Graph Colouring

Project 1-1, Group 20

Department of Data Science and Knowledge Engineering, Maastricht University

B. Caissotti di Chiusano, R. Carioni Porras, Y. Damoiseaux, E. Ozturk, T. Smit, K. Ek

Received January 19, 2021

Abstract

The main aim of this paper is to help improve calculation speeds on the calculation of the chromatic number of a graph, which would help to solve real-life problems. Finding the chromatic number of a graph is an NP-complete problem. For that reason this research focuses on finding the smallest interval of the chromatic number with lower and upper bounds. The problem was approached with different methods such as graph decomposition, recognizing simple graph topologies, solving the Max-Clique problem for the calculation of the Lower Bound and Reverse Backtracking, Greedy, RLF and Genetic algorithms for the Upper Bound. The chromatic number of small graphs and the ones with special cases get calculated instantly, whereas, to a certain extent, large graphs get calculated in a reasonable amount of time. Given that computers nowadays cannot solve NP-complete problems (even Quantum computers), this study shows that giving good instructions to the computer, with a multi-algorithmic approach and optimized code, allows us in some cases to reach the result even for graphs that have high complexity.

Table Of Contents

1 Introduction

The term Graph corresponds to objects, known as vertices, connected to each other by edges. This section introduces the fundamental concepts of the graph colouring problem and is also used as a gateway to present the algorithms discoursed throughout this report.

Graph colouring is the assignment of labels, or colors, to parts of a graph subject to bound restrictions. This colouring technique is fundamental for colouring the vertices of a graph such that two adjacent vertices cannot be given the same color, also referred to as vertex colouring **(see section 1.2)** and underpins the concept of Chromatic Number **(see section 1.2)**. Graph colouring was first used in the form of planar graphs as a map colouring technique in demarcation of England and France[1]. Since then, the area of utilization has undergone a significant change as it has an implementation to an extensive variety of complicated problems, such as scheduling, chemistry and Sudoku [2].

Finding the Chromatic Number of a Graph G ($\chi(G)$) has given rise to numerous algorithmic implementations, some of which will also be discussed throughout this report. Notable examples of these include the *XRLF* algorithm, a heuristic and color sequential algorithm presented by Leighton, Johri and Matula, or the *DSATUR* algorithm by Brelaz and its improved Branch-and-bound version[3]. Some exact algorithms have been implemented, and (according to Thomas J.SAGER et. al.) these "*generally fall into four categories: vertex sequential, color sequential, dichotomous search, and integer linear programming*"[3]. Overall, vertex sequential algorithms that dynamically reorder vertices have been shown to perform better. Nonetheless, it is important to note that Graph Coloring remains considered an NP-hard problem, and because of the time needed to find the chromatic number of larger and more complex graphs, these are usually just exposed to heuristic algorithms.

This report is going to explain and outline the methodology and implementation of different graph colouring algorithms. The *Greedy*, *Reverse Backtracking*, *Recursive-Large-First and Genetic* Algorithms for finding an accurate Upper Bound **(see section 2.3)** and a *Maximum Clique* Algorithm for computing a Lower Bound **(see section 2.2).**The algorithm implemented for the exact calculation of the $\chi(G)$ is discussed in **section 2.4** and a brief outline of the program can be found in **section 2.1.** Moreover, **Section 2.4.2** includes the application of Graph Decomposition with means of reducing the overall problem size. Each section will be equipped of an introductory paragraph which will provide an outline of the text that follows. Lastly, the aim of this report is to investigate its Problem Statement, further explained in **Section 1.2.**

For reference **Appendix A** contains a list of the abbreviations used throughout the report.

1.1 The Chromatic Number and Vertex Colouring

The terminology of vertex colouring dates back to map colouring and can be defined as a type of labeling a graph using colours such that two adjacent colours cannot be assigned the same colour Vertex colouring may cover several sub-terms, for example, *k-colouring,* a colouring utilizing at most k colours, and *chromatic number*[4].

The term of chromatic number is emerged within the term of vertex colouring. It can be defined as the minimum number of colours that are necessary for colouring any graph, such that any two adjacent vertices cannot be assigned the same colour[5]. The arrangement and amount of the vertices plays a significant role for finding the chromatic number, however the logic for calculating the minimum colours for colouring a graph is the same.

1.2 Problem Statement

This report will investigate, analyse and test numerous algorithms with the aim of answering the following problem statement: "*Can we implement a collection of algorithms that can accurately calculate the chromatic number of a graph in a reasonable time?*"

1.2.1 Research Questions

To fully tackle the problem statement of this report, the following research questions will be investigated and answered:

1. How could Graph Decomposition improve the performance of our algorithms?
2. To what extent does the exact algorithm calculate the chromatic number of a graph when starting from the lower bound/upper bound?
3. How efficient are Genetic, RLF, Reverse Backtracking and Greedy Algorithms when it comes to the calculation of an Upper Bound?

2 Method

2.1 Section Overview

The proposed approach for the calculation of the $\chi(G)$ is a combination of different methods. Algorithms of low computational complexity are executed first, if the $\chi(G)$ has not been found, algorithms with higher computational complexity are run.

1. The algorithm starts by checking if the graph can be classified as one of the trivial cases from the graph topology **(see section 2.4.1)**, in case it is identified as one of these, the $\chi(G)$, *LB* and *UB* are calculated immediately and returned to the user. In case the topology of the graph is not recognised as one of the trivial cases, two processes start simultaneously: on one side the calculation of the *UB* **(see section 2.3)** and *LB* **(see section 2.2)** is executed and if these values coincide, the program returns the value as the $\chi(G)$ to the user. On the other side, the process of graph decomposition is executed.
2. The Graph decomposition process is made up of various different sub-algorithms. First the algorithm checks if there are disconnected components **(see section 2.4.2.1)** in the graph and if so, a division is made, resulting in y subgraphs, from this step on, subgraphs are treated as independent graphs with their own $\chi(G)$, *LB* and *UB*. The next step executed consists in the identification of Bridges **(see section 2.4.2.4)** and if found, a decomposition process is executed. This is followed by the identification of Articulation Points **(see section 2.4.2.3)** and if possible another decomposition process is executed resulting in more subgraphs.
3. Once this subgraphs cannot be decomposed further, a check of trivial graph topologies is run in each subgraph, if not found, the calculation of the *UB* and *LB* is executed and if these

values match, the $\chi(G)$ is returned immediately as being equal to them. If the values do not match, the Exact Algorithm **(see section 2.4)** is run. This is executed last because it is the most computationally demanding. To try and reduce the time needed we implemented vertex elimination **(see section 2.4.2.2)** everytime a number of colours is tried as a solution for the graph, the effectiveness of this is discussed in **section 6.0.**

4. Finally, after each subgraph has been assigned with a $\chi(G)$, the highest of these values is returned to the user as the $\chi(G)$ of the entire graph.

For reference, **Appendix C** displays a flowchart of the program.

2.2 Lower Bound

One of the constraints applied to the exact algorithm was a lower bound. The purpose of its output is to learn whether the same or a greater amount of colours are needed to find the chromatic number and colour a certain graph. This bound corresponds to the Maximum Clique in a graph, an NP-Complete problem. To tackle this, the following algorithm described will make use of recursion and pruning [6]. To avoid repetition, a Clique remains only defined in **Section 2.4.1.3**.

It is important to note that an example of more detailed version/s of this algorithm is presented by Patrick Prosser, from the University of Glasgow, in his article “Exact Algorithms for Maximum Clique: A Computational Study”[6]. For reference, **Appendix B** displays a flowchart of the LB algorithm, while **Section 3.1** includes its pseudocode. Overall, to find the maximum clique of a graph, the algorithm iterates through all vertices of the graph, analyzing each of them one by one. For every vertex, we check its adjacent vertices, also iterating through all of them with the use of a recursive call, until a clique containing the original vertex analysed is found and saved. To speed up this process, the algorithm is able to understand if the clique that will result from the specific search will be smaller than the current maximum clique saved. If that is the case, the search for that vertex and its adjacent vertices is abandoned. This is known as pruning, or in other words, the action of reducing the size of the search tree.

2.3 Upper Bound

Another technique to approximate the exact $\chi(G)$ is to find an upper bound. The output of this algorithm will be a viable colouring of the graph, although it may not be the most optimal one. At first we implemented two different algorithms, the Greedy algorithm and the RLF algorithm, that will find the initial upper bound. In our final program we only use the RLF algorithm for this purpose. In the discussion (for reference, **section 6**) it will be discussed why we made this decision. For improving this initial *UB* we use on one side the Genetic Algorithm and on the other side the Reversed Backtracking Algorithm.

The final upper bound is chosen by selecting the lowest answer received from these various algorithms.

2.3.1. Greedy algorithm

Greedy algorithms can be a useful tool to calculate an *UB* to the chromatic number because the algorithm generally provides very fast results and it has a simple implementation. However, these results are not always the most optimal solution [7].

The algorithm finds the *UB* by giving all the vertices the first available colour. A modification that we did to the original implementation consists in a change of the order in which it loops through the vertices, based on the degrees. It starts at the vertex with the highest degree and ends at the vertex with the minimum degree. A colour is considered available if it is not used to colour one of the adjacent vertices. After colouring all of the vertices, the algorithm returns the amount of colours used to colour the graph. For reference, **Appendix G** includes a more detailed description of the algorithm itself, including pseudocode of its implementation.

2.3.2 Recursive-Large-First (RLF)

The RLF algorithm is, next to the greedy algorithm, an algorithm that is not the hardest to implement. The global idea of this algorithm is building color classes. The first vertex that is added to the color class is the vertex with the most uncoloured vertices. After the first vertex the vertices are chosen based on how many uncoloured neighbors they have, that cannot be assigned to the current color class because of adjacency. This process continues until every vertex has a valid colour assigned. The resulting *UB* is the amount of color classes made. [8]

For a more detailed explanation about the implementation see **Section 3.2.1.**

2.3.3 The Genetic Algorithm (GA)

As mentioned before, a Greedy or RLF algorithm will not always provide the most optimal solution. Therefore, we used a GA to increase the accuracy of our *UB*. On execution, a population of individuals is created, where each individual contains a colouring of the graph G with $\omega(G) = UB - 1$. After the initialization phase, the algorithm can be branched into four sections: assignment of fitness score, selections of individuals to reproduce, crossover and mutation [9]. All of these branches are executed repeatedly until a new solution is found. The GA will keep trying to find new *k*-colourings with decreasing *k*, until it has reached the lower bound. For reference, **section 3.2.2** includes a more detailed description of the methods used.

2.3.3.1 Fitness function

In the fitness function, all individuals are assigned a fitness value (a real number between 0 and 1). This value represents the similarity of the individual to the targeted individual. In the case of this graph colouring research, the fitness value will be calculated based on the amount of incorrect edges (the edges that connect two of the same coloured vertices) in the colouring. Based on this fitness value the individuals are sorted from higher to lower values.

2.3.3.2 Selection function

To reach the most optimal result, the algorithm uses Elitist selection. A fixed percentage of the fittest individuals are selected, where the word fittest, in this case, refers to the most correct solutions. The selection function in this GA selects the top 5% of the solutions that will be used in the crossover phase.

2.3.3.3 Crossover function

After the selection process, the selected individuals reproduce with each other to create a completely new generation of offspring. This is done by the so-called "crossover function". There exist several types of crossover functions [10], however this GA implements the uniform crossover. In uniform crossover, the colour of each vertex will be chosen from one of the two parents at random. Both parents have equal probability to be selected from.

2.3.2.4 Mutation function

To increase the uniqueness of the offspring, and the probability of obtaining an individual with fitness 1 we also implemented a mutation function. The mutation function has a chance of changing a colour of some vertex into a different colour. Mutation can happen at different rates. During this research we used a mutation rate of 1% for every vertex.

2.3.4 Reverse backtracking (RB)

RB is used for improving the initial *UB* of the RLF Algorithm. On $\sigma(G)$ we apply backtracking to determine if it is possible to recolour every $v_{max} \in max(\sigma(G))$ with $\omega(G) = UB - 1$. Each time an appropriate solution is found this process is repeated with $max(\sigma(G))$ and $\omega(G) = \omega(G) - 1$, until no feasible solution is found, which leads to $UB = \omega(G) + 1$.

Because backtracking is extremely time-consuming, the amount of vertices we try to backtrack needs to be as low as possible, while still achieving an improved result. Therefore we have two levels representing the amount of backtracking that is being done. The first level backtracks $\alpha(v_{max})$ including v_{max}. When this does not lead to a solution, the second level of backtracking is executed. In addition to the first level backtracking, the second level backtracking includes $\alpha(\alpha(v_{max}))$. If neither leads to a solution, *UB* is unchanged.

Since a vertex can have many neighbours, which could again lead to long execution times, we created 4 slightly different algorithms that all have their own way of backtracking the different levels. The first algorithm is the standard algorithm which at both levels first resets every colour of $v \in \alpha(v)$, after which it tries to find a solution with backtracking. The second algorithm is different at the first level. It does not wait with backtracking till every colour of $v \in \alpha(v)$ is reset, instead, it accumulates. In other words, first it backtracks $v_1 \in \alpha(v_{max})$ then $v_{1,2} \in \alpha(v_{max})$ and so on until $v_{1,2,\ldots,size(\alpha(v_{max}))} \in \alpha(v_{max})$. For the third algorithm the first level is the same as the first algorithm, however, the second level is the same as the first level of the second algorithm. The fourth algorithm has the implementation of the first algorithm. The only difference is the order in which they take $v_{max} \in max(\sigma(G))$. The first algorithm starts at the vertex that has the maximum degree in $max(\sigma(G))$, and ends with the vertex that has the minimum degree in $max(\sigma(G))$, while the fourth algorithm starts at the vertex that has the minimum degree in $max(\sigma(G))$, and ends with the vertex that has the maximum degree in $max(\sigma(G))$. [11]

For reference, **Appendix D** includes pseudocode for the first, standard, version of the algorithm.

2.4 Backtracking Exact Algorithm

This algorithm is a Greedy algorithm (**See 2.3.1**) that uses backtracking, pruning and Depth-First Search, whenever the greedy algorithm cannot find an appropriate solution with $\omega(G)$.

The algorithm starts by modifying the graph, eliminating vertices that have less than $\omega(G)$ edges, to this new graph the algorithm starts trying to colour it with $\omega(G) = UB - 1$ and each time an appropriate colouring is found for the graph, it repeats the process updating $\omega(G)$ with one less colour and the graph. This process keeps going until the graph cannot be coloured, which results in $\chi(G) = \omega(G) + 1$. The use of backtracking causes the algorithm going forward, if the current set of assigned colors to the vertices is legitimate, and going backwards if it appears that the current set of assigned colors is not legitimate. Going backwards causes decolouring the previous coloured vertex, and recolouring it with another possible colour. If there is not another possible color available, it keeps going backwards until a vertex is found that can have another colour assigned to it. At some point there will be a vertex with a new colour assigned to it. This causes the algorithm going forwards again. These steps are repeated until there is a legitimate set of coloured vertices found, or until the vertices of a (sub)graph tried every colour with in the end no legitimate solution being found. Lastly, pruning is used to decrease the size of the search tree by removing the branches that are not feasible in advance, because of adjacency between the vertices and their assigned colours. [12]

For reference see **Section 3.3**, which includes pseudocode.

2.4.1 Simple Graph Topologies

In some cases by looking into graphs topologies, one can quickly determine the $\chi(G)$ of a graph because there are certain patterns which are easy to identify and to solve. The algorithms needed to find the $\chi(G)$ of these are of low complexity. Six different types of graphs are checked in the program, these are: Bipartite, Cycle, Wheel, Complete and No Edge.

2.4.1.1 Bipartite

The vertices of a bipartite graph can be separated into two sets, *V1* and *V2*, such that edges only exist between vertices in *V1* and *V2* and not inside the same set [13]. Therefore, it has a $\chi(G) = 2$.

The algorithm implemented starts by assigning a vertice into *V1* and looping through all of the edges searching for the vertices that are connected to this and assigning them to the opposite set, *V2*, the process is repeated for all vertices until the entire graph is classified. However, if there exists a vertice that can not be placed in either of the two sets, without breaking the condition of not having the same colour between adjacent vertices, the algorithm will stop running and it would return that the graph is not Bipartite.

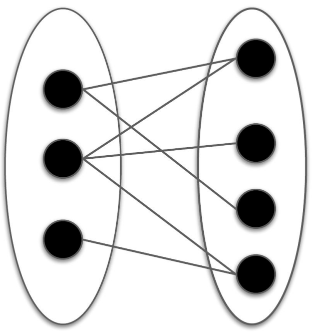


Figure 1: Bipartite Graph

2.4.1.2 Cycle and Wheel

A cycle graph (see **figure 2**), denoted by C_n, where $n \geq 3$, comprises a set of vertices $V = \{v_1, ..., v_n\}$ and a set of edges E of the form $\{\{v_1, v_2\}, \{v_2, v_3\}, ..., \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. If n is even, $\chi(G)$ is equal to 2. Otherwise, if n is odd, $\chi(G)$ is equal to 3. [13]

The algorithm we implemented loops through every vertex and their corresponding edges to check whether they are all connected to exactly two other vertices, because this, as stated above, is a characteristic of a cycle.

Wheel graphs with *n* vertices (see **figure 3**), denoted by W_n, are obtained from the cycle graph C_{n-1} by adding a single extra vertex v_n together with the additional edges $\{v_1, v_n\}, \{v_2, v_n\}, ..., \{v_{n-1}, v_n\}$. In case of a wheel graph there are two possible outcomes for $\chi(G)$. If n is even, $\chi(G)$ is equal to 4. Otherwise, if n is odd, $\chi(G)$ is equal to 3. [13]

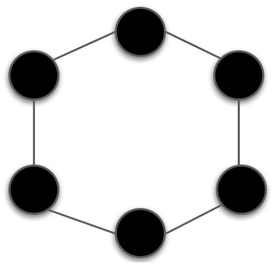


Figure 2: Cycle Graph

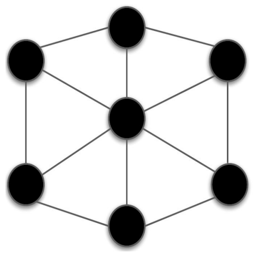


Figure 3 Wheel Graph

To determine if a graph is a wheel, we implemented an algorithm that first checks if there is an universal vertex in the graph. If there is, it could be a wheel if the universal vertex is surrounded by a cycle. To determine this we do exactly the same as the cycle algorithm, however, we exclude the universal vertex and his edges, while this does not belong to the cycle.

2.4.1.3 Complete

A graph is Complete if every pair of distinct vertices is connected by a unique edge. These graphs have $\frac{n(n-1)}{2}$ edges. The $\chi(G) = n$. [13]

The implemented algorithm checks whether a graph is complete or not by seeing if the graph has the same number of edges as the formula previously shown.

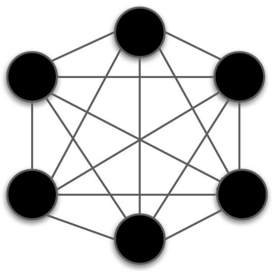


Figure 4: Complete Graph

2.4.1.4 No Edge

A non Edge graph is an edgeless graph. These have $\chi(G) = 1$.

2.4.2 Graph Decomposition

When trying to solve the Graph Colouring problem, decomposing the graph into smaller parts, without interfering with the $\chi(G)$, reduces the complexity of the problem by decreasing the size of the original problem [12]. For certain graphs time can be saved as this increases in the calculation of the $\chi(G)$ as the number of vertices and edges rises.

Four techniques were implemented in this investigation to reduce the size of the problem: Separating Disconnected graphs, Vertices Elimination, Bridge Removal and Separating graphs based on their Articulation Points.

2.4.2.1 Disconnected Graphs

Two graphs are said to be disconnected if they do not have edges that connect them. [12] If one graph contains disconnected parts we can divide them and treat them as independent graphs with their own *LB*, *UB* and $\chi(G)$, and once that all have been calculated, the higher values are chosen for the original graph.

The algorithm implemented starts by checking to which vertices a specific vertex is connected and adding them into a subgraph, in the same way it checks for all of the other vertices, until all of the edges have been considered and vertices have been added. If there are vertices from the original graph which have not been included in the subgraph, it is because there are disconnected components, a new subgraph is created and the process is repeated.

2.4.2.2 Vertices Elimination

A graph can be simplified by reducing its number of connections. When trying to solve a graph with *k* colours, we can eliminate all vertices with degree less than *k* because vertices with fewer than *k* adjacent vertices will always have a feasible colour from the set $\{1, ...k\}$ to which they can be assigned [12]. The algorithm is executed in the Exact Algorithm **(see section 3.3)** and it reduces the size of the graph everytime that *k* colours are tested to be a possible solution for a $\chi(G)$.

2.4.2.3 Articulation Point (AP)

If the elimination of a vertex splits a graph into at least two subgraphs then that vertex is called an articulation point [14], as shown in **Figure 5.**

The approach taken to implement the algorithm is based on a modification of Chang and Huang [15] and Chaudhuri algorithms [14]. The main part of it consists in a Depth-First Search which is executed to traverse the graph and generate information about each vertex, such as time discovered, time when the search on that vertex is finished, the parent, the status of the search on that vertex (not discovered, discovered or finished) and the low value. The low value is initially defined for each vertex as the time discovered and then updated to be the minimum value between the lows of its adjacency vertices.

A vertex v is an AP if it is the root (it does not have a parent) of a DFS tree and it has at least 2 children **(case 1: see section 3.4.1)**, or if it has a child such that no vertex in the subtree of that child is connected to one of the ancestors of v **(case 2: see section 3.4.1).** Based on this, v is defined as an AP if it has a descendant *d,* such that: $LOW\ (d) \geq TimeDiscovered(v)$. [14]

For reference, **Section 3.4.1** includes pseudocode of the algorithm.

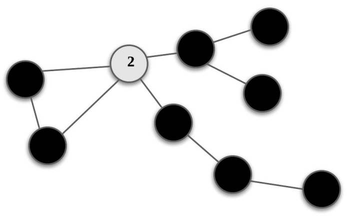


Figure 5: Articulation point at vertex 2.

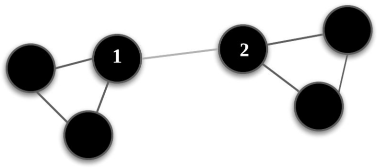


Figure 6: Vertices 1 and 2 form a Bridge.

2.4.2.4 Bridge Removal

If the elimination of an edge E disconnects a graph into a least two subgraphs then E is called a Bridge [16]. **Figure 6** illustrates this.

The algorithm implemented is based on a modification of Tarjan [17] and Chaudhuri [18] algorithms and to a certain extent it follows the same logic as the Articulation Point algorithm described in **Section 2.4.2.3**. However, it differs in the criteria considered when classifying Bridges (before Articulation Points). Consider the vertices *v* and *d*, which form and edge E : if *d* has not be discovered yet, E is a Bridge if $LOW\ (d) \geq TimeDiscovered(v)$.[19]

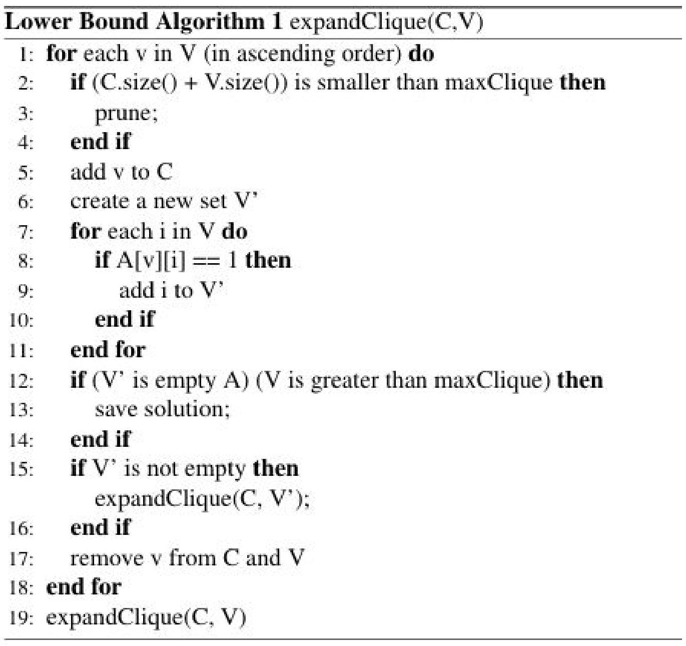
For reference, **Appendix F,** includes pseudocode of the algorithm.

3 Implementation

The following section briefly explains and displays the pseudocode of numerous algorithms (previously explained in **Section 2**) implemented in order to answer this report's problem statement and research questions. It is important to note that these algorithms were implemented using Java and with the same progression as the steps previously described in **Section 2.1**. This section will specifically display the pseudocode for the Lower Bound (**see 3.1**), the GA (**see 3.2.2**) and RLF(**see 3.2.1**) as Upper Bounds and the Articulation Point (**see 3.4.1**) identification for graph Decomposition.

3.1 Lower Bound

The class constructor of the Lower Bound (LB) algorithm takes in two parameters: the number of vertices and an array of edges connecting these vertices. Consequently, a corresponding adjacency matrix is generated and each vertex is given a degree. The degree represents the number of edges a vertex has. The next step is to actually search for the maximum clique of the graph. All vertices are added to a set V. A set C is generated as well, this will eventually contain the maximum clique of the graph. Figure 6 displays the pseudocode of the following method that is executed:



Lower Bound Algorithm 1 expandClique(C,V)

```
for each v in V (in ascending order) do
    if (C.size() + V.size()) is smaller than maxClique then
        prune;
    end if
    add v to C
    create a new set V'
    for each i in V do
        if A[v][i] == 1 then
            add i to V'
        end if
    end for
    if (V' is empty A) (V is greater than maxClique) then
        save solution;
    end if
    if V' is not empty then
        expandClique(C, V');
    end if
    remove v from C and V
end for
expandClique(C, V)
```

Figure 7: The ExpandClique Method

3.2 Upper Bound

3.2.1 RLF Algorithm

The main components of this algorithm are two while loops and three lists that contain vertices. The algorithm begins with adding every vertex to the first list U, which contains the uncoloured vertices. It enters the fist while loop by incrementing the current colour, c, and taking the

$v \in U$ with the highest degree between vertices in U. This v is removed from U and added to the list C, which contains the coloured vertices, causing this vertex obtaining colour c. Every neighbour, adjacent vertex, of this v is removed from U and added to the list W, which contains the neighbours of a just coloured vertex. In the second while loop the vertex from U is taken with the highest degree between vertices in W. Again, this vertex is removed from U and added to C with obtaining colour c. All the neighbours are removed from U and added to W. This process is repeated until there are no vertices in U left. In other words, every vertex has a colour assigned or is a neighbour of a just coloured vertex. The next crucial step is to copy everything from W to U, and repeat the complete process because of the first while loop, until, even after copying W to U there is no vertex left in U. Which means that every vertex is in C and has a colour. [8]

3.2.2 Genetic Algorithm

While trying to approximate the $\chi(G)$, the algorithm keeps executing four different functions, which are described next.

3.2.2.1 Fitness function

The fitness of this GA is based on the amount of incorrect edges in the colouring. Figure 8 describes the workflow of the function.

```
Genetic Algorithm 1 fitness function
for each individual in generation do
    count amount of wrongEdges;
end for
if wrongEdges = 0 then
    solution found
else if wrongEdges = 1 then
    fitness = 0.75
else
    fitness = 1/wrongEdges
end if
```

Figure 8: Pseudocode for the fitness function of the GA.

3.2.2.2 Selection function

The selection procedure in this GA follows an Elitist approach. After sorting the individuals on their fitness values, the selection function takes out the top 5%. It will then create 2 lists, with the same size as the initial population, filled with these top individuals. These lists are referred to as the 'parents', and are then used for the crossover phase.

3.2.2.3 Crossover function

The crossover function used in this GA is uniform crossover. To create an offspring, for every vertex a colour is chosen from one of the parent lists. This will happen at random, with an equal probability for either parent. Crossover is being repeated until we have a new generation of individuals.

3.2.2.4 Mutation function

The mutation function in this GA increases the uniqueness of the individuals. The function loops through all the individuals that were just created in the crossover phase. Every time having a small chance to change a colour to a different random colour from the set of available colours.

3.3 Exact Algorithm

Backtracking Exact Algorithm 1 recurse(E)

```
for each edge in E do
    if first vertex of edge has no colour then
        for each c in colours do
            if this colour is valid according to adjacency then
                assign colour c to the vertex;
                if recurse(E) returns false then
                    reset the colour of the vertex;
                end if
            end if
        end for
        if vertex still has no colour then
            return false;
        end if
    end if
    if second vertex of edge has no colour then
        for each c in colours do
            if this colour is valid according to adjacency then
                assign colour c to the vertex;
                if recurse(E) returns false then
                    reset the colour of the vertex;
                end if
            end if
        end for
        if vertex still has no colour then
            return false;
        end if
    end if
end for
return true;
```

Figure 9: Pseudocode for the recurse method of the Exact algorithm.

3.4 Graph Decomposition

3.4.1 Articulation Point

The information considered for each vertex *(v) i*s discussed in **section 2.4.2.3**. About the status of the search on *v* we considered the initial status which is not discovered= 0, discovered= 1 and finished= 2.

ArticulationPoint Identification 1 Visit(v)

```
time +=1
v.status = 1
v.low = v.timeDiscovered = time
v.children = 0
for e in v.adjacentVertices do
  if e.status == 0 then
    v.children +=1
    e.parent = v
    visit(e)
    v.low = min(v.low, e.low)
    if (v.parent == 0) and (children > 1) then
      // CASE 1: v is a root node
      v.articulationPoint = true
    else if (v.parent != 0) and (e.low >= v.timeDiscovered) then
      // CASE 2: The subtree of e has no connection to parent of v
      v.articulationPoint = true
    end if
  else if v.parent != e then
    v.low = min(v.low, v.timeDiscovered)
  end if
end for
v.status = 2
time +=1
v.timeFinished = time
```

Figure 10: Pseudocode for the Visit method of the Articulation Point.

4 Experiments

The experiments were executed for 20 graphs from **Project 1-1 Phase 3** (see **Appendix H**), all with a limit of time of 4 minutes, in a Windows 10 Home 64-bit operating system with specs: Intel Core I7 10750H 2.60 GHz 6 cores and 16384MB RAM. The research questions and the criteria considered for each experiment are discussed next.

1) How could Graph Decomposition improve the performance of our algorithms?

In this experiment the $\chi(G)$ was computed in six different ways, including the following criteria:

1. Disconnected graph Algorithm only (**see section 2.4.2.1**).
2. Vertices elimination Algorithm only (**see section 2.4.2.2**).
3. Articulation Points Algorithm only (**see section 2.4.2.3).**
4. Bridge removal Algorithm only (**see section 2.4.2.4**).
5. All four Algorithms: disconnected graph, vertices elimination, articulation points and bridge removal.
6. No decomposition at all.

2) How efficient are Genetic, RLF, Reverse Backtracking (1,2,3 and 4) and Greedy Algorithms when it comes to the calculation of an Upper Bound?

In this experiment graph decomposition was not used because we wanted to compare the *UB* algorithms raw, meaning, without the influence of other algorithms. In addition to this, all of the RB algorithms and the GA algorithm tried to improve the initial upper bound of the Greedy Algorithm. One of the goals of this experiment was to compare the performance of the Greedy Algorithm and the RLF Algorithm, to determine which algorithm is more efficient to be used as the initial *UB*. Another goal was determining the result improvements of the RB algorithms and the GA algorithm compared to time.

3) ***To what extent does the exact algorithm calculate the chromatic number of a graph when starting from the lower bound/upper bound?***

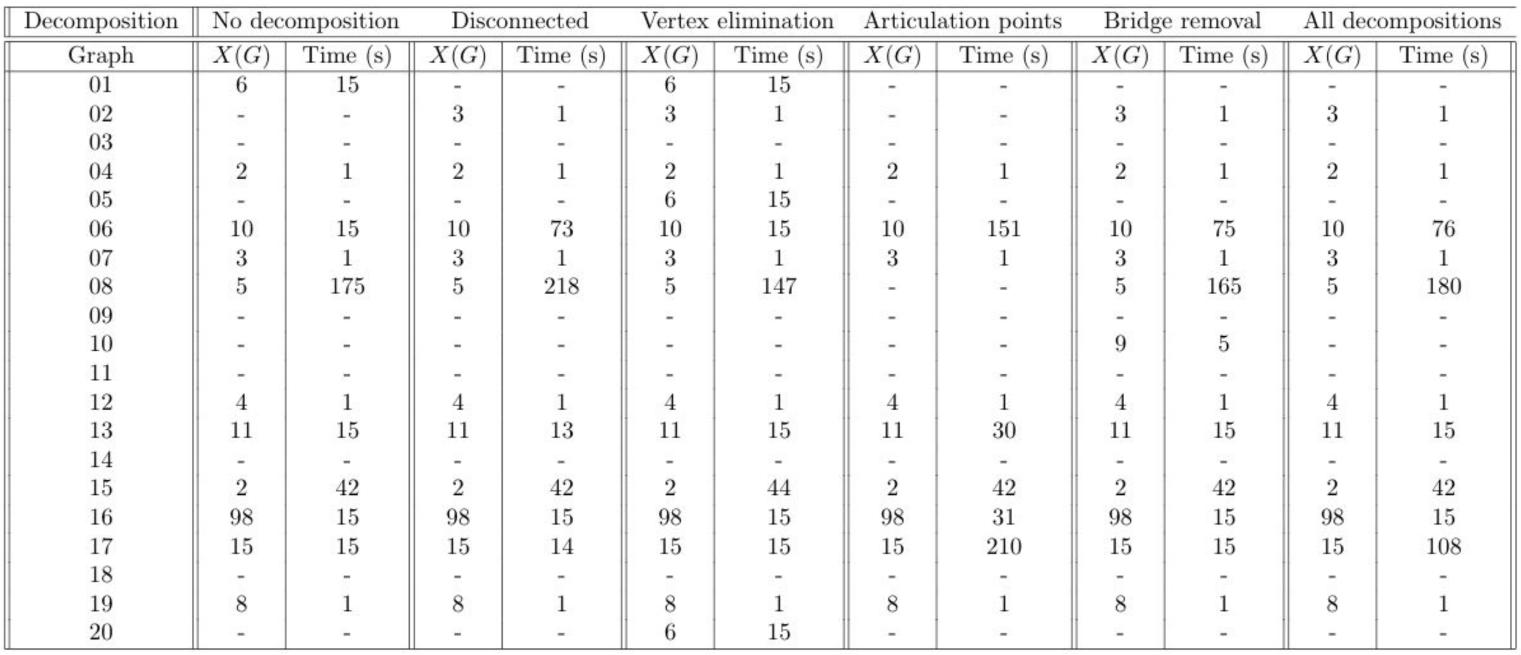
To answer this question and see which implementation gives a faster result when calculating the $\chi(G)$, the Exact Algorithm was executed and measured based on how long it took to get to the chromatic number. The modifications made to this were the following:

1. Start the search at the *UB* and decrease until the graph could not be solved with the number of colours given.
2. Start from the *LB* and increase the number of colours until it could be solved.

5 Results

For the characteristics of the graphs (number of vertices, edges and density) see **Appendix H.**

Table 1*: Time used to calculate the* $\chi(G)$ *with different Graph Decomposition methods.*



Decomposition	No decomposition		Disconnected		Vertex elimination		Articulation points		Bridge removal		All decompositions	
Graph	$X(G)$	Time (s)	$X(G)$	Time (s)	$X(G)$	Time (s)	$X(G)$	Time (s)	$X(G)$	Time (s)	$X(G)$	Time (s)
01	6	15	-	-	6	15	-	-	-	-	-	-
02	-	-	3	1	3	1	-	-	3	1	3	1
03	-	-	-	-	-	-	-	-	-	-	-	-
04	2	1	2	1	2	1	2	1	2	1	2	1
05	-	-	-	-	6	15	-	-	-	-	-	-
06	10	15	10	73	10	15	10	151	10	75	10	76
07	3	1	3	1	3	1	3	1	3	1	3	1
08	5	175	5	218	5	147	-	-	5	165	5	180
09	-	-	-	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	9	5	-	-
11	-	-	-	-	-	-	-	-	-	-	-	-
12	4	1	4	1	4	1	4	1	4	1	4	1
13	11	15	11	13	11	15	11	30	11	15	11	15
14	-	-	-	-	-	-	-	-	-	-	-	-
15	2	42	2	42	2	44	2	42	2	42	2	42
16	98	15	98	15	98	15	98	31	98	15	98	15
17	15	15	15	14	15	15	15	210	15	15	15	108
18	-	-	-	-	-	-	-	-	-	-	-	-
19	8	1	8	1	8	1	8	1	8	1	8	1
20	-	-	-	-	6	15	-	-	-	-	-	-

***Table 2**, Time used to calculate the UB with different Algorithms and their respective outcome.*

Upper Bounds	Greedy		RLF		RB1		RB2		RB3		RB4		GA	
Graph	UB	Time/sec	UB	Time/sec	UB	Time/sec	UB	Time/sec	UB	Time/sec	UB	Time/sec	UB	Time/sec
01	8	0.037	6	0.127	7	1	7	1	7	1	7	0	6	54
02	3	0.042	3	0.316	-	-	-	-	-	-	-	-	-	-
03	7	0.038	5	0.081	-	-	-	-	5	38	6	0	5	95
04	3	0.051	2	1.077	-	-	-	-	-	-	-	-	-	-
05	10	0.041	7	0.199	-	-	9	1	-	-	-	-	-	-
06	11	0.035	11	0.082	10	1	10	1	10	1	10	0	10	166
07	3	0.034	3	0.059	-	-	-	-	-	-	-	-	-	-
08	7	0.033	5	0.031	6	1	6	1	6	1	6	0	6	1
09	12	0.037	12	0.009	-	-	-	-	-	-	-	-	-	-
10	9	0.046	9	1.416	-	-	-	-	-	-	-	-	-	-
11	16	0.037	11	0.032	12	2	11	205	11	100	12	0	11	26
12	4	0.037	4	0.053	-	-	-	-	-	-	-	-	-	-
13	11	0.035	11	0.038	-	-	-	-	-	-	-	-	-	-
14	5	0.044	4	0.432	4	1	4	1	4	1	4	0	4	28
15	7	0.309	-	-	-	-	-	-	-	-	-	-	-	-
16	98	0.055	98	0.072	-	-	-	-	-	-	-	-	-	-
17	15	0.036	15	0.071	-	-	-	-	-	-	-	-	-	-
18	5	0.044	4	3.538	4	1	4	1	4	1	4	0	4	65
19	8	0.033	8	0.015	-	-	-	-	-	-	-	-	-	-
20	3	0.034	3	0.034	-	-	-	-	-	-	-	-	-	-

***Graph 1**, (**<u>Appendix E</u>** for Raw Data): Time used to calculate the $\chi(G)$ starting from the UB vs LB.*

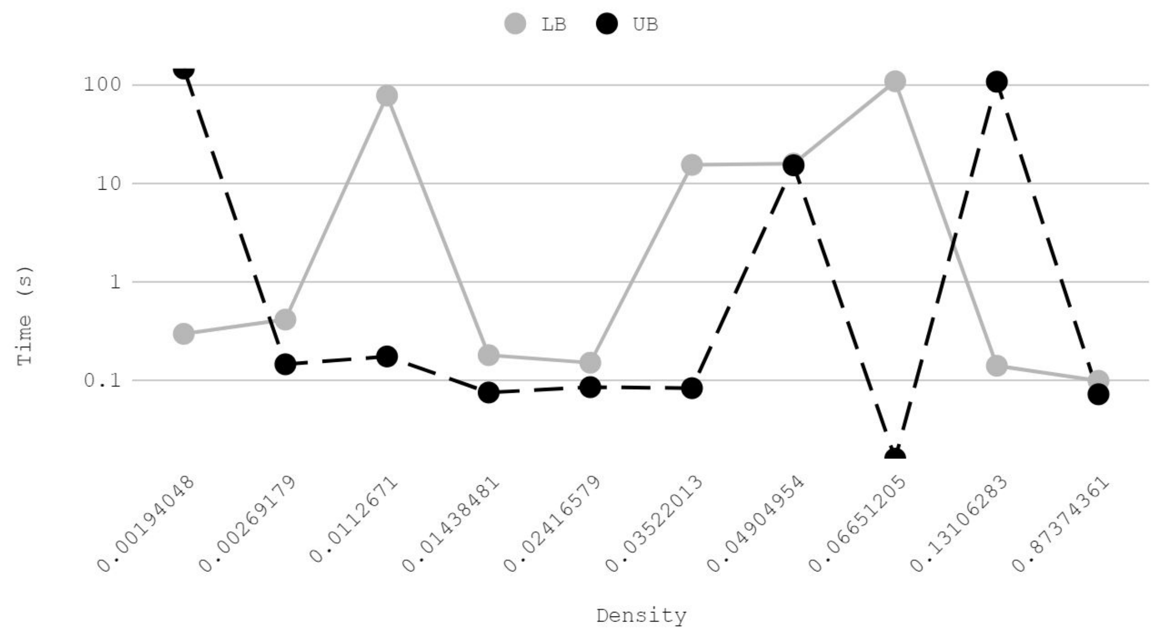
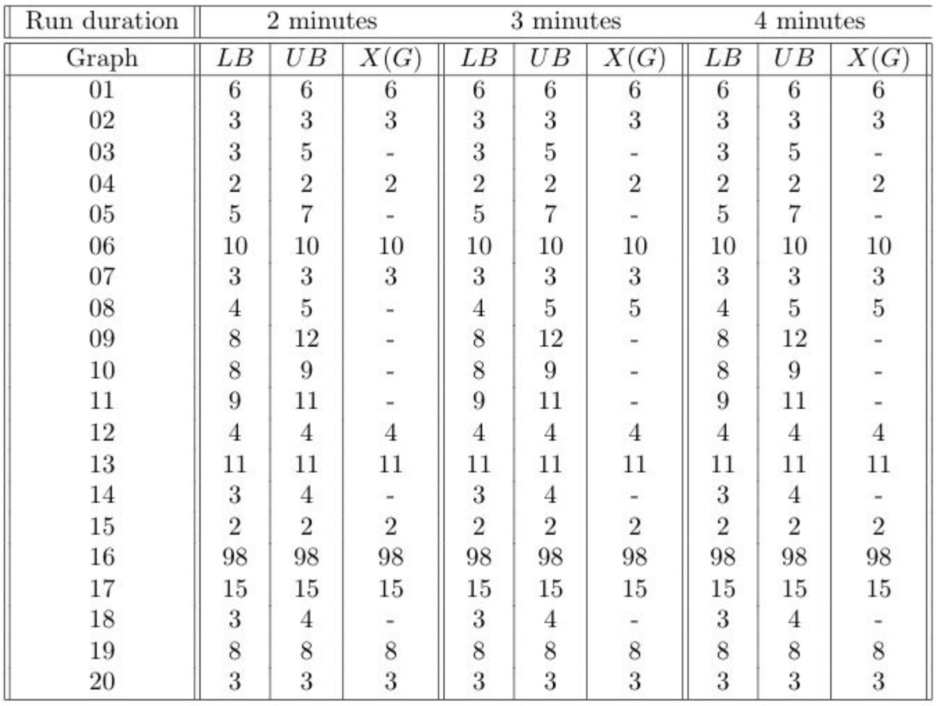


Table 3 *- Testing the final algorithm for different time limits.*



Run duration	2 minutes			3 minutes			4 minutes		
Graph	LB	UB	$X(G)$	LB	UB	$X(G)$	LB	UB	$X(G)$
01	6	6	6	6	6	6	6	6	6
02	3	3	3	3	3	3	3	3	3
03	3	5	-	3	5	-	3	5	-
04	2	2	2	2	2	2	2	2	2
05	5	7	-	5	7	-	5	7	-
06	10	10	10	10	10	10	10	10	10
07	3	3	3	3	3	3	3	3	3
08	4	5	-	4	5	5	4	5	5
09	8	12	-	8	12	-	8	12	-
10	8	9	-	8	9	-	8	9	-
11	9	11	-	9	11	-	9	11	-
12	4	4	4	4	4	4	4	4	4
13	11	11	11	11	11	11	11	11	11
14	3	4	-	3	4	-	3	4	-
15	2	2	2	2	2	2	2	2	2
16	98	98	98	98	98	98	98	98	98
17	15	15	15	15	15	15	15	15	15
18	3	4	-	3	4	-	3	4	-
19	8	8	8	8	8	8	8	8	8
20	3	3	3	3	3	3	3	3	3

6 Discussion

To answer our problem statement (**see section 1.2**) and to test the efficiency of our program we performed four experiments in order to answer our research questions. In this section their results are discussed.

Starting with the results from Experiment 1 **(see Table 1, section 5)**, it can be observed that with Bridge Removal only, we were able to calculate the highest number of $\chi(G)$ (13 results), whereas Articulation Points Removal showed the worst results, not only in time performance but also in the total number of $\chi(G)$ calculated (10 in total). It was found surprising that using all of the decompositions combined (last two columns of Table 1) did not lead to the best results. Similarly, the results obtained from not using graph decomposition at all, called our attention because they were expected to be the worst.

All Decompositions and Bridge Removal differ only in graph 10, and the same can be said when comparing Vertices Elimination and All Decompositions in graph 1. The fact that graph 1 could only be solved without No Decomposition and with Vertices Elimination led us to conclude that Vertice Elimination did not come as an issue when solving for the $\chi(G)$ (within the given time), but the problem lies in the order of the vertices and edges. These plays an important role when calculating $\chi(G)$, in fact, when more decomposition takes place, so does the reordering of vertices and edges.

It is known that by reducing the size of a graph, the complexity of solving $\chi(G)$ decreases. However, if the decomposition, for example, generates 2 graphs and one of these is almost similar in size to the original one, the reduction of the original problem is marginal. In this case the ordering of the vertices and/or edges plays an important role, in the same way as the order followed by the Exact

Algorithm. Based on this, we decided to keep the implementation of our algorithm as doing all of the possible decompositions in a graph.

Following with the results from Experiment 2 **(see Table 2, section 5)** one could say that the RLF algorithm outperformed the Greedy algorithm in most cases, either by being more time efficient or by giving more accurate results. Two exceptions to this were registered, firstly on graph 6, where RLF required more time, and secondly on graph 15, where Greedy gave an answer in 0.309 seconds, and RLF could not compute it within the 4 minutes given. An explanation for this last result could be that graph 15 has over 1.2 million edges and the RLF algorithm loops several times through all of the edges, especially since it is a bipartite graph. Other than graph 6, these results did not come as a surprise.

From the results we can see that the Genetic Algorithm improves 7 of the *UB*. For the ones it does not improve, we can see that either the initial *UB* is already accurate, or the density of the graph is high. The higher the density of the graph, the less possible colourings of the graph exist. Therefore, it takes a longer time to find the correct colouring.

When it comes to the results obtained from the RB1, RB2, RB3, RB4 and GA algorithms, one could see that they are very similar, with the exception that RB1 could only improve 6 of the inputs given by the Greedy algorithm, whereas the rest improve 7. Even though we, as stated above, decided to use the RLF Algorithm as our initial upper bound, we concluded that not all RB versions were necessary, as it would have made use of numerous threads which could lead to worsen the performance for other algorithms. Based on this, we decided to only include one of the RB algorithms and the GA in our final program.

Finally, the results from Experiment 3 **(see Graph 1, section 5)** revealed that when starting from the *UB* in 8 graphs out of 10, the time needed to calculate the $\chi(G)$ was less. Only in graph 2 (density of 0.001940) and graph 6 (density of 0.131063) starting from the *LB* was more time efficient by approximately 100 seconds. We conclude that a relation between densities and time needed cannot be made, as the structure of the graph and the type of connections plays an important role on the time needed to calculate the $\chi(G)$.

The previously described results did not surprise us, except for graphs 2 and 6, because from previous experiments we had notice that the performance of the *UB* algorithm is better, and by this we mean that in most cases the value is closer to the $\chi(G)$. Based on this we decided to start the search in the Exact Algorithm from the *UB*.

7 Conclusion

In this section we will answer our main research question and do an overall discussion of how we could improve our investigation. We will conclude by doing a summary.

This report served as an overall analysis of the work done throughout Project 1-1. It explained, outlined and implemented numerous heuristic and exact algorithms for the lower and upper bounds with the aim of investigating the graph colouring problem and providing an answer to our ***Problem Statement***. This exploration was guided by three research questions, each of which were used to conduct experiments and create tables and graphs displaying the corresponding results. To restate, the collection of algorithms outlined in this report were: The *Maximum Clique algorithm* for the lower bound, and the *Greedy, Recursive-Large-First, Reverse Backtracking, Genetic Algorithms* for the upper bound. Moreover, in a search for a closer optimization, Graph Decomposition was used to reduce the problem size, and an Exact Algorithm, which relies on the upper and lower bounds, was also implemented.

Several conclusions can be drawn from the experiments conducted within this report. Regarding the upper bound, to a certain extent, the RLF algorithm was shown to outperform the other heuristic algorithms explored, hence the final decision of not using the Greedy algorithm in our final algorithm. Secondly, the Exact algorithm was shown to be more time efficient when starting from the upper bound, and lastly, the decomposition of a larger graph into smaller subgraphs resulted in reducing the complexity of the problem and, in most cases, the time needed to reach a solution. Furthermore, the Genetic Algorithm was also found to be an interesting and useful tool to improve the upper bound, even though it is difficult to assess whether the exact chromatic number has been converged, or if a new solution is still being calculated.

Overall, this collection of algorithms was able to calculate, what is believed to be, the exact chromatic number of 13 graphs (**see table 3, section 5**) or 14 using (if graph 10 is included, however, this was only calculated by the Bridge decomposition), out of 20, within a reasonable time, chosen to be of 4 minutes. Hence, to answer this report's problem statement, a collection of algorithms can be implemented to accurately calculate the chromatic number of most graphs in a reasonable time. For further research, it is believed that applying an ordered set of edges and/or vertices of a graph (for example, based on their degree) to the Exact algorithm, or using specific decompositions on specific graph types may result in faster and more accurate outcomes.

References

[1] Voloshin, V. (2009). Alabama Journal of Mathematics. *Graph Coloring: History, Results and Open Problems,* 1-3.

[2] Leighton, F. T. (1979). A Graph Coloring Algorithm for Large Scheduling Problems*. *JOURNAL OF RESEARCH of the National Bureau of Standards,* 489-506.

[3] Sager, T. J., & Lin, S. (1991). A Pruning Procedure for Exact Graph Coloring. *ORSA Journal on Computing,* 226-230. doi:https://doi.org/10.1287/ijoc.3.3.226

[4] Malaguti, E., Monaci, M., & Toth, P. (2011). An exact approach for the Vertex Coloring Problem. *Discrete Optimization*, *8*(2), 174–190. https://doi.org/10.1016/j.disopt.2010.07.005

[5] Lewis, R. (2016). Introduction to Graph Colouring. In 1150373885 864988657 R. Lewis (Author), *A Guide to Graph Colouring Algorithms and Applications* (pp. 1-25). Cham: Springer International Publishing.

[6] Prosser, P. (2012). Exact Algorithms for Maximum Clique: A Computational Study. *Algorithms*. doi:10.3390/a5040545

[7] Galinier, P., Hao J. (1998). Hybrid Evolutionary Algorithms for Graph Coloring. In: Journal of Combinatorial Optimization. doi:10.1023/A:1009823419804

[8] Adegbindin, M., Hertz, A., & Bellaiche, M. (2016). A new efficient RLF-like algorithm for the vertex coloring problem. Yugoslav Journal of Operations Research. 26. 3-3. 10.2298/YJOR151102003A.

[9] Mallawaarachchi, V. (2017). Introduction to Genetic Algorithms. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 . Accessed: 06/01/2021.

[10] Riazi, A. (2019). Genetic algorithm and a double-chromosome implementation to the traveling salesman problem. https://doi.org/10.1007/s42452-019-1469-1

[11] Bhowmick S., Hovland P.D. (2008) Improving the Performance of Graph Coloring Algorithms through Backtracking. In: Bubak M., van Albada G.D., Dongarra J., Sloot P.M.A. (eds) Computational Science – ICCS 2008. ICCS 2008. Lecture Notes in Computer Science, vol 5101. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-69384-0_92

[12] Lewis, R. (2016). Advanced Techniques for Graph Colouring. In 1150348151 864971525 R. Lewis (Author), A Guide to Graph Colouring Algorithms and Applications (pp.55). Cham: Springer International Publishing.

[13] Lewis, R. (2016). Can We Solve the Graph Colouring Problem? In 1150348151 864971525 R. Lewis (Author), A Guide to Graph Colouring Algorithms and Applications (pp. 17-20). Cham: Springer International Publishing.

[14] Chaudhuri, P. (1998). An optimal distributed algorithm for finding articulation points in a network. *Computer Communications*, *21*(18), 1707–1715. https://doi.org/10.1016/s0140-3664(98)00211-4

[15] Chang, E. J. H. (1982). Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Transactions on Software Engineering*, *SE-8*(4), 391–401. https://doi.org/10.1109/tse.1982.235573

[16] Mehmet, H., Karaata, P., & Chaudhuri. (1999). A self-stabilizing algorithm for bridge finding. *Distrib. Comput*, *12*(12), 47–53.

[17] *Sorting Using Networks of Queues and Stacks | Journal of the ACM*. (2020). Journal of the ACM (JACM). https://dl.acm.org/doi/abs/10.1145/321694.321704?download=true

[18] Chaudhuri, P. (1994). An efficient distributed bridge-finding algorithm. *Information Sciences*, *81*(1–2), 73–85. https://doi.org/10.1016/0020-0255(94)90090-6

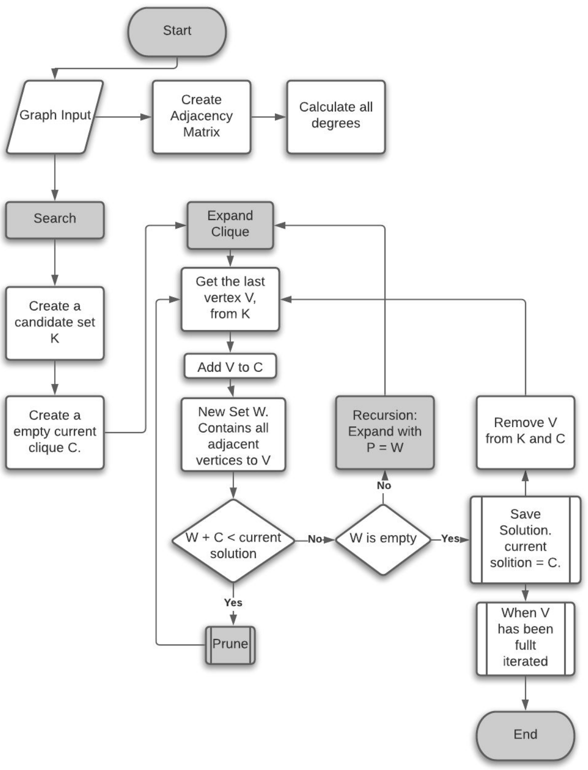
[19] Panda, B. S. (n.d.). *Cut vertices, Cut Edges and Biconnected components MTL776 Graph algorithms*. https://web.iitd.ac.in/~bspanda/biconnectedMTL776.pdf

APPENDIX

Appendix A: Abbreviations:

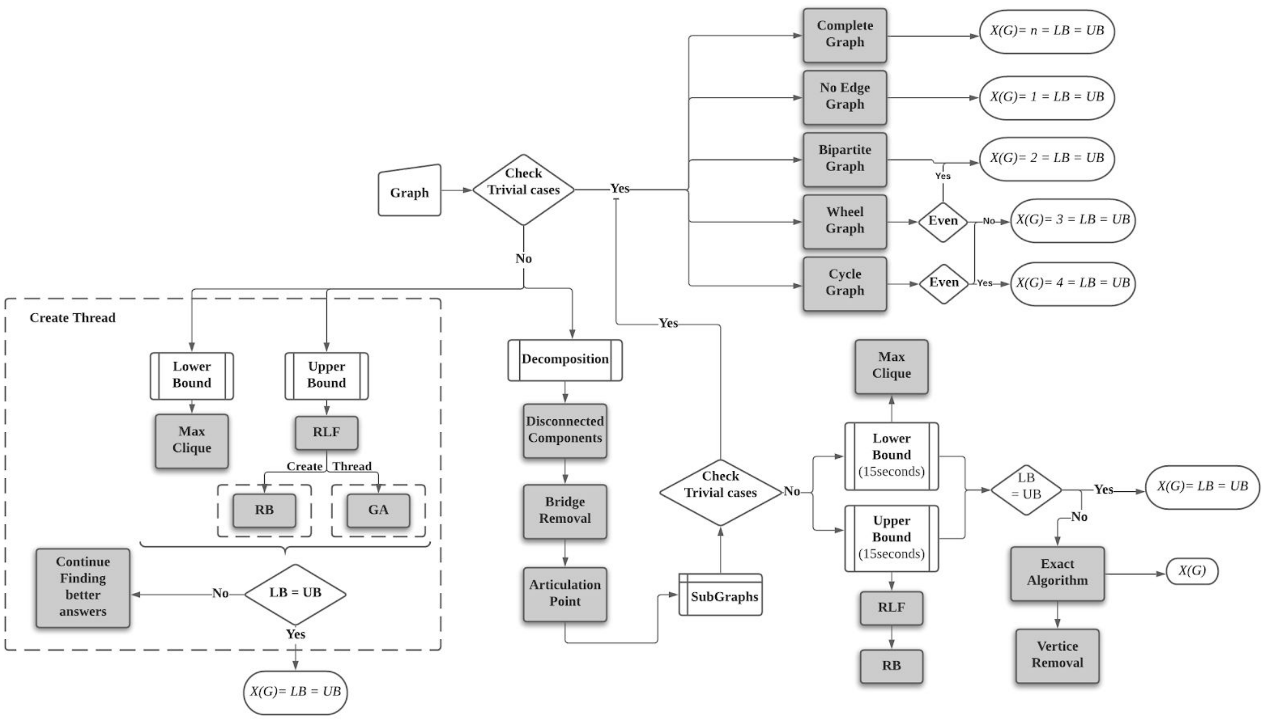
- ❏ G : A given graph
- ❏ V : Set of vertices
- ❏ E : Set of edges
- ❏ $\delta(G)$: A subgraph of G
- ❏ $\chi(G)$: Chromatic number of a graph G
- ❏ n : The number of vertices within a graph
- ❏ ν : Vertex of the graph G
- ❏ $deg(\nu)$: The degree of v is the number of connected vertices to v
- ❏ $\omega(G)$: The number of available colours to colour G
- ❏ $\sigma(G)$: The current most optimal colouring scheme for G
- ❏ $max(\sigma(G))$: The vertices that have the maximum colour value assigned in $\sigma(G)$
- ❏ $\alpha(\nu)$: The adjacent vertices, also called neighbours, of ν
- ❏ *AP*: Articulation Point
- ❏ *LB*: Lower Bound
- ❏ *UB*: Upper Bound

Appendix B: Lower Bound Flowchart



Start
Graph Input
Create Adjacency Matrix
Calculate all degrees
Search
Create a candidate set K
Create a empty current clique C.
Expand Clique
Get the last vertex V, from K
Add V to C
New Set W. Contains all adjacent vertices to V
W + C < current solution
No
Yes
Prune
W is empty
No
Yes
Recursion: Expand with P = W
Remove V from K and C
Save Solution. current solition = C.
When V has been fullt iterated
End

Appendix C: Program Flowchart



Appendix D: Reverse Backtracking Algorithm Pseudocode

Reverse Backtracking Algorithm 1 reverseBacktracking(upperBound, colouringScheme)

```
for each colour in colouringScheme do
    if colour of vertex is equals to the upperBound then
        save the number of the vertex;
    end if
end for
for each vertex with the (current) upper bound assigned do
    for each level/layer do
        reset the colours of the adjacent vertices;
        if backtracking finds a way to colour the vertices with the available colours then
            break;
        end if
        if this is the last level and it didn't find any solution then
            return available colours + 1;
        end if
    end for
end for
we found a colouring solution to every vertex;
try everything again with available colours - 1;
```

Appendix E: Raw Data obtained from Experiment 3

Graph	Density	LB Time/ms	UB Time/ms
02	0.001940	0.295	145
04	0.002692	0.412	0.145
07	0.011267	76.887	0.174
20	0.014385	0.179	0.075
12	0.024166	0.150	0.085
19	0.035220	15.274	0.083
13	0.049050	15.717	15.112
17	0.066512	107.455	0.016
06	0.131063	0.140	106.477
16	0.873744	0.099	0.072

Appendix F: Pseudocode of the Visit method from the algorithm that identifies Bridges.

Information considered for each vertex *(v) i*s discussed in **section 2.4.2.3**. The meaning of the status of the search on *v* is explained in **section 3.4.1.**

Bridge Identification 1 Visit(v)

```
time +=1
v.status = 1
v.low = v.timeDiscovered = time
for e in v.adjacentVertices do
    if e.status == 0 then
        e.parent = v
        visit(e)
        v.low = min(v.low, e.low)
        if (e.low > v.timeDiscovered) then
            bridges.add(e, v)
        end if
    else if (e != v.parent) then
        e.low = min (v.low, e.timeDiscovered)
    end if
end for
v.status = 2
time +=1
v.timeFinished = time
```

Appendix G: Greedy algorithm for the upper bound.

The Greedy algorithm starts creating an adjacency matrix for the graph. Based on this matrix, the algorithm will start looping through all vertices and colour them using the technique described below.

Greedy Algorithm 1 calculateUpperBound(n, e)

```
for each vertex v in graph G do
    colour v with first available colour
    store colour in array C
end for
for each colour in C do
    if colour > maxColour then
        maxColour = colour
    end if
end for
return maxColour
```

Appendix H: Data of the graphs used.

Graph	$\|V\|$	$\|E\|$	Density
01	218	1267	0.05356614
02	529	271	0.00194048
03	206	961	0.04551267
04	744	744	0.00269179
05	215	1642	0.07137579
06	131	1116	0.13106283
07	212	252	0.0112671
08	107	516	0.09098924
09	433	529	0.00565606
10	387	2502	0.03349801
11	85	1060	0.29691877
12	164	323	0.02416579
13	143	498	0.04904954
14	456	1028	0.00990939
15	4007	1198933	0.14938075
16	107	4955	0.87374361
17	164	889	0.06651205
18	907	1808	0.00440041
19	106	196	0.03522013
20	166	197	0.01438481