Maastricht University

Department of Knowledge Engineering

PAUL-HENRI SPAAKLAAN 1, MAASTRICHT, THE NETHERLANDS

Project 1.1.3 Report

An Algorithmic Approach to Graph Coloring

Authors:

Yuewei Wu (i6207778)

Christopher Schiffmann (i6245296)

Endri Dibra (i6256488)

Tjardo Neis (i6261493)

19 January 2021

Supervisors:

Katharina Schüller

Steve Chaplick

Enrique Hortal

Abstract

The graph coloring problem is a practical method of representing various real world problems including time scheduling, frequency assignment, register allocation, pattern matching, schedule optimization , airline traffic and many other situations where the accurate algorithms can handle pattern problems and enhance their functionality. The most important reason that makes the Graph Coloring Problem interesting is an all-inclusive approach to find the chromatic number of a certain graph, using various methods to determine the best lower and upper bounds. This is NP-difficult. In graph theory, a graph coloring is an assignment of colors to the vertices of a graph. Vertices that share an edge can not have the same color. Herein is our approach described to finding the chromatic number of a graph, using various methods to determine trivial bounds and if possible, the exact chromatic number. Infamous graph-traversing algorithms such as; Welsh Powell, A-RLF, Maximum Clique are used concurrently to create accurate upper and lower bounds and Recursive Backtracking for the exact solution. The fundamental methodology and approach of these algorithms is analyzed. Extensive descriptions regarding their results are given. Eventually, the effectiveness and accuracy of each algorithm is motivated for appropriate understanding. Since this is an NP-hard problem. Exact solutions for all graphs were not reached. Our research corresponds with existing research that smaller graphs can be solved in reasonable time, but when the graph size increases, so does the time complexity.

Contents

1 Introduction

The graph coloring problem (GCP) is as follows: Given an undirected graph G with a given number V that represents the number of vertices and E the number of edges. The assignment of colors to each vertex v ∈ V such that no two vertices connected by an edge e have the same color, using as few colors as possible. This project introduces the concept of graph coloring and allows us to research viable methods of analysing and coloring certain graphs. All graphs are undirected, but special cases exist, for example Wheel, Bipartite and Cycle graphs. *[14]* Not every graph can be approached the same way. Coloring a graph optimally requires multiple considerations. First, the size of a graph is crucial to determining whether it can be completed by raw computational power. The density of a graph is also crucial in determining whether the chromatic number is low or high (compared to the amount of vertices). Very dense graphs almost always have a higher chromatic number. The highest chromatic it can have is equal to the number of vertices. In this case we are talking about a completed graph. In completed graphs, all vertices are connected to all other vertices, allowing for no leniency. This report discusses our approaches and methods used for this project.

2 Project Approach

2.1 Graphs

Phase 3 provided us with 20 graphs to test our research and algorithms on. Almost all of these graphs had <1000 vertices and <2000 edges. With one distinct exception graph that had roughly 4000 vertices and 1.2 million edges.

2.2 Methods

We considered these graphs to still be of reasonable size. For this reason we chose to research more conventional algorithms. Graph analysis is also a big part of our implementation. We check for several special structures, because most special structures such as “Wheel”, “Cycle”, “Bipartite” have already existing solutions. *[14]* Therefore checking for these structures is very helpful in solving them.

2.3 Implementation

Our program has 6 distinct steps. In between every step we check our boundaries and chromatic number. Our program terminates as soon as the chromatic number is found.
First we load the graph in a custom graph class that allows for multiple ways of storing the data. The most used format for our algorithms is an adjacency matrix form.
Second, we create some trivial bounds based on math formulas. These are almost instant for any graph size and allow us to have starting boundary guidelines to work with.
The third step is checking for the graph structures. Bipartite, Cycle and Wheel, In that order. If the graph consists of such a structure, the boundaries are updated. The program then

tries calculating more accurate lower and upper bounds with the following algorithms: Max Clique, Welsh Powell, and Alternative-Recursive Largest First. *[4]*
After this step we start reducing the problem size by pruning vertices of the graph. All vertices with less edges than our **MinP**, which is our prune value indicator, are removed. The pruned matrix of the graph is also ordered based on the vertex degree. This ordering allows for faster exact backtracking results. Which is what we deploy last until timeout. All values are continuously updated and posted when better results have been found for our Lower Bound **(LB)**, Upper Bound **(UB)** or The Chromatic Number **x(G)**.

3 Research Questions

Why is it studied?

Graph coloring is used in various real world implementations. It can enhance the functionality of pattern related situations. A very common task where this topic is used is in scheduling tasks. Where the vertices and edges will represent the scheduled tasks and attenders.

What solutions do or do not exist?

The Graph coloring problem can be approached with a set of distinct algorithms to calculate trivial bounds or possibly even the exact chromatic, but no algorithm can solve any graph of any proportion in reasonable time, yet.

Does our research correspond with existing research?

Since we use many conventional methods to tackle this problem. We can fairly certain say yes to this question. Time complexity remains a large issue in calculating larger graphs. Solving small graphs can be done in "reasonable" time, but immense graphs are still nearly impossible to calculate exactly.

Does the graph size have an impact on the approach?

The graph size has a great impact on the approach. Determining the structure of a graph can often result in almost instant solutions. There already exist certain theorems that can solve very specific graph structures.

Will a perfect solution ever be found?

A perfect solution is currently very unlikely due to the fact that graphs can be scaled to infinity. At some point even loading the graph would already be too computationally heavy for our current hardware. This would suggest that only alternative approaches could ever solve this problem to perfection.

4 Algorithms

4.1 Bound Algorithms

4.1.1 Maximum Clique

The maximum clique of a given graph is a very solid way to determine a lower bound for the chromatic number. All graphs presented during this project were still of reasonable size, which made the time complexity of this algorithm not an issue. The decision for this algorithm's implementation was to find a quick and accurate lower bound.

For any given graph **G**. The algorithm creates a set with all vertices **V**. It then takes a vertex **v** from this set and searches for all connected vertices inside **V** and adds them to a set **P**. Let **P** be the set with all vertices connected to the current **v**. It then enters a recursive step where it takes a new current **v(P)** and searches for connected vertices inside **P**. It does this for every vertex inside **P** and when finished it will store the highest clique it could find. The original **v** is then removed from **V** and **P**. A new **v** is chosen from the set **V'** and the process enters its cycle again. After every **v** in **V** has been checked the maximum clique is stored inside a variable **MaxCliq**.
An optimization step was implemented where the algorithm stops if the number of vertices inside **V** and **P** combined is smaller than the current stored **MaxCliq**, as it is impossible to find a bigger clique with the remaining vertices.

4.1.2 Welsh Powell

The Welsh Powell is a variation of the greedy algorithm approach. It colors all vertices based on their degree. Where **v1** is the vertex with the highest degree. This selection continues along all vertices until they are colored and an upper bound is established. The benefit of starting with the vertex with the highest degree is due to the fact that this vertex has the highest number of constraints related to other vertices. The result this algorithm finds is always at most one more than the graphs maximum degree. The algorithm is rather quick as it never has to traverse a certain vertex twice. Once colored it is only ever returned to for checking whether it is adjacent to a current vertex that remains to be colored.

4.1.3 Alternative – Recursive Largest First

This algorithm is a variant of the already existing Recursive Largest First. *[4]* This is a popular greedy heuristic for the vertex coloring problem. It builds color classes and adds all possible vertices to this class before moving on to the next. It does this based on greedy choices. These greedy choices have an impact on the performance and that is why we introduce a slightly different variant. Let **C** be the next color class, let **U** be the set of uncolored vertices and let **W** be the set of uncolored vertices with at least one neighbour in **C**. Every time a vertex **v** from **U** is selected, all its neighbors are moved from **U** to **W**. The first **v** that is selected each time gets added to **C**. The second **v** gets chosen based on the amount of neighbours it has in **W**. This process continues until **U** is empty. A new color class is created as soon as **U** is empty and the process continues until all vertices are in a color class. The main alteration we added was that for every **v** it also checks for the value of least amount of neighbours in **U**.

4.1.4 Exact Backtracking (Recursive)

The backtracking algorithm is an exact algorithm. This means that it can determine true or false for a given amount of colors **X**. It will always either find a solution (and return true) or return that it is impossible to color this graph with **X** colors (and return false). This algorithm is used as a last resort when we can not determine the **x(G)** of a graph with all other methods we employ.
It will start at vertex 0 and slowly work its way up to the last vertex. Assigning a color to each vertex along the way. During this assignment it is constantly checked whether a color is valid. If there is no color available for the current vertex, it then returns to the last vertex, through a recursive step, which still has other color options available. This process continues until either all vertices are assigned valid colors or the recursion brings it back all the way to vertex 0. When this happens it means that there is no valid coloring available.

4.2 Graph Structure Algorithms

4.2.1 Bipartite Algorithm

A graph is bipartite when it can be divided into two different sets, where each node from each set can connect to vertices from the other set, but never with vertices from the same set. In this case, the chromatic number must be 2. The bipartite algorithm begins with a first color **k = 1** and then chooses a first vertex to color. Then it moves on to all vertices connected to this starting node and colors them with a second color **k = 2**. After this, the algorithm repeats this with each of these connected vertices and colors the vertices connected to these with **k = 1**. Then it checks all edges for a case where both vertices share the color 1 or 2, in which case the graph can't be bipartite. This algorithm repeats recursively until it can't find any other uncolored vertices.

To account for disconnected subgraphs, every vertex is stored in an **unseen_vertices** array to begin with. The first vertex chosen by the algorithm is the first vertex in this array and as it passes through the graph it erases each respective vertex from the array. Once the algorithm has gone through the entire subgraph and **unseen_vertices** isn't empty, it begins again by choosing the first vertex in the array as the starting point. This prevents the algorithm from falsely flagging a graph as bipartite if one subgraph can be coloured with two colours, while a disconnected subgraph can be connected with 3, for example.

4.2.2 Special Structure Algorithms

Depth-first search is applied when checking whether a given undirected graph is a circle or wheel graph. *[3]* Depth-first search is a traditional recursive method for systematically searching every vertex and edge in a graph. Every time visiting a vertex, we first mark it as already visited and then use recursion to visit all the vertices adjacent to it that have not yet been marked.

Once an adjacent vertex has been marked as visited and it is not the parent of the current vertex, this indicates that a cycle has been found. When the graph contains cycles and the number of adjacent vertices of all vertices is 2, the graph is considered as a circle graph. *[12]* And for a wheel graph, is defined as an C_{n-1} with one additional vertex which is connected to all the vertices in the cycle. Hence, for each W_n contains n vertices and 2(n-1) edges. We check the number of vertices and edges after a cycle detection of the given graph.

4.2.3 Vertex Pruning Algorithm (Reducing Problem Size)

This algorithm is mainly used to optimize the backtracking algorithm. It reduces the time it takes to find invalid colorings. It does this by removing as many vertices as possible that do not have any effect on the chromatic number. The most obvious one is all vertices with 0 edges. These can always be colored, because they have no restrictions. Secondly we also remove all vertices with only 1 edge as soon as we know the graph's chromatic number is 2 or higher. This also works, for the following reason. If a graph has at least 2 colors then any vertex that is only connected to 1 other can always use the opposite color. After the vertex pruning part is completed it executes one more the final step. This step is sorting the matrix based on the vertex degree. This has also proven to be impactful on the backtracking speed as it covers the most restricted vertices first. Resulting in quicker results when solutions for the current provided color amount can not be found.

5 Experiments

5.1 Experiment Conditions

The experiments are conducted on a workstation that carries an Intel Core i5-8257U Processor and 1.9GiB memory space. The installed OS is Ubuntu 20.10.

We tested all graphs from both Phase 1 and Phase 3 to gather official results.

5.2 Algorithm Performance Comparison

Algorithm performance results for all Phase 1 Graphs. Automatic timeout after 120s.

Graph	BP		MC		WP		A-RLF		A-RLF (V2)	
	LB	RT (ms)	LB	RT (ms)	UB	RT (ms)	UB	RT (ms)	UB	RT (ms)
1	3	0	11	21	11	12	11	29	11	36
2	3	1	22	27	15	365	21	9	22	5
3	3	0	4	4	3	6	4	13	4	2
4	3	0	3	7	2	9	3	4	3	9
5	2	9	3	2	3	16	2	28	2	70
6	3	0	8	26	3	12	5	25	5	16
7	3	0	5	18	2	5	4	6	4	11
8	3	1	12	21	7	10	10	7	10	24
9	3	4	10	23	10	14	10	14	10	9
10	3	11	8	75	3	49	6	154	6	182
11	3	35	31	169	31	15167	31	88	31	228
12	3	25	8	165	3	26	8	289	8	345
13	2	0	2	31	2	33	2	10	2	18
14	3	1	16	23	16	20	16	11	16	17
15	3	0	6	2	3	6	6	32	6	23
16	3	102	-	-	-	-	-	-	-	-
17	3	0	6	36	4	3	5	31	5	47
18	3	1	4	18	4	15	4	18	4	28
19	3	9	31	44	31	1	31	21	31	18
20	3	13	10	48	3	16	8	33	8	15

Algorithm performance results for all Phase 3 Graphs. Automatic timeout after 120s.

Graph	BP		MC		WP		A-RLF		A-RLF (V2)	
	LB	RT (ms)	LB	RT (ms)	UB	RT (ms)	UB	RT (ms)	UB	RT (ms)
1	3	0	6	18	8	72	6	150	6	133
2	3	12	2	45	3	99	3	653	3	624
3	3	1	3	43	6	66	5	106	6	99
4	2	52	2	142	3	458	2	2363	2	2083
5	3	1	5	80	10	165	7	371	7	401
6	3	22	10	38	11	118	11	151	11	217
7	3	1	3	53	3	78	3	215	3	474
8	3	0	4	18	7	63	6	93	5	75
9	3	0	8	233	12	55	12	1	12	9
10	3	26	8	156	9	465	9	982	9	25914
11	3	1	9	68	14	151	11	63	11	61
12	3	0	2	31	4	66	4	196	4	315
13	3	1	11	39	11	89	11	175	11	224
14	3	1	3	59	5	352	4	1402	4	3010
15	2	4777	2	4332	4	6919	2	53361	2	47147
16	3	6	98	289	98	83	98	13	98	14
17	3	1	15	18	15	52	15	67	15	87
18	3	5	3	101	5	221	4	489	4	711
19	3	4	8	14	8	4	8	30	8	47
20	3	1	2	17	3	22	3	65	3	143

(For each graph, five runs were executed)

6 Results

6.1 Phase 1 Graphs

Final results for all phase 1 graphs. Automatic timeout after 120s.

Graph	|V|	|E|	LB	UB	Best k	x(G)
1	76	303	3	11	11	11
2	61	1390	15	21	-	19
3	18	46	4	4	4	4
4	11	16	3	3	3	3
5	93	92	2	2	2	2
6	69	409	5	5	5	5
7	43	127	4	4	4	4
8	56	483	7	7	7	7
9	41	184	10	10	10	10
10	209	1249	4	6	-	5 or 6
11	191	3888	31	31	31	31
12	195	2365	5	8	-	8
13	61	448	2	2	2	2
14	31	385	16	16	16	16
15	41	205	6	5	5	5
16	867	18771	3	70	-	54
17	81	293	4	4	4	4
18	52	52	4	4	4	4
19	32	466	31	31	31	31
20	82	811	6	8	-	7

6.2 Phase 3 Graphs

Final results for all phase 3 graphs. Automatic timeout after 120s.

Graph	|V|	|E|	LB	UB	Best k	x(G)
1	218	1267	6	6	6	6
2	529	271	3	3	3	3
3	206	961	4	5	4 - 5	-
4	744	744	2	2	2	2
5	215	1642	5	7	5 - 7	-
6	131	1116	10	10	10	10
7	212	252	3	3	3	3
8	107	516	5	5	5	5
9	43	529	10	12	10 - 12	-
10	387	2502	8	9	8 - 9	-
11	85	1060	9	11	9 - 11	-
12	164	323	3	4	3 - 4	-
13	143	498	11	11	11	11
14	456	1028	3	4	3 - 4	-
15	4007	1198933	2	2	2	2
16	1007	4955	98	98	98	98
17	164	889	15	15	15	15
18	907	1808	3	4	3 - 4	-
19	106	196	8	8	8	8
20	166	197	3	3	3	3

7 Discussion

7.1 Graph Analysis

Analyzing graphs before running computationally-expensive algorithms appears to prove helpful. It allows for reducing the graph into simpler components, which can then be checked much faster. On certain occasions this allows us to find the exact chromatic number before these expensive algorithms are run. Graph 15 (phase 3) is a perfect example, despite it being the largest graph provided, it was solved rather quickly due to proper analysis. The analysis showed that this was a bipartite graph, so the chromatic is instantly found as 2.

7.2 Upper Bounds

Both the Welsh Powell and A-RLF performed rather well. In most cases their results equaled the chromatic number. Although the Welsh Powell is a little less accurate sometimes, its power lies in its speed, as it is often faster than the A-RLF. The A-RLF however gives the most accurate upper bound results in our program. The difference between the 2 different versions of the A-RLF is small, but still noticeable enough for us to include it. The main reason is that on specific occasions V2 actually returned a lower UB than the V1. Which is ofcourse a massive help to finding the chromatic. The reason why we kept the original A-RLF as well is because there were also, although they were very rare, situations where the V1 actually got a lower UB than the V2.

7.3 Lower Bounds

For the lower bound we mainly used the Max Clique algorithm. This searches for the smallest completed subgraph and will always give very accurate lower bounds. Often resulting in the chromatic number. The runtime for this algorithm is also quite short. Even on larger graphs it almost always finishes within seconds. The final outcome of our LB is also the input for our exact backtracking algorithm.

7.4 Exact Chromatic

To verify our chromatic number we use 2 methods. Firstly the most obvious one is comparing our lower and upper bound. If these are the same then there is no gap and we have found the Chromatic number. If this is not the case we continue with our Exact Backtracking algorithm. This recursively checks for existing solutions with the provided color amount. The input for our backtracking algorithm is the lower bound. The reason for this is because it can often find out faster whether a solution does not exist rather than the opposite. This is due to our pruning and sorting algorithms. We sort the adjacency matrix beforehand based on the vertex degree. This allows the backtracking to start on the most restricted vertices first and can often find out whether certain color solutions exist. We consider the matrix sorting to be our most valuable optimization step for graphs with high chromatic numbers. As well as cases where our bounds are quite far apart.

8 Conclusion

During this project, several approaches were used to do both analysis on our graphs as well as algorithmic calculation to find the best bounds or the chromatic. By using several graph structure analysis methods, we were capable of finding several chromatic numbers. The bipartite analysis being the most prominent one. Accurate lower and upper bounds were obtained using the rest of the algorithms. Overall we consider our performance on the graphs rather decent, as our biggest gap in bounds is 1 (on the phase 3 graphs). We tested on all Phase 1, Phase 3 and some custom graphs to determine our programs strengths and weaknesses.

The graph coloring problem is an NP-hard problem, which makes building perfect algorithms practically impossible. We tried our best to get as close to perfect as we could given the time and participant constraints.

9 References

[1] Gonthier, Georges (2008). Formal Proof: The Four-Color Theorem. Notices of the American Mathematical Society, vol 55(11), 1383-1384.

[2] Tarjan, et al. (1973) Algorithm 447: Efficient Algorithms for Graph Manipulation Communications of the American Mathematical Journal, vol 16(6): 373-378.

[3] Algorithms , 4TH Edition. Graphs. http://algs4.cs.princeton.edu/home/ . (17-07-2020). (10-01-2021).

[4] Mourchid Adegbindin , Alain Hertz , Martine Bellaıche. A new efficient RLF-like Algorithm for the Vertex Coloring Problem. (2/11/2015). RLFPaper.pdf. (07-01-2021).

[5] BEN STEVENS. COLORED GRAPHS AND THEIR PROPERTIES. stevens.pdf (whitman.edu). (11-01-2021).

[6] Bharathi S N. A Study on Graph Coloring.(05-2017).PROPER REFERENCE.pdf. (11-01-2021).

[7] Lewis, R.M.R. (2016). A Guide to Graph Colouring: Algorithms and Applications. Springer International Publishing. 18, 44, 45

[8] DIMACS. DIMACS Graphs: Benchmark Instances and Best Upper Bounds. (16-1-2017) http://www.info.univ-angers.fr/pub/porumbel/graphs/ Cliques, coloring, and satisfiability. Proceedings of the second DIMACS implementation challenge. Notices of the American Mathematical Society, vol 26.

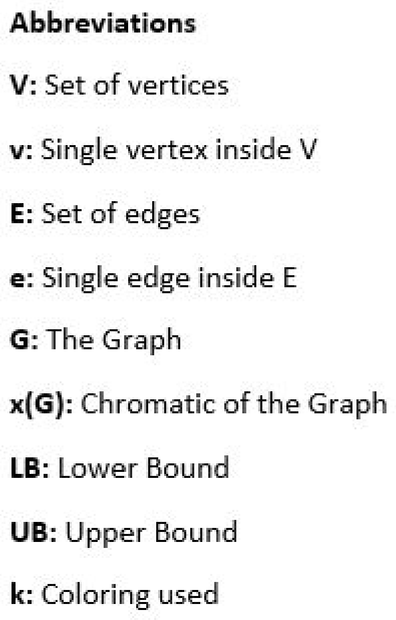
[9] Welsh DJA, Powell MB. (1967). An upper bound for the chromatic number of a graph and its application to timetabling problems. The Computer Journal.

[10] Battista, et al. (1994) Algorithms for Drawing Graphs: an Annotated Bibliography. Department of Informatics and Systems Engineering, University of Rome.

[11] Fahle, Torsten Simple and Fast: Improving a Branch-And-Bound Algorithm for Maximum Clique. (21-1-2017) http://www.dcs.gla.ac.uk/pat/jchoco/clique/indSetMachrahanish/papers/fahle.pdf

[12] tutorialspoint. Graph Theory – Graph of Types. http://www.tutorialspoint.com/graph_theory/types_of_graphs.htm . (2006) . (10-01-2021).

10 Appendix



Abbreviations

V: Set of vertices

v: Single vertex inside V

E: Set of edges

e: Single edge inside E

G: The Graph

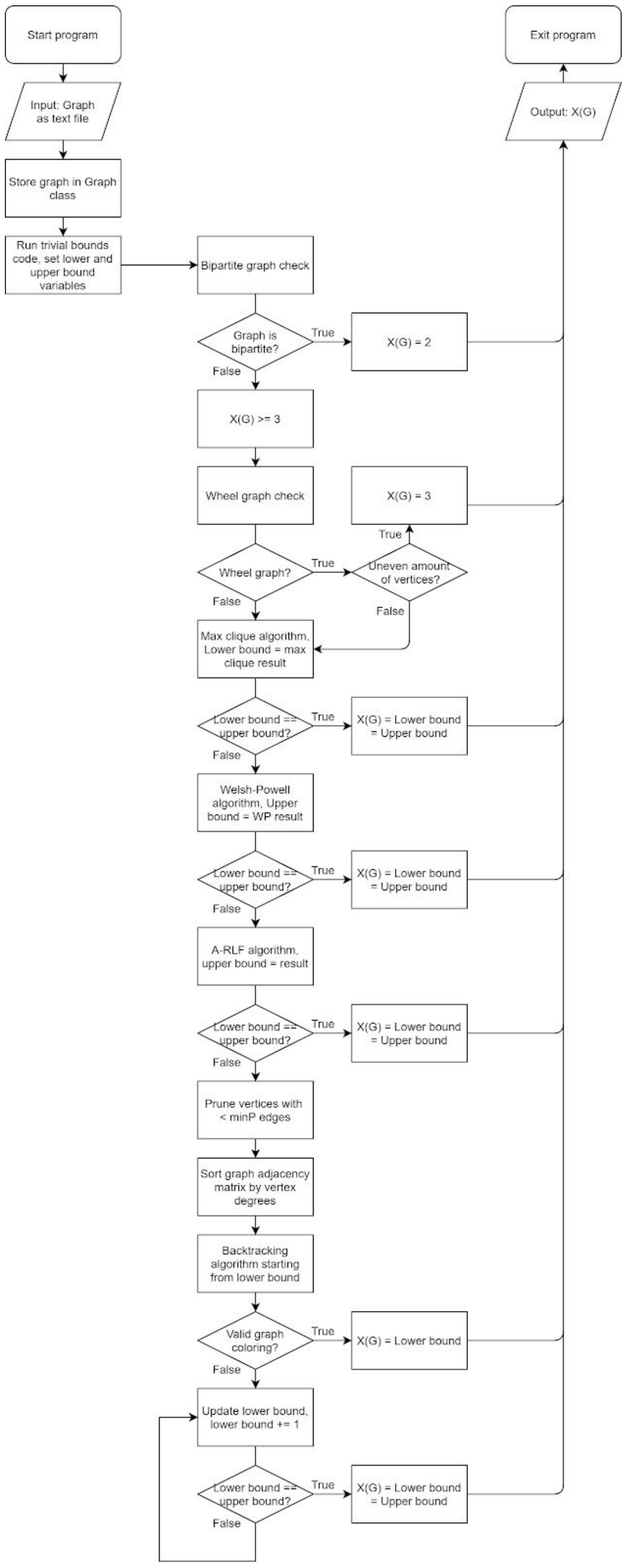
x(G): Chromatic of the Graph

LB: Lower Bound

UB: Upper Bound

k: Coloring used

Flowchart


Start program
Input: Graph as text file
Store graph in Graph class
Run trivial bounds code, set lower and upper bound variables
Bipartite graph check
Graph is bipartite?
True
X(G) = 2
False
X(G) >= 3
Wheel graph check
X(G) = 3
True
Wheel graph?
True
Uneven amount of vertices?
False
False
Max clique algorithm, Lower bound = max clique result
Lower bound == upper bound?
True
X(G) = Lower bound = Upper bound
False
Welsh-Powell algorithm, Upper bound = WP result
Lower bound == upper bound?
True
X(G) = Lower bound = Upper bound
False
A-RLF algorithm, upper bound = result
Lower bound == upper bound?
True
X(G) = Lower bound = Upper bound
False
Prune vertices with < minP edges
Sort graph adjacency matrix by vertex degrees
Backtracking algorithm starting from lower bound
Valid graph coloring?
True
X(G) = Lower bound
False
Update lower bound, lower bound += 1
Lower bound == upper bound?
True
X(G) = Lower bound = Upper bound
False
Output: X(G)
Exit program