Maastricht University Department of Data Science and Knowledge Engineering

---

Project 1-1 2020/21
Group 27 Project Report:

To what extent can computer algorithms be developed and employed to efficiently compute the so-called chromatic number of a given graph?

(To what extent does the execution environment of the code affect the results obtained?)

Çagan Can Idil, Andreea-Violeta Lazăr, Laurence Manoukian, Nils Stolk, Giacomo Terragni, Abhinandan Vasudevan, Vikram Venkat

---

Table of Contents

Abstract

This report focuses on describing the process of creating an algorithm that computes the chromatic number of a given graph – denoted by $\chi(G)$ – by reading in a graph and assigning colours to the respective vertices and edges, along with displaying different varieties of algorithms and methods that fulfil the same function.

Throughout the report, the logic of various algorithms – especially that of our main approach using the Bron-Kerbosch and greedy algorithmic methods – will be explained, such as showing the upper and lower bounds of a given graph on the greedy algorithm during initial testing and development, as they are the primary approach to making computations easier. Furthermore, multiple examples and results of the algorithms will be displayed on diagrams and in tables; every observation was noted after repeating the experiment with each graph multiple times, in order to increase the reliability of the test results.

Moreover, the design of the project consisted of the group addressing each problem by creating additional subtasks, which would be addressed by different people respectively and submitted to the team after finishing said task, all of which would amalgamate to create one rounded off package, designed around the concept of chromatic numbers and graph colouring. Hence, the main aim of calculating the chromatic number by using optimization algorithms to improve run time was accomplished.

Introduction & Project Description

Introduction

As students of the Data Science and Artificial Intelligence Bachelor's Programme at Maastricht University (UM), our academic lives revolve heavily around a concept called "Project-Centred Learning", or PCL. A unique higher education system, similar to that found in other programmes offered by UM, PCL presents students in groups with a certain task – in the case of Data Science (or DKE, as it will be named henceforth) students, a programming project – every semester. The groups are then assigned a task, over the course of the semester, to produce and present a software product according to the guidelines set forth by the DKE professors and Board of Examiners (BoE), culminating in a tournament that decides which project group created the best product of all.

During the course of each separate project, the students are still acquiring new knowledge in their lectures, and they are expected to apply this knowledge to their product as soon and as elegantly as possible, in order to keep their software evolving and improving in performance over the project's duration. Hence the designation of PCL.

This education method not only reinforces the students' ability to better retain their acquired skills, but also reinforces their confidence in said skills, and this is best demonstrated by having the students present their creation to the BoE; by doing so, the examiners can verify if the students have fully understood the task requirements, and can also check if the students know how to explain what exactly they have created.

Project Description

The project revolved around the topic of chromatic numbers and graph colouring in the programming language Java. A graph $G = (V, E)$ is simply a set of points V (also known as *vertices*) with a certain colour C, connected by lines E (also known as *edges*). Each edge connects **exactly** two vertices, and no two vertices of the same colour can be connected by one edge. The chromatic number of a graph G, or $\chi(G)$, is the smallest number of colours needed to colour the vertices of G, such that no two adjacent, or connected vertices share the same colour.

Chromatic numbers and their associated graphs have a broad scope of applications, including complex artificial intelligence, or AI, problems involving neural networks and other problems that involve a number of interconnected variables and/or scenarios that have certain restrictions and/or requirements, such as stock markets and their predictions, threat analysis, and more. To get to that stage of computing vastly complex problems, we must first be able to create a rudimentary algorithm, in the form of code, that computes the chromatic number $\chi(G)$ of a given graph G, and adapt that algorithm to find $\chi(G)$ for more complex and/or larger graphs.

We must also be able to manipulate a graph in real-time, i.e., show that the solving algorithm is capable of handling changes en masse.

In order to achieve this, a number of approaches can be used, but independent of that, we need to create certain algorithms to estimate the lower and upper bounds of $\chi(G)$, and some code to connect both of those together, to finally form our general solving algorithm. Subsequently, the group had to ensure that our solving algorithm works, as previously mentioned, on larger and/or more complex graphs that are presented to the software. How exactly these algorithms are implemented determines the approach used in this project, and there are a number of options to choose from. Their efficiency and focus in certain areas also vary from one approach to the next.

Therefore, as a group we set out to answer the question:

"To what extent can computer algorithms be developed and employed to efficiently compute the so-called chromatic number of a given graph?"

In this report, we also seek to answer a secondary question, one that may not be vital to the outcome of this research report, but an important one nonetheless:

"To what extent does the execution environment of the code affect the results obtained?"

This of course leads to a couple other optimisation-based questions. These being questions such as: Does the environment of execution effect how well the programme runs? If so, how does it influence the programme? Does the usage of different environments only affect the computational time of the programme? Why might this be the case?

In the upcoming sections, we will break the whole project down into its three phases and we will dissect our approach and algorithms and explain how they work to achieve their intended purpose, which is to solve a graph G and return its chromatic number $\chi(G)$ as efficiently, as precisely and as accurately[1] as possible. We will also use the data we acquire, in order to answer the questions which, we previously posed; in hopes that we are able to help further the state-of-the-art regarding this topic.

Implementation – Breakdown and Explanation of Our Approach

The main implementation of the project was centred around the concept of chromatic numbers and their various applications, and our ability to successfully integrate a set of computational instructions, or algorithms, that will fully take advantage of the properties of chromatic

[1] Precision and accuracy are different; refer to Appendix C **[1]**

numbers. For example, one could simply create an algorithm that computes the chromatic number of a graph, or design and build a game around the concept… which is what we did!

Computation of chromatic number χ(G) of given graph G in Java

Beginning with the algorithm to compute the chromatic number of a given graph, we were first introduced to the entire concept of graph colouring, and its various uses, as mentioned before. Simultaneously, we were being taught the programming language Java, as not only our first programming language taught in the Bachelor's Programme, but also as the language to be used for this project.

Following our brief, yet useful introduction to chromatic numbers, our group set out to research certain algorithms that could fulfil our requirements of efficiently, precisely, and most importantly, accurately return χ(G) for a given graph. These graphs were provided to us by our examiners and professors; below, we have depicted two examples of these complex webs that we call graphs.

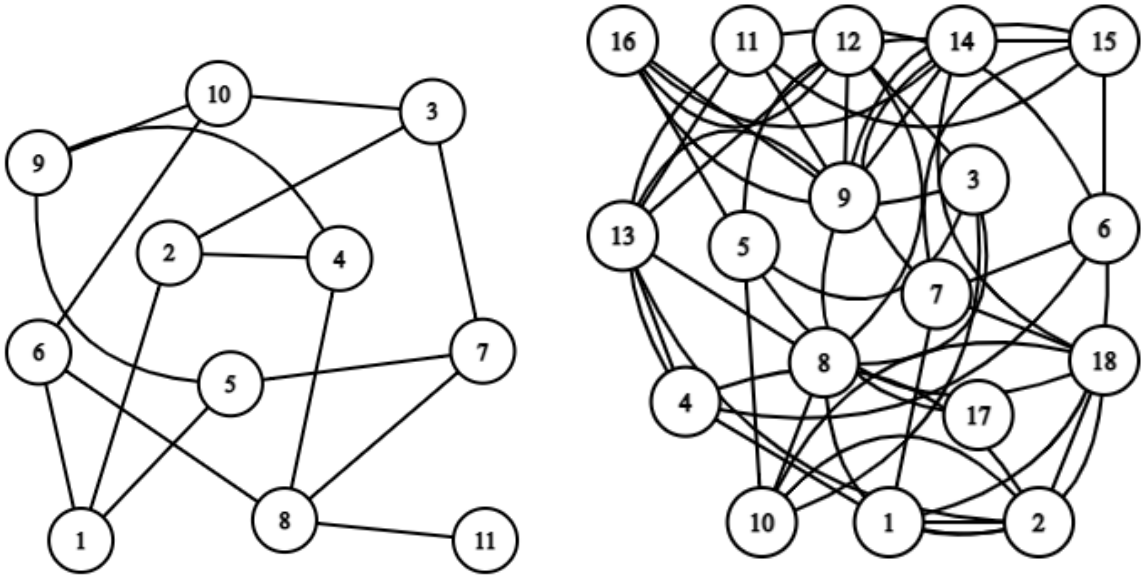

Figure 1 & Figure 2: Graphs from initial set of test graphs

Initially, we considered two algorithmic approaches to solve a given graph, one technique known as the **backtracking algorithm**, wherein the solution is built incrementally by probing all possible solution paths and removing solutions that do not fit the given restrictions; the other technique is called the **greedy algorithm**, where the solution is built using what the algorithm deems to be the most optimal of the available options in that instance of a decision. Both algorithms have their strengths and weaknesses, however it can be argued that since the greedy algorithm possesses one fatal flaw; utilizing backtracking is indeed the better approach:

The greedy algorithm is most suitable for small, quick problems, as it lacks the ability to "think further down the line", i.e., unlike the backtracking algorithm, it doesn't base its final decision on the upcoming choices to be made, but rather on the single choice it has to make at a given moment in time. In comparison, the backtracking approach is able to do just that which cannot be accomplished by the greedy algorithm, i.e., a backtracking approach, in a way, possesses hindsight which it can use retroactively to change its decisions.

At first, we were unsure which approach to choose, considering our relatively rudimentary skills and overall knowledge, as both had their benefits and drawbacks. However, after careful consideration, we decided that our approach would be centred around the greedy algorithm, as the choice came down to the following reasons, that we – at the time – thought gave it an edge over its backtracking counterpart:

- It has a simpler iterative style which would be easier for beginners (e.g., our entire group) to implement, and implement correctly;
- The logic of the greedy algorithm made was straightforward and made more sense to us than the backtracking algorithm (the specifics of this algorithm will be explained shortly).
- Our first – rather naïve – thought was that this approach was more time-efficient to implement, and that it would suffice in providing us with the exact chromatic number of a graph. However, in hindsight (much like the backtracking algorithm), we realised that this was not always the case, especially as the graphs become larger and more complex, and therefore computationally more challenging.

While on the topic of the greedy algorithm, building it from scratch was one of the first challenges we encountered during this venture; as a reminder, the main task was to create a set of instructions that would attempt to minimize the number of colours, also known as the chromatic number, used to colour the vertices on the graphs.

The main goal of the greedy algorithm is to assign colours to their respective vertices as quickly as possible in order to complete the entire graph in the shortest time possible. However, the algorithm doesn't always apply the minimum number of colours to the vertices; it instead guarantees an upper bound on the number of colours (GeeksForGeeks, 2018) .
Below is our breakdown of the algorithm:

- The pseudocode of the greedy algorithm states that no more **n + 1** colours, (where **n** is the number of vertices available on the graph), can be used.
- The maximum number of different colours that are possible for a graph of **m** vertices is **m** colours, depending on the order of the picked nodes and the degree of each vertex.
- Therefore, the order of the vertices, along with the positions of the neighbouring nodes relative to each other, is a major factor that will affect the upper and lower bounds of $\chi(G)$ for a given graph.

Graph Colouring Game Using A GUI

The computation of $\chi(G)$ can be applied in many areas, such as solving complex multivariable problems, or even creating a game that runs on the computation of several chromatic numbers, which are all then put to use in their own ways. Our aim was to create a single-player game revolving around the player completing tasks directly related to solving graphs. Considering our intermediate skills in Java at that point in time, we wanted to make it complex, but we also did not want to punch above our skill level. Keeping that in mind, here are the main principles we adhered to while creating this game:

- It has to be a single-player, user-friendly game.
- The application should be able to generate a random graph, or alternatively, the user should be able to read in/load a graph – in the end, we decided that random generation or user-specified dimensions were the way to go.
- The way the game is played is similar to the method in which the chromatic number is computed in phase 1, i.e., the player has to colour the graph with as few colours as possible. Accordingly, warnings will be generated if the user chooses an invalid colour (if 2 adjacent vertices have the same colour).

Now that the fundamentals of the game have been established, there remains the question of the actual game design. For this, we wanted to implement the following three game modes, which are each described in detail below:

I. **To the Bitter End:** The user has the simple task of colouring the graph as quickly as possible. The chromatic number has already been computed by the computer for that particular graph (from phase 1) and the user is not allowed to finish until the minimum of colours has been reached.
II. **Best Upper Bound:** The player is given a fixed amount of time and they have to find a colouring with as few colours as possible in the given time. (Here it is not necessary that the user finds the minimum number of colours)
III. **Random order:** Here the computer generates a random ordering of the vertices and the player has to pick the colours of the vertices in exactly that order. Once the colour of a vertex has been chosen, it cannot be changed again. (Note that the player can never get stuck because they are always allowed to introduce a new colour.) The goal for the player is to use as few colours as possible.

Using these key points, we were able to design a graphical user interface, or GUI, that the user would be able to use to play the game as designed without any errors/bugs.

Additionally, we thought that a hint system of sorts would be useful in aiding the player's decisions at each step. This system would analyse the given graph and display useful clues that the player can use to their advantage and better their chances at beating the game. Our hint system works as follows:

- A specific node is targeted and fixed as the focus node.

- The node is then checked whether it has any neighbouring nodes.
- The neighbouring nodes' neighbours are also checked, thus creating a tracing pattern in order to verify and compare what chromatic colour the nodes have and whether there's a possibility of simplifying the chromatic number.
- If there's a certain option of simplifying by accessing one of those nodes, the hint system will display a separate panel, informing the user which node to access and recommending an appropriate colour that should be applied to that vertex.

Algorithm optimization to find $\chi(G)$ for a larger, more complex graphs

Sometime after creating the game, we thought to ourselves if there was any way of improving the ability of our graph solving algorithms to solve more complex graphs, as during our initial testing with the first iteration of the algorithm, the inaccuracy and failure rate to achieve their goals grew considerably, relative to the increasing complexity of the graphs provided to it. As a result, the team came to a realization that there might be a different method to analyse and exploit certain characteristics present in some of the given graphs, such as special structures or cycles or otherwise, to increase the efficiency and restore the accuracy of our solving algorithms, while also decreasing computation times for each graph. The figure below represents a flowchart that we initially created as a developmental model, which describes the planned function of our improved graph solving algorithm(s):

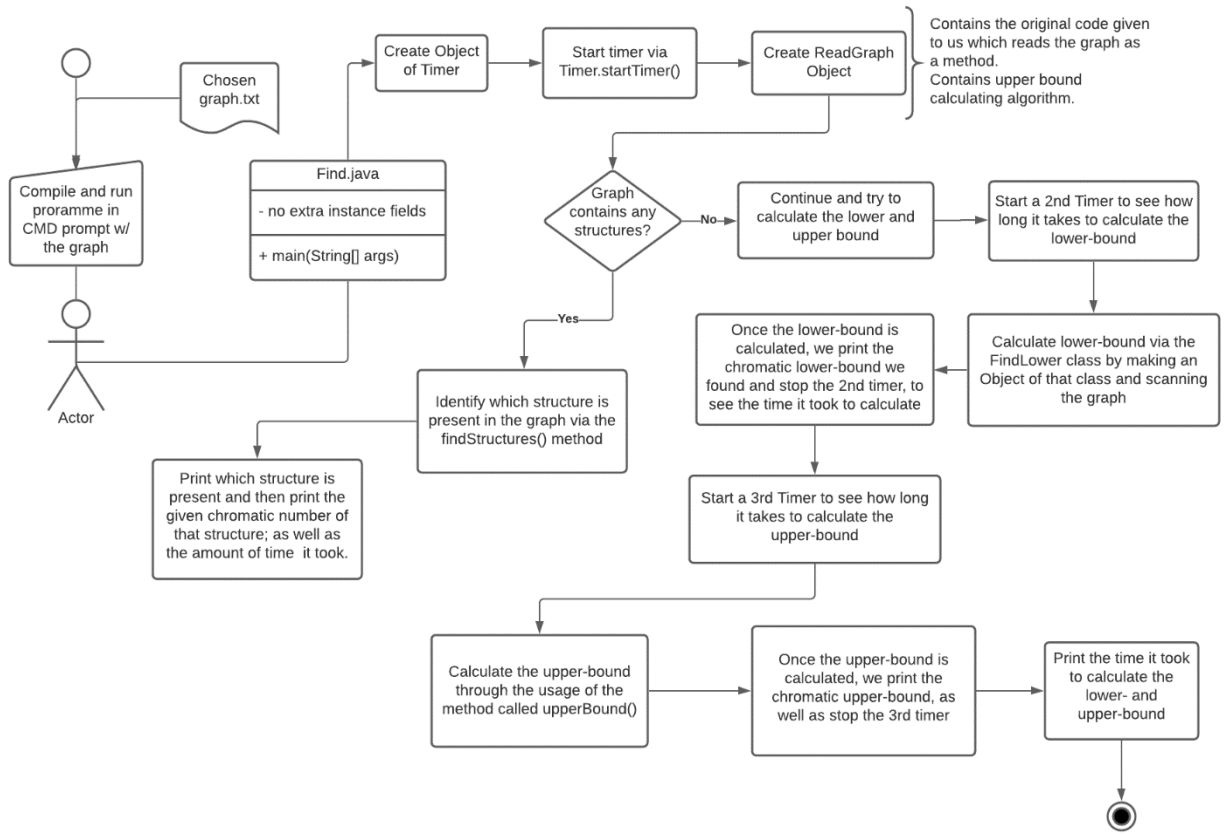

Figure 3: Development flowchart of our new graph solving algorithm

For simplicity's sake, we opted to avoid using the GUI, to prevent any unwanted side effects and to improve efficiency of the program as a whole – and also because we felt it was unnecessary, as this would mostly be a backend program providing displayable information for the frontend. For this phase, we decided to focus our efforts on optimizing our grassroots algorithms to work for the new and more complex graphs, as well as writing a more efficient lower bound algorithm for all graphs, to ensure consistent and accurate results. Explained below in detail is how we implemented these revised algorithms:

We will start with the fact that the program has 7 methods:

- **Timer method**: This measures the time to calculate the upper bound, the lower bound and the chromatic number.
- **Lower bound method:** This method computes the lower bound of $\chi(G)$, in case the other structure search algorithms do not find the aforementioned special structures, i.e., this is the backup brute force algorithm to compute $\chi(G)$ (Zhou, 2013).
- **Adjacency matrix creation method:** This method transforms the graph text file into an adjacency matrix, which will detect structures in the graph.
- **Upper bound method:** This returns the computation of $\chi(G)$ based on the results of the other algorithms.
- **Transform graph method:** This method generates a text file with a new format to fit the requirements of our lower bound algorithm.
- **Bipartite checking method:** checks if the given graph can be coloured with at most two colours.
- **Cycle-checking method:** checks if the given graph is cyclical and determines whether the graph can be coloured in at most 2 or 3 colours.

Coming to the control flow of the program: First, the program checks for any special structures present in the graph; if a structure is found, the chromatic number, along with the time taken to compute it, will be printed. Otherwise, if these structures are not detected, the upper and lower bounds will be calculated, and the time taken to compute both is also calculated/timed. In case the upper and lower bounds are the same, the program will output the chromatic number along with the result of the greatest stopwatch time.
Hence, the final result consists of the following values:

- **In case there is a structure:**
    - Type of structure
    - Chromatic Number
    - Time Elapsed
- **In case no structures are found and lower and upper bound are different:**
    - Lower Bound of $\chi(G)$
    - Upper Bound of $\chi(G)$
    - Time Elapsed for both
-

- **In case no structures are found and the lower and upper bound are the same:**
    - Chromatic Number
    - Time Elapsed

Algorithm Optimization – Explanation of each algorithm

The class that is the topic of this section contains three methods, all of which work together to produce the actual result we seek to obtain in the first place, the chromatic number. Hence, we will seek to describe it by delving into the more detailed explanations of certain algorithms and approaches that were mentioned before that either had no explanation to them, or were not fully explained, so as to preserve the reader's attention while reading the important parts of our report. We wanted to keep these for their own separate sections, as it would make reading and understanding the rest of our work a lot easier, and because each of these algorithms deserve their own space to be laid out in detail.

Bipartite algorithm:

A Bipartite graph is one that can be divided into 2 sets, depending on the individual connections of edges in each set. Our approach was the 2 colourable logic, i.e., modifying our original backtracking algorithm.

Another option to put to use was that if a graph is bipartite, we didn't need to implement the odd cycles algorithm, since it will never contain them.

The algorithm was written on the basis of BFS (Breadth First Search), part of which we learnt in period 1 during the DSAI course. We opted for this approach instead of the odd cycle, since we felt it was better suited and more compatible with the java language. The principle is to assign a colour to a vertex in one set, and then assign the other colour to the other set, and go back and forth between the two sets, checking the validity each time.

Lower Bound: Bron-Kerbosch Algorithm (Bron & Kerbosch, 1973):

- Initially, we planned on implementing a variation of the Zykov algorithm and Kruskal's MST, but upon further research, decided that the Bron-Kerbosch (with pivoting version) would be the best approach.
- Advantages of including pivoting – only tests the pivot vertex and its non-neighbours in the recursive calls to the vertices subset (R).
- Lower bound method:
    - The Bron-Kerbosch algorithm finds the maximal cliques[2] of an undirected graph.

[2] See Bron, C., & Kerbosch, J., Algorithm 457 (1973), for additional information on cliques (575 – 577)

```
P = {V} //set of all vertices in Graph G
R = {}
X = {}

proc BronKerbosch(P, R, X)
    if P ∪ X = {} then
        print set R as a maximal clique
    end if
    Choose a pivot u from set P U X
    for each vertex v in P \nbrs(u) do
        BronKerbosch(P ∩{v}, R ∪{V}, X \{v})
        P = P \ {v}
        X = X ∪{v}
    end for
```

Figure 4: Pseudocode for lower bound method

In order to find the cliques, the algorithm uses three sets: the first one called "P" stores the possible cliques, the second one called "R" contains all the vertices that adjacent to every vertex of the first set and the third one called "X" stores the nodes that are already in some clique, this set is especially important because avoid the creation of duplicate cliques. Once a clique is found, its vertices are stored in a set so that the program will not compute that clique again.

When the first clique is found, its number of vertices is compared to the second clique, then the size of all the next cliques found is compared to the size of the previous clique and during this process the higher degree of vertices is stored. The size of the bigger clique corresponds to the lower bound of that graph.

Upper Bound Algorithm: Pseudocode

This method is one that mediates, rather than actually carry out a definitive operation. It does so by checking if the vertices have all been coloured. In the case that the nodes aren't assigned, the algorithm recursively assigns colours to the uncoloured vertices according to certain set requirements that have been determined by other computation results. To better describe this specific method, we have written some pseudocode that lays it out, which can be seen below:

Function upperBound() {

Send the matrix containing the edges to the method assigningColours(,,,) so that we can assign colours to each vertex using that information.

Return the number of colours that we had to use to properly fit the given information (Look through verticesColoured and take the largest number).

end
}

Function assigningColours(Matrix theEdges, Array verticesColoured, int numVertices, int checkVertex) {

if (at the final vertex) {

chromatic number = number of colours used;
Return chromatic number;
}

For (each vertex of the graph) {

If (okay to assign a colour (check via isColourAllowed(,,,))), then
assign next colour to that vertex
Test the next vertex using recursion assigningColours(,,,vertex+1)
If the next vertex has no edges, then
Assign the next vertex the first colour

end
}

Function boolean isColourAllowed(Matrix theEdges, Array verticesColoured, Int numVertices, Int checkVertex) {

For each vertex in relation to checkVertex:

If the neighbouring vertices are related to the vertex to be checked, then {

Return false;
}

If neighbouring vertices aren't related to the checkVertex, then {

Return true;
}

end
}

Cycle graph:

The cycle graph is another graph structure which our code is designed to identify. A cycle graph is a graph wherein every vertex has 2 edges, thus creating a circular shape when graphically represented. We check for this property by using a Boolean method called

checkCycleGraph() in the ReadGraph3.java file. This is a method which goes through the matrix which represents the edges of the graph, and checks whether each vertex in the graph has only 2 edges. If each vertex has only the 2 edges, then the method will return “true” and allow the programme to give its assigned output (depending on whether the number of vertices are even or odd). If this does not apply, i.e. not all graphs have only two edges, then the programme continues calculating the chromatic number.

All of this amounts to the following flowchart, which is how our actual program works:

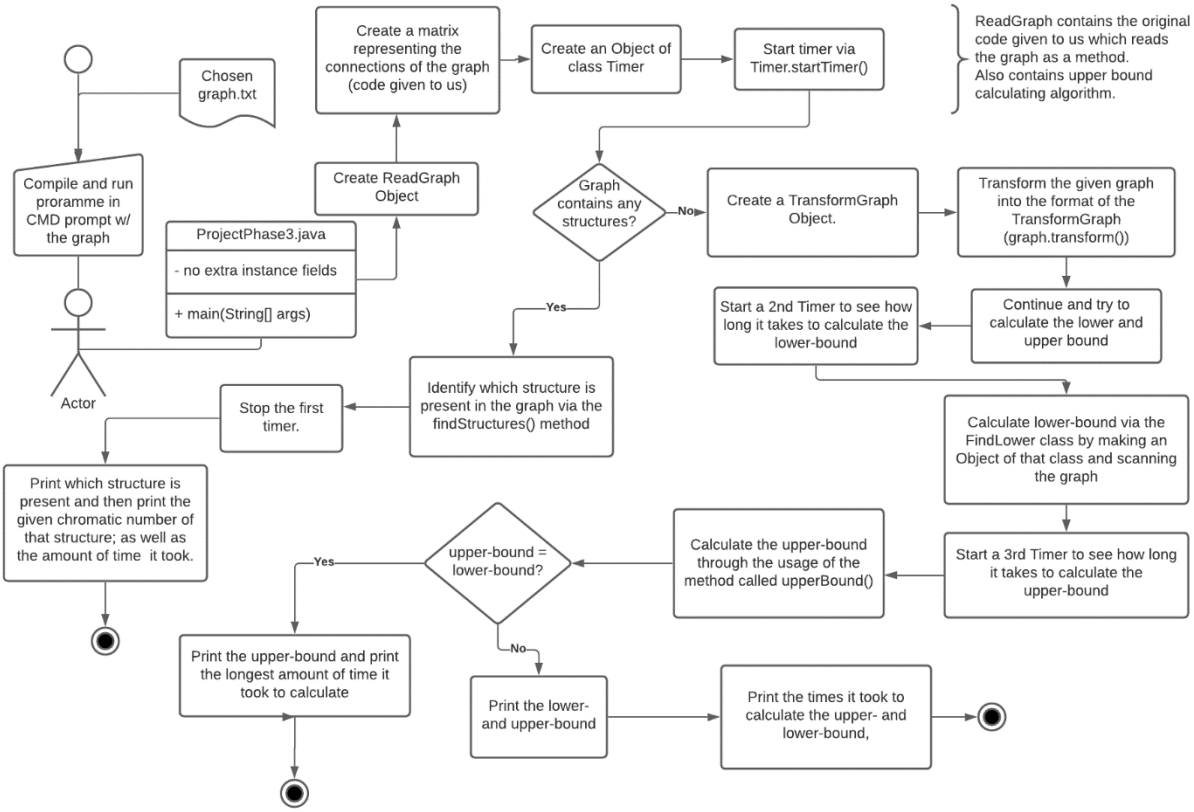

Figure 5: Final flowchart of our new graph solving algorithm, post-implementation

Application Testing (Experiments)

In order to answer the primary and secondary questions, we have decided to conduct two large tests.

First, we would test our code/product by running the example graphs, which we were given in the beginning of our study, on 2 separate operating systems (OSs). We decided that this experiment would be the optimal choice, as we would be able to test the speed of our product, as well as being able to tell which operating system gives us the most consistent performance for algorithms which solve this "Graph Colouring" problem. The exact reasoning for two different OSs is specified further down below.

The second test we would conduct is to test another set of example graphs, which were given to us towards the end of our project. However, the difference is that these graphs are a lot more complex than the other set. The group chose to conduct this experiment, as it would allow us to test more complicated graphs using our product, and possibly give an insight on how well our product works. If the product is able to handle graphs like these, we would know that it is an efficient product which can be relied upon.

For the sake of having increased consistency, and also to prevent the operating system from influencing the data, we conducted our testing across two operating systems; the technical details of each machine are listed below in the following table:

	Computer 1	**Computer 2**
Processor Model and Clock Speed	Intel Core i7-8750H @ 2.2 GHz	Intel Core i7-8750H @ 2.2 GHz
Installed RAM	16 GB	16 GB
Operating System and Version	Windows 10.0.18363 Build 18363	Linux Ubuntu 20.04 LTS

Figure 6: Technical specifications of the experimental machines

Both experiments were identically conducted using the following procedure:

- Run the code through the command prompt
- Test a given graph 10 times.
- Take the final upper and lower bound to be the most commonly occurring upper and lower bound from the 10 tests.
- Finally, take the average amount of time it took to calculate the chromatic numbers/ boundaries.

Running the experiments in this standardised manner allows for it to be very easily replicated in the case of an error, with the added benefit of making the experiment concise. These findings will ensure that the obtained results have a direct relation to the experiments, and are as such of relevance to the team and its goals. This procedure also has the benefit of being able to easily

isolate any incongruencies we may occur during testing, of which, thankfully, there were only a few, and hence they were quickly and effectively resolved.

Results & Discussion

From initial testing of our grassroots graph solving algorithm, we noticed a certain peculiar difference pertaining to running the program on different operating systems (OSs), specifically our code a faster run/compute time on Ubuntu, compared to Windows. At first, we were not able to fully understand why this was the case, although we had a number of theories, however only one stood to be true; this will be discussed shortly. This observation was also found to be consistent with the more complex files we created and ran, the results of which are shown in the data below:

Graph	Chromatic #	Upper bound	Lower bound	Average Computing time Chromatic # Windows	Average computing time upper bound Windows	Average computing time lower bound Windows	Average Computing time Chromatic # Linux	Average computing time upper bound Linux	Average computing time lower bound Linux	Vertices	Edges
phase3_02	2 (bipartite)			0.20			0.40			529	271
phase3_04	2 (bipartite)			10.20			7.50			744	744
phase3_07	3			16.10			6.90			212	252
phase3_13	11			8.20			9.50			143	498
phase3_15	2 (bipartite)			64.90			76.30			4007	1198933
phase3_16	98			13.70			12.80			107	4955
phase3_17	15			5.30			11.60			164	889
phase3_19	8			8.40			12.00			106	196
phase3_01		6	8		15.70	2.50		11.70	1.10	218	1267
phase3_03		3	7		9.30	3.10		12.60	1.10	206	961
phase3_05		5	10		14.70	1.90		12.80	1.10	215	1642
phase3_06		10	13		18.20	2.00		16.50	1.60	131	1116
phase3_08		4	7		33.90	2.00		11.20	1.80	107	516
phase3_09		8	12		28.10	2.00		21.30	2.00	43	529
phase3_10		8	9		18.40	3.80		19.00	1.70	387	2502
phase3_11		9	14		20.00	1.80		15.00	2.00	85	1060
phase3_12		2	5		6.50	3.20		8.50	1.70	164	323
phase3_14		3	6		14.80	3.40		13.40	1.50	456	1028
phase3_18		3	5		20.10	9.40		30.50	6.80	907	1808
phase3_20		2	4		9.30	2.00		6.60	1.40	166	197

Figure 7: Data table for the graph in Figure 8

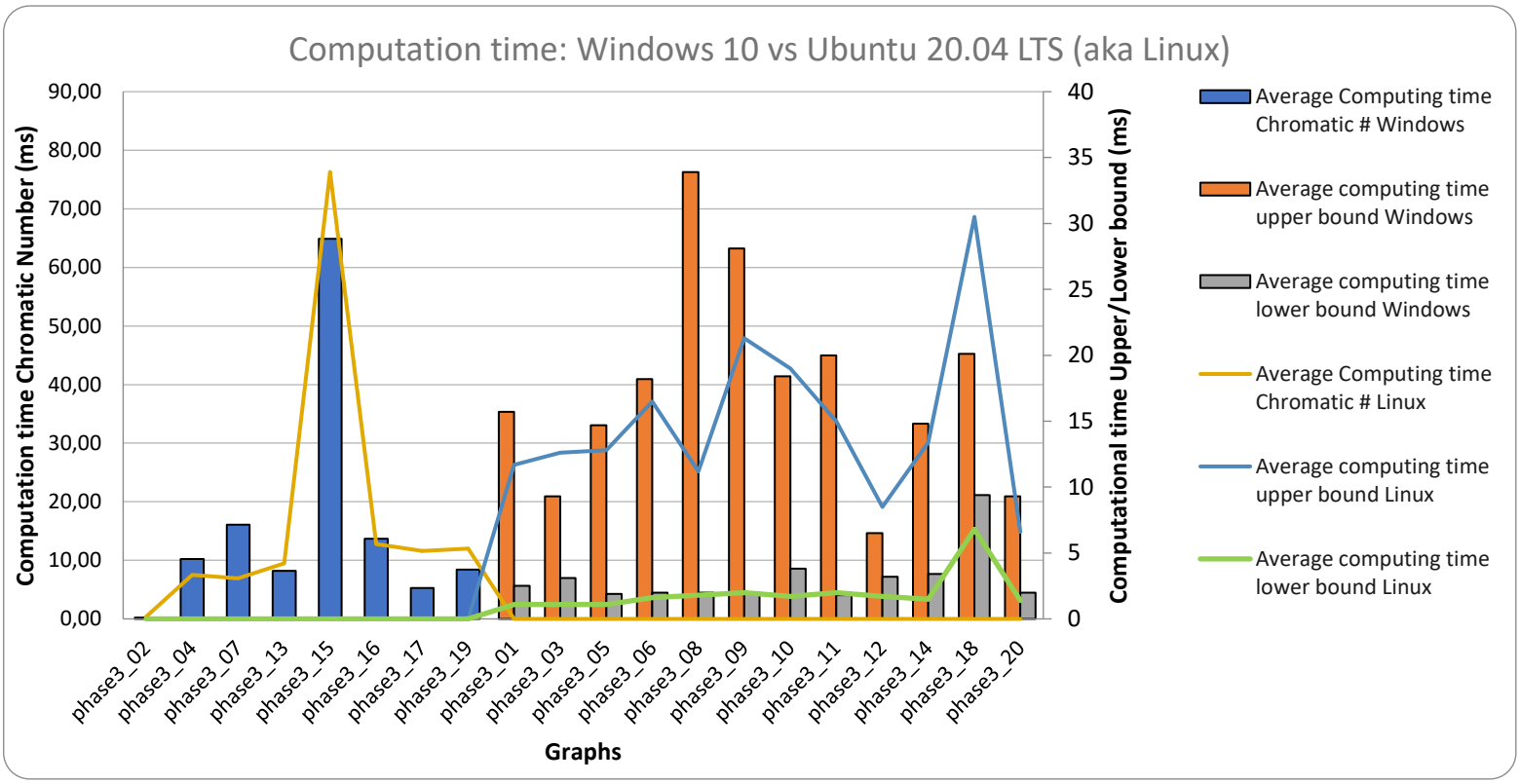

Figure 8: Testing data in a graph, comparing runtimes and compute times on Win10 vs. Ubuntu 20.04

This can largely be attributed to the more lightweight nature of the Linux Ubuntu OS, and also to the fact that compared to Windows 10, Ubuntu 20.04 – or any Ubuntu version that is comparable with Windows 10 – does not require as many background programs or clients to be running to support the OS's functions, hence requiring less RAM to be run, which in turn means that the code being run in the terminal is able to compute results much faster due to the increased amount in memory available compared to the same task being performed on a Windows 10 machine. Furthermore, upon further research into the topic, we found that the Linux kernel used by Ubuntu is more geared towards code development and efficiency, while the kernel of the Windows 10 OS is more geared towards user-friendliness, and due to the OS's design, it requires a lot of non-native clients to be running to support some of the user's desired functions, which leads to Windows 10 being a "fattier" operating system compared to virtually any Linux variant, e.g. Ubuntu 20.04. This enabled us to discover the disparities in time consumption, and hence, in efficiency, between the two operating systems.

While implementing our designed algorithms, we noticed that by improving our algorithms to run more efficiently, we could attain better runtimes, e.g. by removing the backtracking aspect from our algorithm – backtracking is inherently a more time-consuming approach, with the benefit being that as a standalone, it is more reliable – and replacing it with the greedy algorithmic approach. This not only improved runtime, but also enabled us to implement measures that would better detect certain structures, such as a bipartite structure. This leads us to our final conclusions.

Summary/Conclusion, Evaluation, and Discussion

Summary/Conclusion

To summarise our undertakings of the past few months, our goal was to create one or more algorithms that would solve a given graph for its chromatic number – adhering to our approach of using the greedy algorithm as our inspiration – and to then use that algorithm in various applications, and if possible to improve its accuracy and consistency, and also function with more complex graphs in delivering the desired results, which was the exact chromatic number for its corresponding given graph. We then had to pack everything together into a cohesive and coherent package for the examiners to inspect, for which we had to make sure that the various implementations of our design were consistent with one another.

Along the way, many improvements and many other changes were made to the overall package. For instance, we discovered some time into the development that we preferred using the greedy algorithm approach over the backtracking approach, which led to a massive revision of the programs we had thus far written. However, it paid off, because we were able to augment our existing algorithms using the Bron-Kerbosch algorithmic methods, which enabled us to improve our accuracy and efficiency for more complex graphs, as mentioned earlier. All of these results led us to interesting conclusions, and therefore, also our final evaluation.

Evaluation

Coming back to our primary question, which asked "To what extent can computer algorithms be developed and employed to compute the so-called chromatic number of a given graph?", we have come to some interesting conclusions:

Using machine learning (ML) algorithms was a possibility for us, however, given our restricted knowledge on actually coding such an algorithm, we decided that it would be more useful and a better use of our time to create hard-coded algorithms, in order to avoid unnecessary errors, and to ensure that our results were always consistent, and more importantly, correct. ML would have certainly been an ambitious approach; however, it would have led us further away from our original research question, which asked for the effectiveness and the efficiency of such computer algorithms.

Based on all of the prior discussion, development, and analysis, we can come to the conclusion that our algorithms, which were based on optimization, the Bron-Kerbosch and greedy methods, augmented by a bipartite graph search, provide an ideal package to confidently and reliably compute the chromatic number of a given graph. Hence, we can comfortably say that our primary question has been answered beyond satisfaction, i.e., a more efficient way of computing the chromatic number, and that our goals were accomplished.

Discussion

Achieving what we did, in the time we were given, was by no means an easy feat, considering that our group as a whole lacked any type of prior coding experience; something other groups had in the form of one or more members. To add to our struggles, we were faced with the global situation at the time, namely the coronavirus pandemic, which also affected our social and academic lives and careers drastically. As a result, our face-to-face meetings as a group were virtually non-existent, as not all members were able to travel to campus due to the restrictions at their place of residence.

This threw a curveball at our aspirations, and our plans. Nonetheless, it brought some useful soft skills with it, some that future employers may value more than hard skills. These soft skills include being able to work as a team despite time differences and members being dispersed everywhere, being able to coordinate efforts despite the great distance separating us, and most importantly, getting along with the team members despite having little to no face-to-face interactions with the other members. Needless to say, the pandemic also affected us mentally; some were affected more than others, but we learned a lot of things about each other, ourselves and about self-control while working together. These skills and experiences have only made us more experienced and stronger team players.

Despite our relative inexperience and various hardships, we were able to surmount the academic challenge that we were presented with, while actively dealing with the mental and social ones, and persevered through an entire semester of highs, and admittedly also lows, however at every step of the way, we were accompanied by our desire to learn more and to complete this project successfully; a goal, which we were able to achieve, as individuals, as students, and most importantly, as a team.

References

Barnwal, A. (2020, October 22). Kruskal's Minimum Spanning Tree Algorithm: Greedy Algo-2. Retrieved January 11, 2021, from https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/?ref=leftbar-rightbar

Barnwal, A. (2020, November 09). Check whether a given graph is Bipartite or not. Retrieved January 13, 2021, from https://www.geeksforgeeks.org/bipartite-graph/

Bron, C., & Kerbosch, J. (1973). Algorithm 457: finding all cliques of an undirected graph. *Communications Of The ACM*, *16*(9), 575-577. doi: 10.1145/362342.362367

Dsouza, A., & Waterloo, R. (2014, October 27). Maximal Cliques (Bron Kerbosch with Pivot): Java. Retrieved January 13, 2021, from https://algosdotorg.wordpress.com/maximal-cliquesbron-kerbosch-without-pivot-java/

GeeksForGeeks, (2018, May 01). Graph Coloring: Set 2 (Greedy Algorithm). Retrieved January 09, 2021, from https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/

Roberts, E. (2003, April). The Maximum Clique Problem. Retrieved January 10, 2021, from https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/dna-computing/clique.htm

Sakar, A. (2014, January 07). Finds all maximal cliques in a graph using the Bron-Kerbosch algorithm. Retrieved January 08, 2021, from https://gist.github.com/abhin4v/8304062

Zhou, B. (2013). A lower bound for the chromatic capacity in terms of the chromatic number of a graph. *Discrete Mathematics*, *313*(20), 2146–2149. https://doi.org/10.1016/j.disc.2013.05.012

“Andrew1234” (2020, September 29). Maximal Clique Problem: Recursive Solution. Retrieved January 10, 2021, from https://www.geeksforgeeks.org/maximal-clique-problem-recursive-solution/

Appendix A

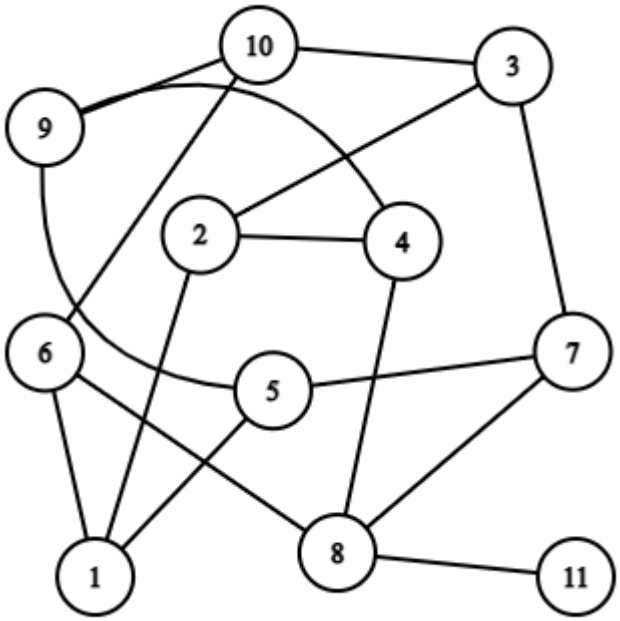

Figure 1. Graph number 4 from the phase 1 graphs

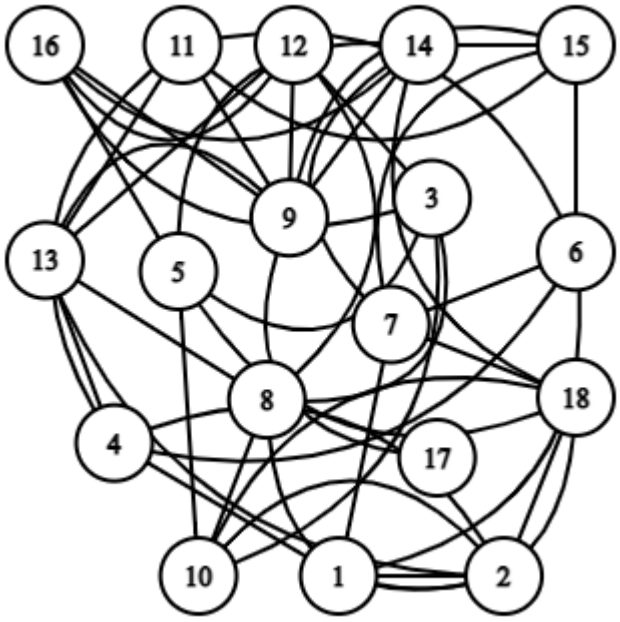

Figure 2. Graph number 3 from the phase 1 graphs

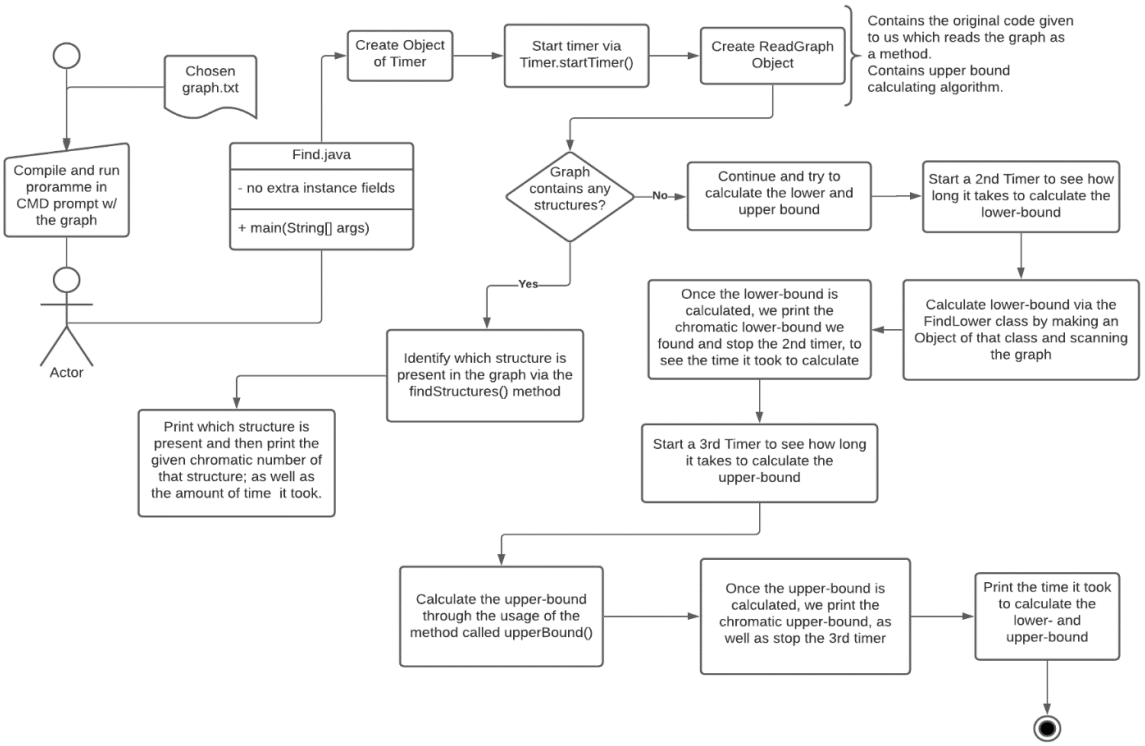

Figure 3. Initial idea of the code's flow.

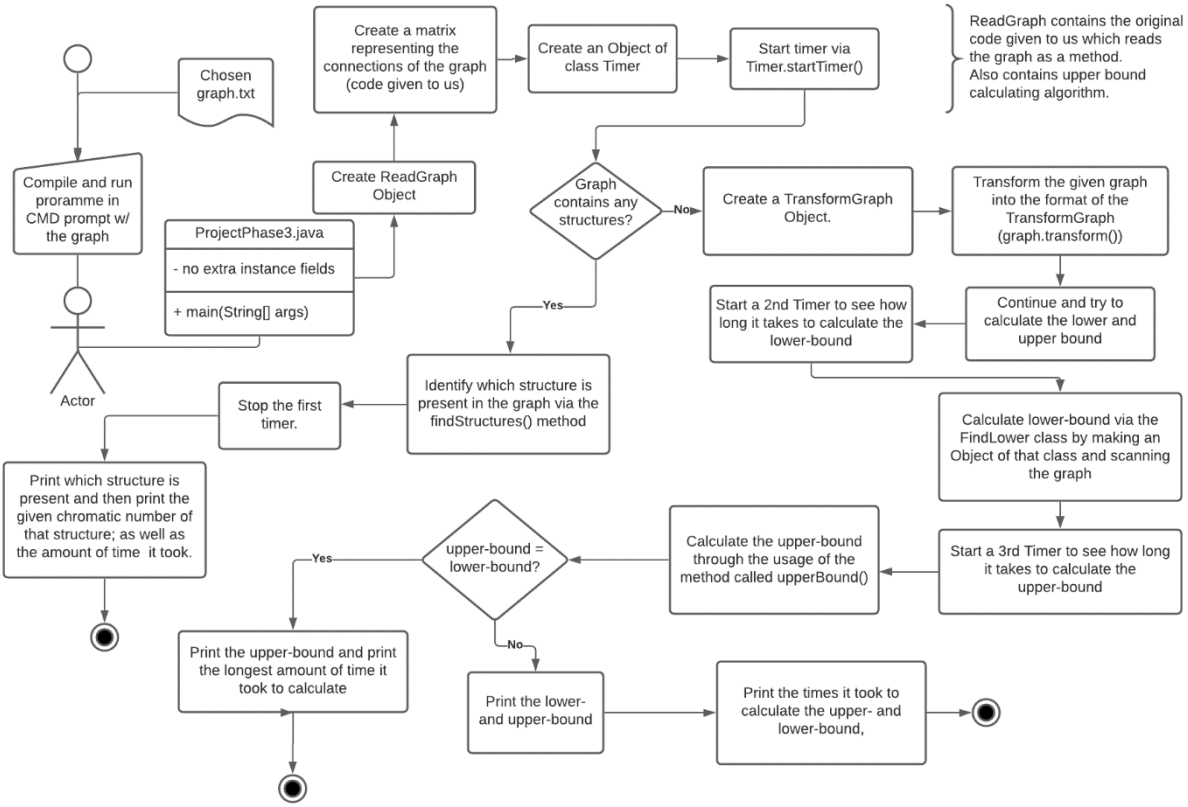

Figure 4. Final iteration of the graph solving algorithm.

Appendix B

Graph	Chromatic #	Upper bound	Lower bound	Average Computing time Chromatic # Linux	Average computing time upper bound Linux	Average computing time lower bound Linux	Vertices	Edges
phase3_01		6	8		11.70	1.10	218	1267
phase3_02	2 (bipartite)			0.40			529	271
phase3_03		3	7		12.60	1.10	206	961
phase3_04	2 (bipartite)			7.50			744	744
phase3_05		5	10		12.80	1.10	215	1642
phase3_06		10	13		16.50	1.60	131	1116
phase3_07	3			6.90			212	252
phase3_08		4	7		11.20	1.80	107	516
phase3_09		8	12		21.30	2.00	43	529
phase3_10		8	9		19.00	1.70	387	2502
phase3_11		9	14		15.00	2.00	85	1060
phase3_12		2	5		8.50	1.70	164	323
phase3_13	11			9.50			143	498
phase3_14		3	6		13.40	1.50	456	1028
phase3_15	2 (bipartite)			76.30			4007	1198933
phase3_16	98			12.80			107	4955
phase3_17	15			11.60			164	889
phase3_18		3	5		30.50	6.80	907	1808
phase3_19	8			12.00			106	196
phase3_20		2	4		6.60	1.40	166	197

Figure 5. Table showing experiment 2 – Computational time on Linux

Graph	Chromatic #	Upper bound	Lower bound	Average Computing time Chromatic #	Average computing time upper bound Windows	Average computing time lower bound Windows	Vertices	Edges
phase3_01		6	8		15.70	2.50	218	1267
phase3_02	2 (bipartite)			0.20			529	271
phase3_03		3	7		9.30	3.10	206	961
phase3_04	2 (bipartite)			10.20			744	744
phase3_05		5	10		14.70	1.90	215	1642
phase3_06		10	13		18.20	2.00	131	1116
phase3_07	3			16.10			212	252
phase3_08		4	7		33.90	2.00	107	516
phase3_09		8	12		28.10	2.00	43	529
phase3_10		8	9		18.40	3.80	387	2502
phase3_11		9	14		20.00	1.80	85	1060
phase3_12		2	5		6.50	3.20	164	323
phase3_13	11			8.20			143	498
phase3_14		3	6		14.80	3.40	456	1028
phase3_15	2 (bipartite)			64.90			4007	1198933
phase3_16	98			13.70			107	4955
phase3_17	15			5.30			164	889
phase3_18		3	5		20.10	9.40	907	1808
phase3_19	8			8.40			106	196
phase3_20		2	4		9.30	2.00	166	197

Figure 6. Table showing experiment 2 – Computational time on Windows

Graph	Chromatic #	Upper bound	Lower bound	Average Computing time Chromatic # Linux	Average computing time upper bound Linux	Average computing time lower bound Linux	Vertices	Edges
phase1_01	11			8.10			76	303
phase1_02		15	25		184.60	1.70	61	1390
phase1_03		3	5		3.00	2.00	18	45
phase1_04		2	3		6.30	2.00	11	16
phase1_05	2 (bipartite)			0.70			93	92
phase1_06		3	7		9.20	1.90	69	409
phase1_07		2	5		4.70	1.90	43	127
phase1_08		7	12		12.70	1.80	56	483
phase1_09	10			4.20			41	184
phase1_10		3	8		10.80	1.20	209	1249
phase1_11	31			42.20			191	3888
phase1_12		3	9		33.50	1.60	195	2365
phase1_13	2			0.60			61	448
phase1_14		16	17		14.30	2.00	31	385
phase1_15		4	8		6.90	2.00	41	205
phase1_16	54			112.60			867	18711
phase1_17		4	5		8.20	1.89	81	293
phase1_18	2 (bipartite)			0.40			52	52
phase1_19	31			6.30			32	466
phase1_20		3	10		12.60	1.80	82	811

Figure 7. Table showing experiment 1 – Computational time on Linux

Graph	Chromatic #	Upper bound	Lower bound	Average Computing time Chromatic #	Average computing time upper bound Windows	Average computing time lower bound Windows	Vertices	Edges
phase1_01	11			13.10			76	303
phase1_02		15	25		153.30	1.90	61	1390
phase1_03		3	5		3.10	2.00	18	45
phase1_04		2	3		5.90	2.00	11	16
phase1_05	2 (bipartite)			0.20			93	92
phase1_06		3	7		16.10	2.00	69	409
phase1_07		2	5		6.70	2.00	43	127
phase1_08		7	12		9.80	2.00	56	483
phase1_09	10			2.80			41	184
phase1_10		3	8		14.50	2.00	209	1249
phase1_11	31			23.40			191	3888
phase1_12		3	9		13.80	2.00	195	2365
phase1_13	2			0.00			61	448
phase1_14		16	17		15.30	1.90	31	385
phase1_15		4	8		11.90	2.00	41	205
phase1_16	54			84.80			867	18711
phase1_17		4	5		7.80	2.00	81	293
phase1_18	2 (bipartite)			0.00			52	52
phase1_19	31			10.50			32	466
phase1_20		3	10		26.20	2.00	82	811

Figure 8. Table showing experiment 1 – Computational time on Windows

Appendix C

```
P = {V} //set of all vertices in Graph G
R = {}
X = {}

proc BronKerbosch(P, R, X)

    if P ∪ X = {} then
       print set R as a maximal clique
    end if

    Choose a pivot u  from set P U X

    for each vertex v in P \nbrs(u) do

        BronKerbosch(P ∩{v}, R  ∪{V} , X \{v})
        P = P \ {v}
        X = X ∪{v}

    end for
```

Figure 9. Pseudocode for lower bound method

List of definitions

[1] Precision is the variation/distance between the results, accuracy is how close the results are to the actual value.