Project 1.1 - Block 3

Compute chromatic numbers

Group 10

Tu Anh Dinh
Michal Jarski
Vaishnavi Velaga
Rudy Wessels
Oskar Wielgos

January 23, 2019

Maastricht University

Department of Data Science and Knowledge Engineering

Project 1.1 - Block 3

Compute chromatic numbers

Group 10

Tu Anh Dinh
Michal Jarski
Vaishnavi Velaga
Rudy Wessels
Oskar Wielgos

January 23, 2019

Project coordinator: Prof. Jan Paredis

Preface

This report is the outcome of group 10's work for project 1.1: Graph Coloring. It can be used as a guideline to partially solve the NP-complete problem of finding the chromatic number of a graph.

Summary

The chromatic number of a graph is the minimum number of colors needed to color the vertices of the graph such that no two adjacent vertices share the same color. Finding the chromatic number of a graph is a NP-complete problem, which has many applications such as automated timetabling and scheduling. This project solves the problem of finding the smallest range of the chromatic number of a graph. We propose an approach that combines different methods such as graph decomposition, greedy algorithms, identifying special cases of graphs, a brute-force algorithm and genetic algorithm. The proposed approach is analyzed on a set of 20 graphs. It gives decent results on graphs with special structures, even if the graphs' sizes are big. However, for large graphs without special structures, the results are unpredictable.

Contents

Chapter 1

Introduction

The chromatic number of a graph is the minimum number of colors needed to color the vertices of the graph such that no two adjacent vertices share the same color. In this project, we provide an approach to find the smallest range of the chromatic number of a graph.

Formally, a graph is defined as G=(V,E) where V is the set of vertices and E is the set of edges that connect the vertices. In this project, the input graphs are undirected graphs. In addition, a vertex must not connect to itself and the number of edges between two vertices must not exceed 1. Graph coloring is the problem of coloring the vertices of a graph such that no two adjacent vertices share the same color. Graph coloring is one of the important topics of graph theory and is used in various research areas of computer science such as data mining and networking.

The smallest number of colors used in graph coloring is called the chromatic number. The purpose of this project is to find the chromatic number of a graph. If the chromatic number cannot be found, we try to find the closest upper bound and lower bound.

The approach proposed for this project is to combine different methods such as graph decomposition, greedy algorithms, identifying special cases of graphs, a brute-force algorithm and genetic algorithm. A greedy algorithm is used to find the lower bounds; while another greedy algorithm and the genetic algorithm are used to find the upper bounds. Algorithms for testing special cases and brute-force algorithm are used to find the chromatic numbers if possible. Graph decomposition helps impove the execution time and the results of other methods. The approach is then analyzed on a set of 20 graphs, where the structure of each graph is exploited to see the correlation between the types of graphs and the performance of the proposed approach on those graphs. The approach gives decent results on graphs with special structures, even if the graphs' sizes are big. Nevertheless, for large graphs without special structures, the results are unpredictable.

The report is organized as follows. Chapter 2 discusses the methods used to compute the upper bound, lower bound and the chromatic number of a graph.

Different algorithms are used: graph decomposition, greedy algorithm, identifying special cases of graphs, brute-force algorithm and genetic algorithm. Chapters 3, 4 and 5 present experiments and their results. The last chapter draws conclusions and possible future work.

Chapter 2

Methods

This chapter describes the approach used for finding the lower bound, upper bound and if possible, the chromatic number of a graph.

2.1 Overview

The proposed approach is a combination of different methods. Since the execution time is normally limited, methods that give out results fast are executed first. Algorithm 1 describes the general work flow.

The proposed approach works as follows. First, a greedy algorithm calculates an upper bound of the given graph (line 1). Then, the given graph is decomposed into connected components. Each component is checked to see if it is one of the special cases where the chromatic number can be concluded immediately (line 3 - 7). Five special cases are considered.

- No-vertex graph: chromatic number is 0
- No-edge graph: chromatic number is 1
- Bipartite graph: chromatic number is 2
- Odd cycle graph: chromatic number is 3
- Complete graph: chromatic number is the number of vertices

If a component is none of the special cases then a lower bound of the component is 3, since the first three cases have covered all graphs with chromatic number below 3 (line 8). It is then compared to the lower bound computed by a greedy algorithm to decide the final lower bound (line 10 - 12). Then an upper bound of the component is computed by another greedy algorithm (line 13). The reason for running the greedy algorithm for upper bound on both the original graph and its components is that the resulting upper bounds are sometimes different, so we run both on the original graph and its components to increase the chance that a better upper bound is found. However, the greedy algorithm for lower bound normally gives the same results after running on both the original graph and its components, so it is run only on the components. Next, if the lower bound and upper bound are equal, then the chromatic number can be

Algorithm 1 General work flow

```
Require: a graph
upperbound = greedyUpperbound(graph)
components = decompose(graph)
for all components do
  if component is special case then
    chromaticNumber of component = chromaticNumber of special case
    Go to the next component
  else
    componentLowerbound = 3
  end if
  if greedyLowerbound(component) > 3 then
    componentLowerbound = greedyLowerbound(component)
  end if
  componentUpperbound = greedyUpperbound(component)
  if componentUpperbound = componentLowerbound then
    chromaticNumber of component = componentUpperbound
    Go to the next component
  end if
  if number of vertices <= 20 then
    chromaticNumber of component = BruteForce(component)
  end if
end for
newUpperbound = max(upper bounds of components)
if newUpperbound < upperbound then
  upperbound = newUpperbound
end if
lowerbound = max(lower bounds of components)
if has found all chromatic numbers of components then
  chromatic number = max(chromatic numbers of components)
else
  lowerbound = max(chromatic numbers of components)
end if
geneticAlgorithm(graph)
```

concluded (line 14 - 17).

If the chromatic number of a component still cannot be calculated, a brute-force algorithm is used (line 18 - 20). However, only components with number of vertices below 20 are proccessed with the brute-force algorithm since the brute-force algorithm normally takes a long time to execute on bigger graphs. Without time restriction, it might be possible to increase the threshold for brute-force.

After proccessing on the components, the upper bound, the lower bound and possibly the chromatic number of the original graph can be concluded. The biggest upper bound among the components is an upper bound of the original graph (line 22). This new upper bound is then compared to the previously computed upper bound (line 1) to output the better one. Similarily, the biggest lower bound among the components is a lower bound of the original graph (line 26).

If the chromatic numbers of all components have been found, then the chromatic number of the original graph is the biggest chromatic number among those of the components (line 27 - 29). If it is not the case, then the biggest chromatic number found on the components is another lower bound for the original graph (line 30).

Finally, genetic algorithm is used to bring the upper bound closer to the chromatic number (line 32). Genetic algorithm is run last because there is no guarantee on its execution time.

The algorithm for each method is described in the next sections of this chapter.

2.2 Graph decomposition

One graph can contain multiple disconnected parts, which can be considered as independent subgraphs. Graph decomposition means seperating the graph into fully connected components. Figure 2.1 illustrates a graph before being decomposed and after being decomposed. Decomposing the graph allows other methods to work on smaller graphs.

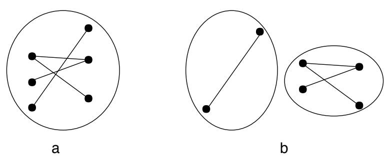

Figure 2.1: A graph (a) before decomposed and (b) after decomposed

The proposed graph decomposition algorithm is based on breadth-first search. A none-classified vertex from the original graph is first added to a component. Then the vertex is expanded to find all its neighbors and the neighbors are added to the same component. The same is done for all neighbors and the neighbors' neighbors, etc. untill no more vertex connecting to the component's vertices can be found. The process is repeated until all vertices in the original graph are classified into connected components.

Note that the vertices of the input graphs in this project are represented by successive numbers. All algorithms are implemented based on this data structure. Therefore, after classifying the vertices into components, each component is then converted to the standard form, where the indexes of vertices are successive numbers.

2.3 Greedy algorithm for upper bound

Greedy algorithm for upper bound (Jensen & Toft, 2011) provides an efficient way of coloring a graph. However, it does not guarantee that the coloring is optimal. Therefore, greedy algorithm can be used to calculate an upper bound.

The greedy algorithm works as follows. First, the vertices are sorted based on their degrees. The degree of a vertex is the number of other vertices connected to that vertex. The vertex with higher degree will be colored first. When coloring a vertex, the available colors is reused as much as possible. If none of the available colors is valid to color that vertex, then a new color is generated and added to the available list. When the graph is fully colored, the number of colors in the available list is returned as an upper bound.

2.4 Greedy algorithm for lower bound

To find the lower bound, a greedy algorithm is used to find several cliques in a graph and returns the size of the biggest clique found (Steven, 2008). However, it does not guarantee to find the maximum clique.

The algorithm works as follows. To form a clique, an initial vertex is added to the clique. Then, for every other vertex, we add it to the clique if it is connected to all vertices currently in the clique. The same proccess is repeated to find multiple cliques, where every vertex in the original graph is used as the initial vertex for a clique. Since the proccess is repeated multiple times, the bigger the clique is, the better chance that it can be found. Although the algorithm does not guarantee to find the maximum clique, there is a high chance that the algorithm can find the largest cliques or cliques with sizes close to the largest clique.

2.5 Special cases

This section describes the algorithms used to test whether a graph is one of the special graphs. These algorithms have low complexity, therefore their execution times are short, even when the input graphs are large. However, the uses of these algorithms are limited, since the special cases tested are only a few of many types of graphs.

2.5.1 Bipartite

A bipartite graph is a graph where vertices can be separated into two sets such that there is no connection between any pair of vertices of the same set. The chromatic number of a bipartite graph is 2. An algorithm based on breadth-first search (Sedgewick & Schidlowsky, 2003) is used to test whether a graph is bipartite.

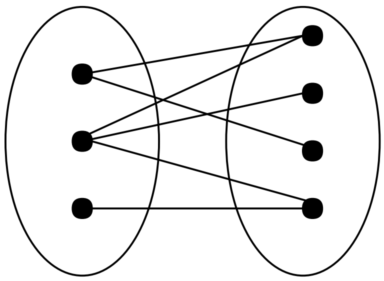

Figure 2.2: An example of bipartite graphs

The algorithm uses two colors to color the graph. First, a none-colored vertex from the graph is assigned with one color. Then the vertex is expanded to find all its neighbors. The neighbors are then assigned with the other color. The same is done for all neighbors and the neighbors' neighbors, etc. until the graph is fully colored. While coloring the graph, if an invalid coloring is found, then the graph is not bipartite. Otherwise, if the graph can be successfully colored with two colors, then it is a bipartite graph.

2.5.2 Odd cycle

An odd cycle is a cycle with an odd number of edges and vertices. Figure 2.3 shows an example of odd cycle graphs. The chromatic number of an odd cycle graph is 3.

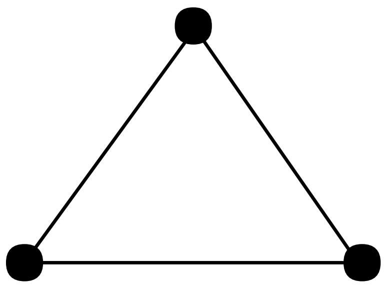

Figure 2.3: An example of odd cycle graphs

The method for testing if a graph is an odd cycle checks for three conditions:

- The number of vertices is equal to the number of edges

- Every vertex has two neighbors
- The number of vertices is odd

A graph is an odd cycle graph if and only if all three conditions are satisfied.

2.5.3 Complete graph

A complete graph is a graph where every vertex is connected to all other vertices. An example of complete graphs is illustrated in Figure 2.4. The chromatic number of a complete graph is the number of vertices. The method checks whether a graph has the above conditions to determine if it is a complete graph.

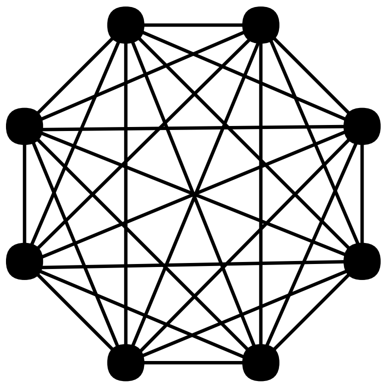

Figure 2.4: An example of complete graphs

2.6 Brute-force algorithm

The brute-force algorithm works as follows. Given a fixed number of colors, the brute-force algorithm generates every possible coloring and checks if it is a valid one. In order to check the validity of coloring, it iterates through the array representing vertices' colors and searches for a conflict, that is, two connected vertices assigned the same color. The number of colors gradually increases until a valid coloring is found. The first number of colors when a valid coloring is found is the chromatic number.

Brute-force algorithm has a high computational complexity. Although finding the chromatic numbers is guaranteed sooner or later, in reality, the execution time of brute-force depends on the sizes of input graphs and how big their chromatic numbers are.

The brute-force algorithm in the proposed approach uses the lower bounds calculated by the methods described above as the first number of colors, reducing the time needed to find the chromatic number.

2.7 Genetic algorithm

The genetic algorithm is an algorithm for calculating the upper bound together with working its way down to find smaller and better upper bounds for a particular graph. It starts with creating a population of individuals each containing a randomly colored version of the input graph. The size of this population can be set to any number preferred. Once the population is created, it assigns a fitness (which is a real number between 0 and 1) to each individual based on the number of incorrect edges (the edges that connects two vertices with the same color). Following up the individuals are sorted by fitness from high to low, at which it becomes clear what part of the population has the highest correctness of coloring.

After the individuals are sorted, the selection method picks out the parents for the next generation through an elitist approach. These parents are utilized for the crossover method to create combinations of two of the parents until there are enough new individuals for the next generation with equal size to the previous one. Afterward, the mutation method, depending on the extent of the mutation rate, will mutate some individuals' colorings of the graph to achieve possibly better results., that is, individuals with higher fitness.

Lastly, this process runs over several generations/populations through a loop until the algorithm finds an individual with fitness "1" (no incorrect edges). When this is the case, the new upper bound will be printed in the command prompt based on the number of colors used to achieve this solution. Following up the entire process starts over with one less color, so that the upper bound will be lower after each successful finding.

However, the downside of using this algorithm is that it does not know when the chromatic number is reached. When this is the case, the genetic algorithm gets stuck in an infinite loop. This might cause misinterpretation with larger graphs, because the user will not know if the chromatic number is reached or the algorithm is taking a longer time to complete.

By combining the greedy algorithm with the genetic algorithm, the genetic algorithm can be sped up. First of all, the genetic algorithm makes use of the upper bound which the greedy algorithm gathers and starts running from one less color than the upper bound to get the fastest start. Secondly, the speed is improved by taking the output of right coloring which the greedy algorithm generates. Instead of letting the genetic algorithm create the first generation randomly, it has the ability to utilize the right coloring by altering it partially. Through decreasing the number of colors by one, the right colored graph is still largely correct and thus outputting a high fitness in the genetic algorithm, giving it a head start and speeding it up quite a bit.

Chapter 3

Experiments

The experiments are set up to run on the given 20 graphs from phase 3 of the project. Each graph is exploited to see the following properties:

- The number of vertices (or the size of the graph)
- The number of connected components
- The size of each component
- The number of components that are special cases
- The biggest clique found

Chapter 4

Results

This chapter represents the results of the proposed approach on the given 20 graphs from phase 3 of the project. Table 4.1 shows the upper bounds, lower bounds and chromatic numbers found on the graphs, while table 4.2 shows the properties of the graphs.

Table 4.1: Results on the given 20 graphs from phase 3. The column "Gap" represents the differences between upper bounds and lower bounds

Graph no.	Upper bound	Lower bound	Chromatic number	Gap
1	3	–	3	0
2	5	3	–	2
3	6	6	6	0
4	5	4	–	1
5	2	–	2	0
6	3	–	3	0
7	12	8	–	4
8	98	–	98	0
9	6	3	–	3
10	3	–	3	0
11	15	15	15	0
12	2	–	2	0
13	11	9	–	2
14	5	3	–	2
15	10	5	–	5
16	4	3	–	1
17	8	–	8	0
18	11	10	–	1
19	11	11	11	0
20	9	8	–	1

Table 4.2: Properties of the given 20 graphs from phase 3, along with the gaps between upper bounds and lower bounds

Graph no.	Size	Number of components	Components' sizes	Special cases	Biggest clique found	Gap
1	212	1	212	0	3	0
2	456	8	448, 2, 1 (6 times)	7	3	2
3	218	2	212, 6	1	3	0
4	107	1	107	0	4	1
5	4007	1	4007	1	–	0
6	529	260	8, 5, 2 (258 times)	260	–	0
7	43	1	43	0	8	4
8	107	1	107	0	98	0
9	206	1	206	0	3	3
10	166	1	166	0	2	0
11	164	1	164	0	15	0
12	744	1	744	1	–	0
13	85	1	85	0	9	2
14	907	1	907	0	3	2
15	215	1	215	0	5	5
16	164	1	164	0	2	1
17	106	12	92, 2, 3, 1 (9 times)	11	8	0
18	131	5	26 (2 times), 25 (2 times), 29	0	10	1
19	143	1	143	0	11	0
20	387	1	387	0	8	1

Chapter 5

Discussion

As can be seen from the results, the proposed approach performs well on several types of graphs:

- For graphs that are special cases, the chromatic numbers can be found quickly, even when the graphs' sizes are large. For example, the sizes of graph 5 and graph 12 are 4007 and 744 respectively, but since they are bipartite graphs, their chromatic numbers were found in less than 1 minute. The reason for this is that the time complexities of the special case testing algorithms are low.
- For graphs that are disconnected, the algorithm also gives good results. For example, graph 2, 3, 6, and 17 can be decomposed into 8, 2, 260, 12 and 5 components respectively, so their ranges for chromatic numbers are small (less than or equal to 2).
- For graphs that have big clique sizes compared to the graphs' sizes, the gaps between the upper bounds and lower bounds found are small. For example, graph 8 has the size of 107 and the biggest clique size found is 98, so our algorithms can find its chromatic number. This shows that the bigger the clique is, the more likely it can be found.

On the other hand, the algorithms do not give good results on certain graphs that do not have special structures. For example, graph 15 is fully connected. The graph is none of the special cases and its biggest clique size found is small compared to the graph's size. Therefore, the range for the chromatic number of graph 15 is higher (where the difference between the upper bound and lower bound is 5).

Chapter 6

Conclusion

In this project, the problem of finding chromatic number for a graph is partially solved. This is a NP-complete problem, so the proposed approach is to use multiple combined algorithms to find the smallest range for the chromatic number. A greedy algorithm is used to find the lower bounds while another greedy algorithm and the genetic algorithm are used to find the upper bounds. Algorithms for testing special cases and a brute-force algorithm are used to find the chromatic numbers if possible. Graph decomposition helps impove the execution time and the results of other methods.

The experiment results show that the proposed approach perform well on graphs with special structures, even if the graphs' sizes are big. For large graphs without special structures, the results are unpredictable.

The results from the experiments give insights into the proposed approach. Greedy algorithms often give decent results with short execution time, but do not guarantee the optimal results. On the other hand, brute-force algorithm guarantees the optimal results, but has high complexity, therefore it can only be used on graphs with small number of vertices. Genetic algorithm normally gives better results than greedy algorithm, but the execution time is longer, and the results are also not guaranteed to be optimal. Special cases testing algorithms such as bipartite testing and complete graph testing have low complexity, thus they can give out the results fast. However, the use of special cases testing algorithms is limited on graphs with special structures.

The approach proposed in this report can be used as a starting point for solving the graph coloring problem. Advanced methods such as linear programming or constraint satisfaction can be used for further research.

References

Jensen, T. R., & Toft, B. (2011). *Graph coloring problems* (Vol. 39). John Wiley & Sons.

Sedgewick, R., & Schidlowsky, M. (2003). *Algorithms in java, part 5: Graph algorithms* (3rd ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Steven, S. (2008). The algorithm design manual. *Springer*.