalos [4] was used, very efficient and simple to implement: First, the vertices were ordered from smallest to biggest degree in G. This vertex ordering reduces the computational time with large graphs. Furthermore, heuristics and pruning techniques are applied to reduce the search space. The algorithm is based on depth search, if we consider vertex v_1, at depth 2 we consider all vertices adjacent to v_1. At depth 3, we consider all vertices that are in depth 2 and adjacent to the considered vertex in depth 2 and the previous one, and so on. Let v be the vertex we are currently expanding at depth d with m being the number of connected vertices and i being the vertex number at depth d. If $d + (m - i) \leq size\ of\ the\ current\ best\ clique$, we prune because the size of the largest possible clique from this step is less or equal to the current best clique. If we are in depth 1 and this holds, then we stop.

1.3.5. Graph type recognition

Our program recognizes multiple types of graphs based on the rules and definitions provided in graph theory [11]. We recognize 6 different types: complete graph, singleton graph, tree graph, bipartite graph, cycle graph, and wheel graph.

We define a complete graph as a graph where each vertex is adjacent to all other vertices in the graph. The number of edges in a complete graph is equal to n*(n-1)/2, where n is the number of vertices of that graph. We check for this condition to be met in order to identify a complete graph. Its chromatic number is equal to the number of vertices. A singleton graph is composed of one vertex with no edges. We use the same condition to identify this type of graph. Its chromatic number is equal to 1.

A tree, a star graph, and a bipartite graph have the exact chromatic number equal to 2. A bipartite graph, also known as a bigraph, can be divided into two disjoint sets, such that no two vertices within the same set are adjacent. A tree graph is an undirected graph, that contains no cycles or self-loops. There is one path between any two vertices, and the number of edges is equal to n-1. To be able to recognize a tree we are using a depth-first search algorithm. A star can be considered a special type of tree graph, where the root is the node with degree n-1, and all other nodes are children. A star graph does not have cycles or self-loops and has n-1 edges. In our program, we are checking if the degree condition is met to identify this type of graph.

The cycle graph contains a cycle through all nodes. Its chromatic number differs by the number of nodes: if n is even the chromatic number is 2, otherwise, for n odd, it is 3. We use a depth-first search algorithm to check if a graph has a cycle through all nodes. As we want the whole graph to be a cycle, not just to contain one, we count the number of different vertices that were seen in the search and check if it's equal to the number of vertices in the graph. A cycle graph returns to the initial vertex, so we adapted the search to check that. For efficiency reasons, we start the search with the node with the highest degree. A wheel graph can be defined as the junction of a singleton graph and a cycle graph

of cycle n-1. Based on the parity of the vertices, the chromatic number can be 4 if n is even, or 3 if it is odd. For a wheel graph, we check the number of edges and whether it has more than 3 nodes.

1.3.6. Backtracking

The backtracking algorithm is a type of brute force search. [12]

Backtracking is a depth-first search because it will explore one full branch as a possible solution before moving to another one. The approach of this algorithm can be summarized as [5]:

While there is another configuration, it generates the next one. A configuration is valid if the graph has no two connected vertices colored with the same color. If the combination is valid, it tries decreasing the upper bound and tries to generate another coloring, with fewer colors. Then, the algorithm returns the number of colors used in this combination of vertices.

The exact algorithm takes the graph, lower and upper bounds as parameters. It then creates a variable m that takes the value of the upper bound and decreases by 1 after each successful call of the backtracking algorithm. When reaching an unsuccessful try, it takes the previous number, which is the chromatic number. If it is possible to color the graph, the algorithm returns the number of colors (m). If it is not, it returns 0. See appendix K.

1.3.7. Graph Reduction Method

The graph reduction method is used to deal with large graphs. The method is based on a key concept called degree bounded independent set [13]. An important observation is that we can reduce the graph by removing a ℓ-degree bounded independent set (ℓ is the lower bound of the chromatic number) while preserving optimal solutions. That is, any optimal solution to the remained graph can be extended to an optimal solution to the original graph by coloring the removed vertices iteratively.

Definition Given a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, a k-degree bounded independent set is an independent set I such that $\forall v \in I, d(v) < k$.

With the above definition, [13] propose a reduction rule denoted as BIS-Rule, where BIS stands for Bounded Independent Set. Additionally, [13] proves an important property about this rule:

BIS-Rule: Given a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ and a lower bound on $\chi(\mathrm{G})$, denoted as ℓ, find an ℓ-degree bounded independent set I, and remove all vertices in I and their incident edges from G.

Proposition Given a graph G = (V, E) and an ℓ -degree bounded independent set I in it, and $\chi(\mathrm{G}) \geq \ell$,
1) if $\chi\ (\mathrm{G}\ [\mathrm{V} \setminus \mathrm{I}]) < \ell$, then $\chi(\mathrm{G}) = \ell$.
2) if $\chi\ (\mathrm{G}\ [\mathrm{V} \setminus \mathrm{I}]) \geq \ell$, then $\chi(\mathrm{G}) = \chi\ (\mathrm{G}\ [\mathrm{V} \setminus \mathrm{I}])$.

This proposition (Proved by [13]) shows that the BIS-Rule can be used to find the chromatic number. That is, using the graph G' which is reduced from G by the BIS-Rule we can find the optimal solution for the original graph much faster. As [13] has shown, the BIS-Rule exploits a lower bound on the chromatic number. Thus, before each round of reduction, we need to calculate a lower bound for the remaining graph. It is important to note that, any lower bound for any subgraph of G is a lower bound on $\chi(\mathrm{G})$ for the original input graph G. This is formally expressed in the following lemma.

Lemma Given a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, for any subgraph $\mathrm{G'} = (\mathrm{V'}, \mathrm{E'})$ of G, that is, $\mathrm{V'} \subseteq \mathrm{V}$ or $\mathrm{E'} \subseteq \mathrm{E}$ (or both), then $\chi(\mathrm{G}) \geq \chi(\mathrm{G'})$. (Proved by [13])

In our method, we find the lower bound of the chromatic number by finding the maximum clique number. Since its computation depends on the size of the graph, the fact that we reduce this graph makes it simpler at every iteration. Therefore, while the algorithm is processing, the lower bound for the chromatic number becomes more accurate. [13]

To resume the process, the goal is to reduce the size of the graph and then try to colorize the parts of the graph we removed with the smallest amount of colors. This can help us improve the lower bound.

NB: the implementation of this concept was started but not finished, and thus is only joined to this report to allow further improvements.

2. IMPLEMENTATION

In the following is first described the method to store the graph given as input or created. Then we describe the implementation of the Graph Coloring Game before observing the way the Chromatic Number is calculated.

2.1. Storing methods

Given as input a txt file containing the data describing a graph, we store the graph using two classes. The first one, called in the following *Vertex*, describes a node of the graph, storing its color and all its connected vertices and containing their accessors (with the extra care of making it impossible to modify the connected vertices after their creation). The second one, *Graph*, contains an array of *Vertex* objects and the methods allowing the modification of their colors by group or individually, with or without the condition of having a legal resulting coloring.

2.2. Graph Coloring Game

We only consider in the following the Graph Coloring Game and the stage of development reached at the end of the phase 2 for the exact chromatic number calculation.

2.2.1. Global overview

See appendix L. The process of the application is divided in two parts that we chose to call MainGUI and GraphGUI. The MainGUI part is the part driving all the game, by calling the different game modes, but also doing the interface between the game modes and the high score for example. The GraphGUI part is building the backbone of the game modes, which are going to extend this branch.

2.2.2. Graph creation

The graph creation works by iteratively creating all vertices such that they form a single graph. When all vertices are created, the remaining edges are chosen randomly.

Algorithm 1: Graph creation
Input: Vertices, Edges
Output: Random Graph
begin

```
1.      while (until all vertices are created)
2.      .       select 2 random vertices;
3.      .       if (there is no connection between them)
4.      .       .       if (1st cycle)
5.      .       .       .       add each vertex to checker;
6.      .       .       .       create an edge in between;
7.      .       .       else if (checker contains both, but not all vertices)
8.      .       .       .       skip the cycle;
9.      .       .       else if (checker contains 1st vertex, but not the 2nd)
10.     .       .       .       add 2nd vertex to checker;
11.     .       .       .       create an edge in between;
12.     .       .       else if (checker contains 2nd vertex, but not the 1st)
13.     .       .       .       add 1st vertex to checker;
```

```
14.     .       .       .       create an edge in between;
15.     end while
end
```

2.2.3. Graph display

The general algorithm of the graph display is given as follows:

```
Algorithm 2: Graph display
Input: G, a graph
Output: the graph is displayed in the GUI
% x the x-coordinate of the middle of the drawing
% y the y-coordinate of the middle of the drawing
% angleInter the angle between two vertices
% angleCounter the current angle
% d the size of a vertex
% D the size of the circle
% connected an array storing all connected vertices for a specific vertex
% middlePoint point used to make an edge go around the circle
begin
1.      angleCounter=0
2.      for every vertex in G
3.      .       Store coordinates of the vertex as the point on the circle at an angle angleCounter
4.      .       angleCounter=angleCounter+angleInter
5.      for every vertex in G
6.      .       connected=get list of the connected vertices
7.      .       for all connected vertices
8.      .       .       if edge crosses another vertex
9.      .       .       .       middlePoint=get middle of the edge+projection outside the circle
10.     .       .       .       draw line between the vertex and middlePoint
11.     .       .       .       draw line between middlePoint and the connected vertex
12.     .       .       else draw line between vertex and connected vertex
13.     draw all vertices
end
```

2.2.4. Hints

2.2.4.a. The regular Hint algorithm

The regular hint displays in a dedicated window all connected vertices and their color.

```
Algorithm 3: The regular Hint
Input: V* – current Vertex
Output: Displays connections of current vertex and colors of connected vertices
begin
1.      get adjacent vertices of V*
2.      for all adjacent vertex v
3.      .       Creates a space for the vertex in the hint frame
4.      .       if v has a color then
5.      .       .       set color of the dedicated space to color(v)
end
```

2.2.4.b. The advanced Hint algorithm

The advanced hint displays in a dedicated window the suggested color of a vertex and the minimum amount of colors needed for the coloring at a specific step.

Algorithm 4: The advanced Hint
Input: V* – current Vertex
Output: Displays the suggested color for V* and the minimum amount of colors needed at this step
%Ub Upper bound
%Lb Lower bound
%TL Time limit
begin
2. run backtrack to find the best coloring
3. **if** TL reached **then**
4. . Stop backtrack
5. . Displays a window to ask for the next action
6. . . **if** User chooses to continue without the backtracking **then**
7. . . . **end**
8. . . **else**
9. . . . continue the backtrack with no time limit
10. **else if** solution found **then**
12. . Display the color of V* found and the chromatic number in the GUI;
end

2.3. Chromatic number Calculation

We now consider the third development phase consisting of improving the chromatic number calculation.

2.3.1. Subgraph Extraction

The subgraph extraction works as follows:

Algorithm 5: Subgraph extraction
Input: A graph G= (V, E)
Output: Array of graphs
% L list of vertices of G
% SL list of vertices of a subgraph
% Q list of vertices to check
begin
1. **while** L is not empty
2. . Add L(0) to Q and SL
3. . **while** Q is not empty
4. . . Add all vertices connected to Q(0) to Q and SL if they are not already present
5. . . Remove Q(0) from Q
6. . Create the adjacency matrix for all vertices of SL
7. . Remove every vertex of SL from L
8. . Normalize the adjacency matrix by removing empty lines and columns
9. . Store the adjacency matrix as new element of an array of graphs
end

2.3.2. Upper bound

The different algorithms used to calculate the upper bound are described in this section.

2.3.2.a. Tabucol

The algorithm of the Tabucol is given as follows: [14]

Algorithm 6: Tabucol
Input: A graph G= (V, E), an integer k> 0.
Output: Solution $c*$.
%MaxIter, the maximal number of iterations

%lambda, constant of the tabu list
begin
1. Build a random solution c
2. Set the tabu list to the empty set (No move is tabu)
3. Set $c*$ = c and iter = 0
4. **while** f(c) != 0 and iter != MaxIter
5. . Set iter := iter + 1
6. . Choose a candidate 1-move (v, i) with minimum value $\delta(\mathrm{v}, \mathrm{i})$
7. . Set c = c + (v, i)
8. . **if** f(c)<f($c*$)
9. . . Set $c*$ = c
10. . Introduce move $(\mathrm{v}, \mathrm{c}(\mathrm{v}))$ into the tabu list for $L + \lambda F(c)$ iterations;
end

2.3.2.b. Genetic algorithm – UIS

The algorithm of the genetic algorithm can be given as follows: [10]

Algorithm 7: Genetic algorithm
Input: A graph G= (V, E), an integer k> 0.
Output: Solution $s*$.
%MaxIter, i the maximal number of iterations and the current one
%f, f* cost function and its best value encountered so far
%s* best individual encountered so far
%best(P) returns the best individual of the population P
begin
1. i=0
2. Build a random population P_0
3. s*= best(P_0)
4. f*=f(s*)
5. **while** f* > 0 and i < MaxIter
6. . P_i = crossing(P_i) (using UIS crossover or its variation)
7. . P_{i+1} = mutate(P_i) (using Tabucol)
8. . **if** f(best(P_{i+1})) < f*
9. . . $s*$ = best(P_{i+1})
10. . . f* = f(s*)
11. . i = i+1
12. return s*
end

NB: the best element is found by sorting the population by fitness with the use of a Heap Sort as described during the practical sessions of KEN1210 - Computational and Cognitive Neuroscience. The Heap Sort is a sorting method exploring elements as if they were in a heap such that only part of it needs to be explored at each iteration.

And the UIS crossover:

Algorithm 8: UIS crossover
Input: two individuals Ind1 and Ind2
Output: two individuals Child1 and Child2
%MaxIter, i the maximal number of iterations and the current one
%f, f* cost function and its best value encountered so far
%c(i) color of vertex i
%isValid(i) returns if the coloring of i is valid

Begin
1. **for** i going through all colors
2. . create $colorClass_{1i}$ array
3. . create $colorClass_{2i}$ array
4. **for** i going through all vertices of Ind1
5. . **if** isValid(i)
6. . . add i to $colorClass_{1c(i)}$
7. **for** i going through all vertices of Ind2
8. . **if** isValid(i)
9. . . add i to $colorClass_{2c(i)}$
10. Child1=createChild(Ind1, $colorClass_1$, $colorClass_2$)
11. Child2=createChild(Ind2, $colorClass_2$, $colorClass_1$) (note the order of the color classes)

function createChild(Ind, $colorClass_1$, $colorClass_2$)
11. **for** i going through all colors
12. . find x such that $colorClass_{2x}$ has the most common elements with $colorClass_{1i}$
13. . array=union of $colorClass_{2x}$ and $colorClass_{1i}$
14. . **for** j going through all elements of array
15. . . $c(Child_j)=c(array_j)$
16. **for** i going through all elements of array
17. . **if** $c(Child_i)$ does not exist
18. . . $c(Child_i)=c(Ind_i)$
19. **return** Child

end

2.3.3. Lower bound

The algorithm of the maximum clique number is given as follows: [4]

Algorithm 5: Maximum Clique
Input: A graph G= (V, E)
Output: Solution k (size of the maximum clique).
% U array of vertices
% N_v array of vertices adjacent to v
begin
function lowerBound
1. ordered=(order V by minimum degree)
2. max=clique(ordered,0,0)
3. **return** max

function clique(U, size, max)
4. **if** $|U|=0$ **then**
5. . **if** size>max **then**
6. . . max=size
7. . **return** max
8. **while** $|U|\neq 0$
9. . **if** size+$|U|\leq$max
10. . . **return** max
11. . $v=U_0$
12. . $U=U\backslash v$
13. . clique($U\cap N_v$, size+1, max)
14. **return** max

end

2.3.4. Backtracking

The backtracking process tries to find the lowest number for which a coloring is possible by trying all possibilities. The implementation is given as follows:

Algorithm 9: Backtracking
Input: Upper Bound (UB), Lower Bound (LB), Time Limit (TL), G= (V,E)
Output: Chromatic number
begin
1. **while** backtrack(UB,0,V,empty), UB>LB, TL is not reached
2. . UB=UB-1
3. **if** TL was not reached
4. . **return** UB
function backtrack(UB, index, V, actual coloring):
5. **if** index=V
6. . return true
7. **for** every possible color c in [1,UB]
8. . **if** index can be colored in c
9. . . **if** backtrack(UB,index+1,V, actual coloring)=true
10. . . . **return** true
11. . . **else** remove color c from index (doesn't lead to a solution)
12. **return** false
end

2.3.5. Graph Reduction Algorithm

The algorithm of the graph reduction is given as follows: [13]

Algorithm 12: FastColor
Input: Graph (G), LowerBound(lb), UpperBound (ub).
Output: Potential chromatic number
% G_k working graph which reduced from the G and becomes smaller during the algorithm.
% G_m all the vertices that have been removed from G
% lb^*,ub^* best found lower bound and upper bound on chromatic number
% α the coloring the independent set
% α^* the best coloring found during the algorithm.
% I ℓ-bounded independent set
begin
1. $G_k = G$, $lb_k = 0$, $lb^* = 0$
2. **while** (elapsed time < cutoff)
3. . lb_k = LowerBound.getSolution(G_k)
4. . **if** ($lb_k > lb^*$)
5. . . $lb^* := lb_k$
6. . I = graph.getMaxClique()
7. . G_k = remove I from G_k according to BIS-Rule;
8. . Delete all vertices, which do not have connection at all from new graph
9. . $G_m = G_m \cup I$;
10. . **if** $I \neq \emptyset$
11. . . $lb_k := 0$
12. . α = Backtracking(I)
13. . α^* = Backtracking(G_k)
14. . gets new lb^*,ub^* from the Backtraking
15. . **if** $ub^* = lb^*$
16. . . **return** α^*

17. **return** α^*;
end

3. EXPERIMENTS

3.1. Benchmarks

In this report we use a panel of 40 graphs of different sizes provided as two sets of 20 graphs by the DKE department of Maastricht University.

3.2. Score and Timer Formula Generation

For the Best Upper Bound score formula, we introduced the timer formula to explain the *dif* variable. The coding implementation procedures of the score and time calculation formulas simply depend on the direct use of the game variables that build up during the game itself.

However, the coefficients of the timer formula are worth a more detailed explanation on why they were implemented in the formula the way they were.

Essentially, before setting any kind of timer, we played several games individually registering the number of vertices, edges and the time taken to completely color the graph (correctly), at the fastest speed we could. Those would work as reference values for when we tried to compute a single linear regression formula that ideally would give us a realistic time for the player to do so in a game. Thus, we came up with two linear regression formulas (in the form of $ax + b$) based on the dataset we had, the time over the number of vertices and the time over the number of edges.

We got the equations **1.52*x* + 4.23** and **5.91*x* - 0.9**, respectively. Now the goal was to come up with a formula that fairly distributed the influence of both these parameters (number of vertices and edges) on the time required. Establishing the new **b** as the equidistant point from the b's already obtained, we got the point **P (0,1.7)**. Picking that point **P** as well as the intersection point of these two functions, **I (6.01,1.17)** a new **a** was generated. To further contribute towards the goal, the new equation should equally depend on those parameters, and that is the reason why their average behaves as the x in the newly generated timer formula.

We realize that the bigger the dataset used is, the more accurate the formula is. The reason why we stopped updating the formula at these coefficients has to do with the fact that the times it was generating were quite adequate every time we played a game.

3.3 Upper bound

We implemented four different methods to compute an upper bound of the chromatic number. We decided to use the Welsh Powell algorithm anyway in order to get a first upper bound from which to start in the Tabucol or in the Genetic Algorithm. We now have to find which one between the remaining ones is more appropriate in which condition, with the goal of finding the lowest upper bound in a reasonable amount of time. We can also compare the results with and without the subgraph extraction to see if the fact of working with only parts of the graph and then comparing them improves the result or not. We then need to find the optimal time of running, knowing that the time limit assigned to each graph is of 2 minutes, such that the upper bound is reasonably low but that we still have time left to be able to find the exact chromatic number.

The experiments were done on the following parameters:

On the Tabucol: variation of λ between 0.6 and 2, two values suggested in [2] and [3], variation of time between 10 and 30s or limited by the number of iterations, with and without graph extraction.

On the Genetic Algorithm: Variation of crossing probability, random walk probability, time, version of crossing over, number of Tabucol iterations, and with and without graph extraction.

4. RESULTS

4.1. Upper Bound

The detailed results can be found in the Appendices. In general we can notice that a running time of 10s is enough to find the best results, and a longer time isn't really needed, whatever the size the graph is. A limit by number of iterations however does not give us uniform results, hence a time limitation (10s allows us to get our best result, more is useless). For the Tabucol, the optimum value of λ appears to be 0.6, and in the Genetic Algorithm, a crossing probability of 0.02, a random walk of 0.01, and 500 iterations of Tabucol for the mutation. The Genetic Algorithm has much better results than the Tabucol. Moreover we can notice that the subgraph extraction, except on very big graphs does not have any effect on the result of the upper bound, but simplifies the backtracking, hence the decision to keep it. On a more interesting matter, we can notice that the variation of UIS crossover we implemented as the same kind of results as the original one, but has the advantage of a simplification of complexity ($O(n^2)$ versus $O(n^3)$) for the independent set selection, showing that the importance of having a big number of common vertices doesn't really improve the quality of the result. This is an improvement on the algorithm of [3].

4.2. Final Program results

The final program, chosen for the tournament, uses the following parameters: Upperbound method: Genetic Algorithm – UIS Variation, time limit of the upper bound calculation: 12s, number of iteration of the Tabucol: 500, population size: 10, crossover probability: 0.02, random walk probability: 0.01. This allows us to reach the following results on the benchmark:

Phase 1- Graph number	Lower Bound	Chromatic number	Upper Bound	Phase 3- Graph number	Lower Bound	Chromatic number	Upper Bound
Phase 1 - 1	11	11	11	Phase 3 - 1	6	6	6
Phase 1 - 2	15	X	19	Phase 3 - 2	2	3	3
Phase 1 - 3	3	4	4	Phase 3 - 3	3	X	5
Phase 1 - 4	2	3	3	Phase 3 - 4	2	2	3
Phase 1 - 5	2	2	3	Phase 3 - 5	5	X	8
Phase 1 - 6	3	X	5	Phase 3 - 6	10	10	10
Phase 1 - 7	4	4	4	Phase 3 - 7	3	3	3
Phase 1 - 8	7	7	7	Phase 3 - 8	4	5	5
Phase 1 - 9	10	10	10	Phase 3 - 9	8	X	12
Phase 1 - 10	3	X	6	Phase 3 - 10	8	X	9
Phase 1 - 11	31	31	31	Phase 3 - 11	9	X	11
Phase 1 - 12	3	X	8	Phase 3 - 12	2	X	4
Phase 1 - 13	2	2	2	Phase 3 - 13	11	11	11
Phase 1 - 14	16	16	16	Phase 3 - 14	3	X	4
Phase 1 - 15	4	5	5	Phase 3 - 15	2	2	7
Phase 1 - 16	54	54	54	Phase 3 - 16	98	98	98
Phase 1 - 17	4	4	4	Phase 3 - 17	15	15	15
Phase 1 - 18	4	4	4	Phase 3 - 18	3	X	4
Phase 1 - 19	31	31	31	Phase 3 - 19	8	8	8
Phase 1 - 20	3	X	7	Phase 3 - 20	2	3	3

5. DISCUSSION

Some ways of improvement can be considered:

5.1. Graph Coloring Game: Graph display

Even if the method employed reaches his goals by having a universal method for every type of graphs and avoiding the cases of crossing edges, we can notice that the result is lacking readability in case of larger graphs. This problem however can be balanced by the fact that this representation is used in a coloring game, and that as such, such a big graph is unlikely to be played. A solution to explore could be to divide the graph in groups of vertices that are strongly connected to each other, displayed on a circle following the actual method. Every circle would then have to be connected to the others, the challenge being to determine an optimum location to be able to reach all the vertices not part of the same circle without any crossing. We could then modify the GUI to be able to travel from circle to circle or have a general overview of the entire graph.

5.2. Chromatic number Calculation: Graph reduction

Graph reduction is a method used today to find the best upper bound in the graph, however, we tried to implement in the graph reduction the best lower bound also by trying to paint over deleted independent sets. In experiments, we ran into the problem that after several deletions of independent sets, the backtracking algorithm was not able to compute a satisfying solution.

To improve this algorithm, we could either find a new and more efficient way of colorizing the independent sets, or optimize the search of these sets.

6. CONCLUSIONS

The main goal of the presented algorithm was to compute, in the shortest time possible, an upper bound, a lower bound, and if possible, the exact chromatic number of any type of graph.

This was achieved by developing an efficient approach using a combination of a pattern recognition program working on previously extracted subgraphs, subgraphs on which we are able to compute very efficient boundaries, using for the upper bound a genetic algorithm working with the principle of a Tabucol mutation and an innovative crossover, and an efficient way to compute the maximum clique number as a lower bound. These boundaries are then further improved by a backtracking method that will eventually reach the exact chromatic number. Further research and improvements on this approach could be made by using a graph reduction program allowing us to further improve our boundaries.

7. REFERENCES

[1] M. R. Garey and D. S. Johnson, *Computer and Intractability: A Guide to the Theory of NP-Completeness*, vol. 44, no. 2. 1979.

[2] P. Galinier and A. Hertz, “A survey of local search methods for graph coloring,” *Comput. Oper. Res.*, vol. 33, no. 9, pp. 2547–2562, Sep. 2006, doi: 10.1016/j.cor.2005.07.028.

[3] R. Dorne and J. K. Hao, “A new genetic local search algorithm for graph coloring,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 1498 LNCS, no. March 2013, pp. 745–754, 1998, doi: 10.1007/bfb0056916.

[4] R. Carraghan and P. M. Pardalos, “An exact algorithm for the maximum clique problem,” *Oper. Res. Lett.*, vol. 9, no. 6, pp. 375–382, Nov. 1990, doi: 10.1016/0167-6377(90)90057-C.

[5] R. M. R. Lewis, *A Guide to Graph Colouring*. 2016.

[6] P. Galinier and A. Hertz, “A survey of local search methods for graph coloring,” *Comput. Oper. Res.*, vol. 33, no. 9, pp.

2547–2562, Sep. 2006, doi: 10.1016/j.cor.2005.07.028.

[7] D. J. A. Welsh and M. B. Powell, “An upper bound for the chromatic number of a graph and its application to timetabling problems,” *Comput. J.*, vol. 10, no. 1, pp. 85–86, Jan. 1967, doi: 10.1093/comjnl/10.1.85.

[8] A. Hertz and D. de Werra, “Using tabu search techniques for graph coloring,” *Computing*, vol. 39, no. 4, pp. 345–351, 1987, doi: 10.1007/BF02239976.

[9] P. Galinier and J. K. Hao, “Hybrid Evolutionary Algorithms for Graph Coloring,” *J. Comb. Optim.*, vol. 3, no. 4, pp. 379–397, 1999, doi: 10.1023/A:1009823419804.

[10] R. Dorne and J. K. Hao, “A new genetic local search algorithm for graph coloring,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 1498 LNCS, no. March 2013, pp. 745–754, 1998, doi: 10.1007/bfb0056916.

[11] D. B. West, “Introduction to Graph Theory (2nd Edition),” *Vaccine*, vol. 43, no. 2. p. 383, 2012, [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0521888956%5Cnhttp://view.ncbi.nlm.nih.gov/pubmed/9149362%5Cnhttp://view.ncbi.nlm.nih.gov/pubmed/14596537%5Cnhttp://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN.

[12] “Brute force algorithm - Oxford Reference.” https://www.oxfordreference.com/view/10.1093/oi/authority.20110803095532553 (accessed Jan. 11, 2021).

[13] J. Lin, S. Cai, C. Luo, and K. Su, *A reduction based method for coloring very large graphs*, vol. 0. 2017.

[14] J. K. Hao and R. Dorne, “Tabu search for graph coloring, T-coloring and set T- colorings.,” in *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, no. December 2013, S. Voß, S. Martello, I. H. Osman, and C. Roucairol, Eds. Boston, MA: Springer US, 1998.

8. APPENDICE

Appendix A

Detailed results of the upper bound experiments (the green means it is the best result found among all methods):

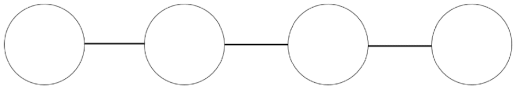
	Tabucol										
Number of iterations											
Population											
Crossover											
Random Walk											
Subgraph Extraction	N	N	N	N	N	N	N	N	N	N	Y
λ	2	2	0,6	0,6	0,6	0,6	2	2	0,6	0,6	0,6
Number of iterations	500	1000	500	1000	5000	10000	10000	30s	30s	10s	10s
Graph 1	11	11	11	11	11	11	11	11	11	11	11
Graph 2	20	20	20	20	20	20	20	20	20	21	20
Graph 3	4	4	4	4	4	4	4	4	4	4	4
Graph 4	3	3	3	3	3	3	3	3	3	3	3
Graph 5	3	3	3	3	3	3	3	3	3	3	3
Graph 6	6	6	5	6	5	5	6	6	5	5	6
Graph 7	4	4	4	4	4	4	4	4	4	4	4
Graph 8	9	9	9	9	9	9	9	9	9	9	9
Graph 9	10	10	10	10	10	10	10	10	10	10	10
Graph 10	6	6	6	6	6	6	6	6	6	6	6
Graph 11	31	31	31	31	31	31	31	31	31	31	31
Graph 12	8	8	8	8	8	8	8	8	8	8	8
Graph 13	2	2	2	2	2	2	2	2	2	2	2
Graph 14	16	16	16	16	16	16	16	16	16	16	16
Graph 15	6	6	6	6	6	6	6	6	6	6	6
Graph 16	54	54	54	54	54	54	54	54	54	54	54
Graph 17	5	5	5	5	5	5	5	5	5	5	5
Graph 18	4	4	4	4	4	4	4	4	4	4	4
Graph 19	31	31	31	31	31	31	31	31	31	31	31
Graph 20	8	8	8	8	8	8	8	8	8	8	8
Graph 21	6	6	6	7	6	6	7	7	7	7	6
Graph 22	3	3	3	3	3	3	3	3	3	3	3
Graph 23	6	5	5	6	5	5	6	5	6	5	5
Graph 24	3	3	3	3	3	3	3	3	3	3	3
Graph 25	8	8	8	8	8	8	8	8	8	8	7
Graph 26	11	11	11	11	11	11	11	11	11	10	11
Graph 27	3	3	3	3	3	3	3	3	3	3	3
Graph 28	6	6	6	6	6	6	6	6	6	6	6
Graph 29	12	12	12	12	12	12	12	12	12	12	12
Graph 30	9	9	9	9	9	9	9	9	9	9	9
Graph 31	11	11	11	12	12	11	11	11	12	11	12
Graph 32	4	4	4	4	4	4	4	4	4	4	4
Graph 33	11	11	11	11	11	11	11	11	11	11	11
Graph 34	4	4	4	4	4	4	4	4	4	4	4
Graph 35	4	4	4	4	4	4	4	4	4	4	7
Graph 36	98	98	98	98	98	98	98	98	98	98	98
Graph 37	15	15	15	15	15	15	15	15	15	15	15
Graph 38	4	4	4	4	4	4	4	4	4	4	4
Graph 39	8	8	8	8	8	8	8	8	8	8	8
Graph 40	3	3	3	3	3	3	3	3	3	3	3

	GA UIS Variation						
Number of iterations	10	10	50	10	10	10s	10s
Population	10	10	10	10	10	10	10
Crossover	0,02	0,5	0,02	0,02	0,02	0,02	0,02
Random Walk	0,01	0,01	0,01	0,01	0,01	0,01	0,01
Subgraph Extraction	N	N	N	N	N	N	Y
λ	0,6	0,6	0,6	0,6	2	0,6	0,6
Number of iterations	100	100	100	500	500	500	500
Graph 1	11	11	11	11	11	11	11
Graph 2	22	22	21	21	21	19	19
Graph 3	4	4	4	4	4	4	4
Graph 4	3	3	3	3	3	3	3
Graph 5	3	3	3	3	3	3	3
Graph 6	6	5	6	5	6	5	5
Graph 7	4	5	4	4	5	4	4
Graph 8	9	9	9	9	9	9	9
Graph 9	10	10	10	10	10	10	10
Graph 10	6	6	6	8	6	6	6
Graph 11	31	31	31	31	31	31	31
Graph 12	8	8	8	8	8	8	8
Graph 13	2	2	2	2	2	2	2
Graph 14	16	16	16	16	16	16	16
Graph 15	6	6	6	6	6	5	5
Graph 16	54	54	54	54	54	54	54
Graph 17	5	5	5	5	5	4	4
Graph 18	4	4	4	4	4	4	4
Graph 19	31	31	31	31	31	31	31
Graph 20	9	8	10	8	9	7	7
Graph 21	8	8	7	7	6	6	6
Graph 22	3	3	3	3	3	3	3
Graph 23	6	6	6	6	6	5	5
Graph 24	3	3	3	3	3	3	3
Graph 25	10	10	10	10	10	7	7
Graph 26	11	11	11	11	11	10	10
Graph 27	3	3	3	3	3	3	3
Graph 28	6	7	7	6	6	5	5
Graph 29	12	12	12	12	12	12	12
Graph 30	9	9	9	9	9	9	9
Graph 31	12	14	12	12	12	11	11
Graph 32	4	4	4	4	4	4	4
Graph 33	11	11	11	11	11	11	11
Graph 34	4	5	5	5	5	4	4
Graph 35	4	4	4	4	4	4	7
Graph 36	98	98	98	98	98	98	98
Graph 37	15	15	15	15	15	15	15
Graph 38	4	4	4	4	4	4	4
Graph 39	8	8	8	8	8	8	8
Graph 40	3	3	3	3	3	3	3

	GA UIS Original								
Number of iterations	10	10	50	10	10	50	2	10s	10s
Population	10	10	10	10	10	10	10	10	10
Crossover	0,02	0,5	0,02	0,02	0,02	0,02	0	0,02	0,02
Random Walk	0,01	0,01	0,01	0,01	0,01	0,01	0	0,01	0,01
Subgraph Extraction	N	N	N	N	N	N	N	N	Y
λ	0,6	0,6	0,6	0,6	2	2	0,6	0,6	0,6
Number of iterations	100	100	100	500	500	500	500	500	500
Graph 1	11	11	11	11	11	11	11	11	11
Graph 2	22	21	22	21	22	22	22	19	19
Graph 3	4	4	4	4	4	4	4	4	4
Graph 4	3	3	3	3	3	3	3	3	3
Graph 5	3	3	3	3	3	3	3	3	3
Graph 6	6	6	6	6	7	5	8	5	5
Graph 7	4	5	4	4	4	5	5	4	4
Graph 8	10	10	9	10	9	9	11	8	8
Graph 9	10	10	10	10	10	10	10	10	10
Graph 10	7	8	7	6	6	6	7	6	6
Graph 11	31	31	31	31	31	31	31	31	31
Graph 12	8	8	8	8	8	8	8	8	8
Graph 13	2	2	2	2	2	2	2	2	2
Graph 14	16	16	16	16	16	16	16	16	16
Graph 15	6	6	5	6	6	6	6	5	5
Graph 16	54	54	54	54	54	54	54	54	54
Graph 17	5	5	5	6	5	5	6	4	4
Graph 18	4	4	4	4	4	4	4	4	4
Graph 19	31	31	31	31	31	31	31	31	31
Graph 20	9	10	10	9	9	10	10	7	7
Graph 21	8	8	7	6	6	6	7	6	6
Graph 22	3	3	3	3	3	3	3	3	3
Graph 23	6	6	6	5	6	5	6	5	5
Graph 24	3	3	3	3	3	3	3	3	3
Graph 25	10	10	10	10	10	10	10	7	7
Graph 26	11	11	11	11	11	11	11	10	10
Graph 27	3	3	3	3	3	3	3	3	3
Graph 28	6	6	6	7	6	7	7	5	5
Graph 29	12	12	12	12	12	12	12	12	12
Graph 30	9	9	9	9	9	9	9	9	9
Graph 31	12	12	12	13	12	11	13	11	11
Graph 32	4	4	4	4	4	4	4	4	4
Graph 33	11	11	11	11	11	11	11	11	11
Graph 34	5	5	4	4	4	4	5	4	4
Graph 35	4	4	4	4	4	4	4	4	7
Graph 36	98	98	98	98	98	98	98	98	98
Graph 37	15	15	15	15	15	15	15	15	15
Graph 38	4	4	4	4	4	4	4	4	4
Graph 39	8	8	8	8	8	8	8	8	8
Graph 40	3	3	3	3	3	3	3	3	3

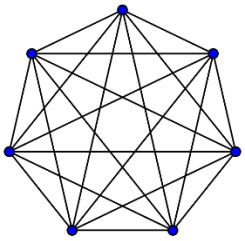
Appendix B

Minimum number of edges for 4 vertices.



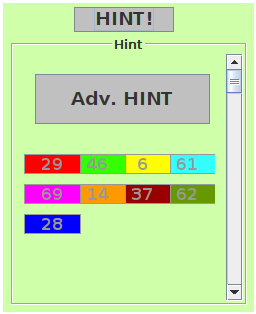
Appendix C

The complete graph with 7 vertices.



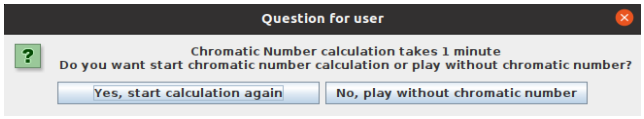
Appendix D

Screenshot of the regular Hint representation.



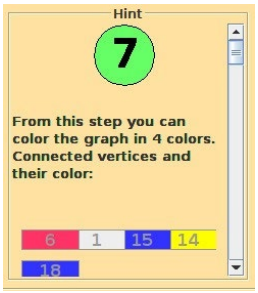
Appendix E

Screenshot of the warning representation.



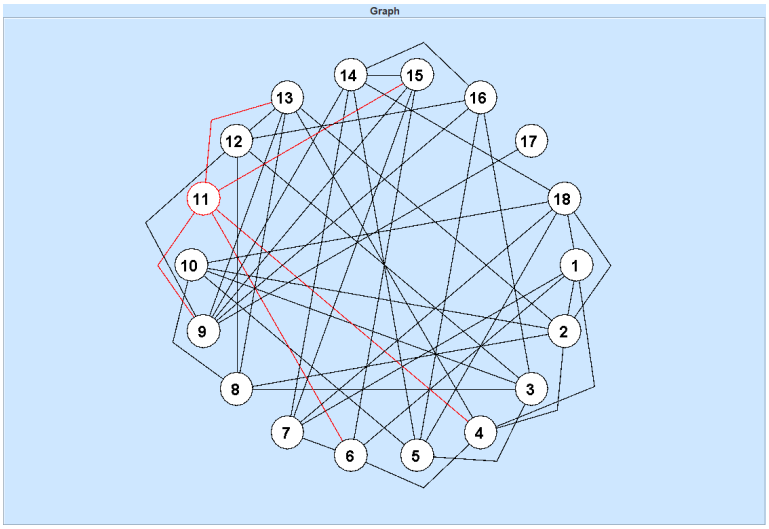
Appendix F

Screenshot of the advanced Hint representation.



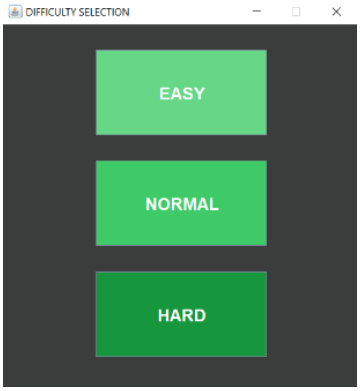
Appendix G

Screenshot of the graph display.



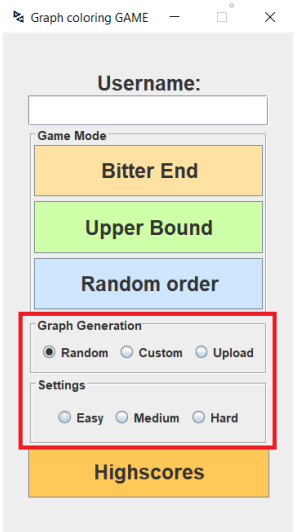
Appendix H

Best Upper Bound's Time Difficulty Selection Window.



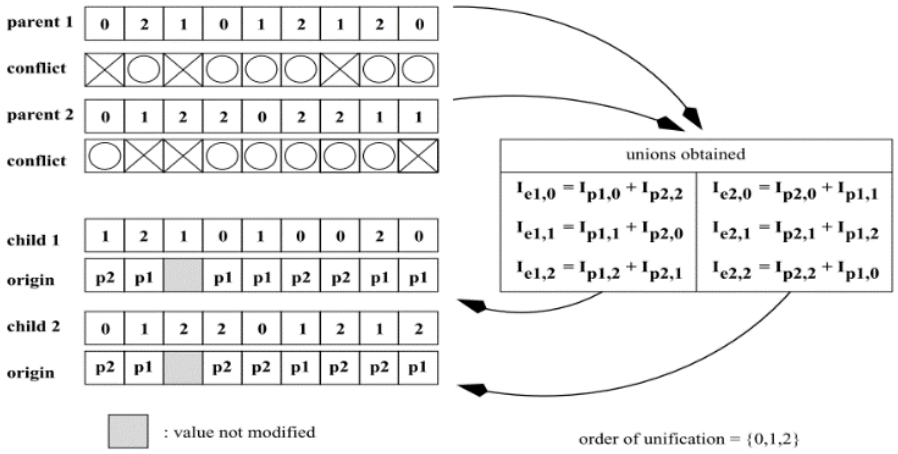
Appendix I

"Random" Graph Generation Difficulty Selection Panel



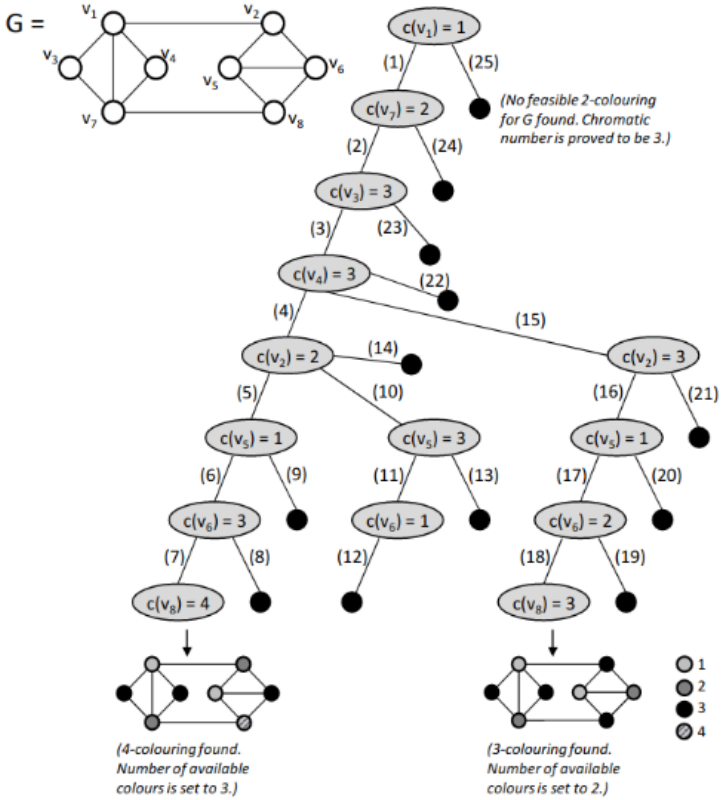
Appendix J

UIS crossover (image from [10])



Appendix K

Backtracking process (image from [5]).



Appendix L

Global process overview: Graph Coloring Game.

