Project 1.1 - Block 3

Compute chromatic numbers

Group 10

Tu Anh Dinh
Michal Jarski
Vaishnavi Velaga

Rudy Wessels
Oskar Wielgos

Submitted: Wednesday 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

Submitted: Wednesday 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. For large graphs without special structures, the results are unpredictable.

Contents

Chapter 1

Introduction

As stated above, 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.

The problem description is as follows. Given a graph 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 graph, where the edges have no orientation. In addition, a vertex must not connect to itself and the number of edges between two vertices must not exceed 1. Graph coloring is 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. A lower bound of a graph is a number that is less than or equal to the chromtic number, while an upper bound is a number that is greater than or equal to the chromtic number. The purpose of this project is to find the closest upper bound, lower bound and, if possible, the chromatic number of a graph.

The approach proposed for this project is combining 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 results on those graphs. The approach gives decent results on graphs with special structures, even if the graphs' sizes are big. For large graphs without special structures, the results are unpredictable.

The rest of the report is divided as follows. Chapter 2 is about 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 are about experiments and their results. The last chapter gives the conclusion for the project.

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 in this report 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 the 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). The special cases are listed below:

- 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

In line 8, 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. It is then compared to the lower bound computed by a greedy algorithm to update the bigger 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 result 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 concluded (line 14 - 17).

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)
```

If the chromatic number of a component still cannot be calculated then 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 long to execute on bigger graphs. Without time-restriction, the more time we have, the higher the threshold for brute-force can be (about 30 to 40).

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 a upper bound of the original graph (line 22). This new upper bound is then compared to the previously computed upper bound in 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, in line 32, genetic algorithm is used to bring the upper bound closer to the chromatic number. 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 (Figure 2.1). Graph decomposition means seperating the graph into fully connected components. Decomposing the graph will allow other methods to work on smaller graphs. Algorithm 2 describes the method for decomposing a graph into components.

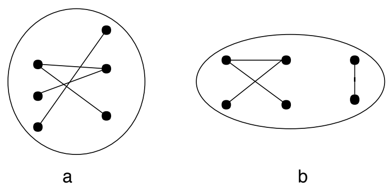

Figure 2.1: A graph (a) before decomposed and (b) after decomposed

The proposed graph decomposition algorithm is based on breadth-first search. A unchecked-list stores the vertices whose neighbors are not yet added to the current expanding component. In line 6 and 7 the first vertex is added to a component and the unchecked list. Then all the neighbors of the vertex are added to the component and the unchecked list, and the first vertex of the unchecked list is removed (line 9 - 19). To avoid loops, only vertices not in the component

Algorithm 2 Decomposing a graph

```
Require: a graph
 1: Create listOfVertices
 2: listOfVertices.add(all vertices in the graph)
 3: while listOfVertices is not empty do
 4:    Create a new component
 5:    Create a new uncheckedList
 6:    component.add(firstVertex in listOfVertices )
 7:    uncheckedList.add(firstVertex in listOfVertices)
 8:    listOfVertices.remove(first element)
 9:    while uncheckedList is not empty do
10:       checkingVertex = first vertex in the uncheckedList
11:       for all neighbors of checkingVertex do
12:          if neighbor is not in component then
13:             uncheckedList.add(neighbor)
14:             component.add(neighbor)
15:          end if
16:          listOfVertices.remove(neighbor)
17:       end for
18:       Remove checkingVertex from uncheckedList
19:    end while
20:    Convert component to standard form
21:    components.add(component)
22: end while
23: return components
```

are added. The same is done for all vertices in the unchecked list, until the list is empty. The process is repeated until all vertices in the original graph are classified into 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 (line 20).

2.3 Greedy algorithm for upper bound

Greedy algorithm for upper bound provides an efficient way of coloring a graph, with short execution time. However, it does not guarantee that the coloring is optimal. Therefore, it can be used to calculate a upper bound. Algorithm 3 (Jensen & Toft, 2011) describes this method.

First, the vertices are sorted based on their degrees (line 1). The degree of

Algorithm 3 Greedy algorithm for upper bound

```
Require: a graph
 1: Sort vertices in non-increasing order of degrees (the number of neighbors)
 2: Create availableColors list
 3: for all vertices do
 4:   for all colors in availableColors do
 5:     if color is valid for vertex then
 6:       Assign the color for the vertex
 7:       break
 8:     end if
 9:   end for
10:   if The vertex is still not colored then
11:     Create a new color
12:     Assign the new color for the vertex
13:     Add the new color to availableColors list
14:   end if
15: end for
16: return size of availableColors list
```

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 (line 4-9). If none of the available colors is valid to color that vertex, then a new color is generated and added to the available list (line 10-14). 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

A greedy algorithm is used to find several cliques in a graph and returns the size of the biggest clique found (Steven, 2008). The biggest size is then return as a lower bound. 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.

Even though the algorithm does not guarantee to find the maximum clique, it is still promissing that the algorithm will find cliques that are big compared to the graphs' sizes. Since the algorithm goes through every vertex in the graph and forms a clique that contains the vertex, the bigger the clique is, the better chance that it can be found.

2.5 Special cases

2.5.1 Bipartite

A bipartite graph is a graph which its 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. Algorithm 4 (Sedgewick & Schidlowsky, 2003) describes the steps to test whether a graph is bipartite, using breadth-first search.

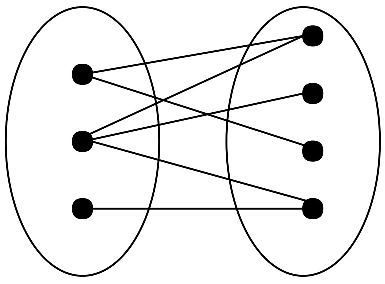

Figure 2.2: An example of bipartite graphs

Two colors are used to color the graph. A unchecked list stores the vertices whose neighbors are not yet considered. First, one color is assigned to the first vertex and the first vertex is added to the unchecked list (line 2-3). Then, all its neighbors are considered and the vertex itself is removed from the unchecked list. For each neighbor, if the neighbor has not been colored then it is assigned with the oposite color (line 10). If the neighbor has been colored, then we check if it is a valid coloring. If the coloring is invalid, the graph is not bipartite (line 12-13). The same is done for all elements in the unchecked list, until the list is empty. The process is repeated until all vertices in the graph are colored. If the graph is successfully colored, then it is bipartite.

Algorithm 4 Bipartite testing

```
Require: a graph
 1: Create unchecked list
 2: unchecked.add(first vertex)
 3: Assign one color to the first vertex
 4: while The graph is not fully colored do
 5:   while unchecked list is not empty do
 6:     checkingVertex = unchecked.getFirstElement()
 7:     unchecked.removeFistElement()
 8:     for all neighbors of checkingVertex do
 9:       if neibor not yet colored then
10:         Assign the opposite color of checkingVertex's color to neighbor
11:         unchecked.add(neighbor)
12:       else if neighbor has invalid color then
13:         return false
14:       end if
15:     end for
16:   end while
17: end while
18: return true
```

2.5.2 Odd cycle

An odd cycle is a cycle with an odd number of edges and vertices (Figure 2.3). The chromatic number of an odd cycle graph is 3.

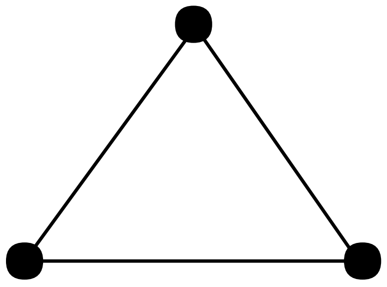

Figure 2.3: An example of odd cycle graphs

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

- 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 (Figure 2.4) is a graph where every vertex is connected to all other vertices. The chromatic number of a complete graph is the number of vertices. The method checks weather a graph has the above conditions to determine if it is a complete graph.

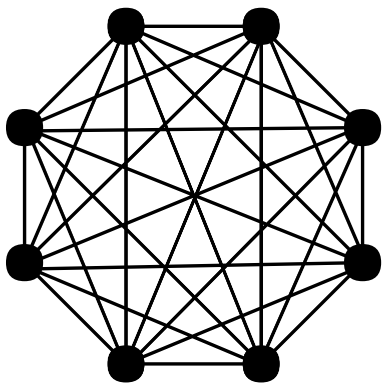

Figure 2.4: An example of complete graphs

2.6 Brute-force algorithm

The brute-force algorithm simply generates every possible coloring and checks if it is a valid one. In case the coloring is valid, the algorithm terminates with a possibilty of returning an array of colors assigned to each node and the chromatic number. In order to check the validity of coloring, it utilizes isValid algorithm, which iterates through the array representing nodes' colors and searches for a conflict (two adjacent/connected nodes assigned the same color). If a conflict is found, the method returns false, meaning certain coloring is not a valid one.

Brute-force is based on raw computetional power, thus making it heavily dependent on the hardware that runs it. Finding the chromatic number is guaranteed sooner or later. However in reality, its use is limited to graphs not bigger than 20 vertices, and even then, depending yet on how the vertices are being connected.

Algorithms calculating lower bound or applying pruning, might futher optimize it, reducing the time needed to find the chromatic number.

Implementing a greedy-type brute-force could also bring significant improvements on effectiveness and execution time, but on the other hand, causing a risk of omitting the right coloring and eventually not finding the chromatic number, but only its approximation.

2.7 Genetic algorithm

The genetic algorithm is an algorithm for calculating the upper bound together with working its way down to finding lower and better upper bounds for a particular graph. It starts with creating a population of individuals each containing a randomly colored version of the selected 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. Following up the individuals are sorted by fitness from high to low (1>0), 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 creating 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' coloring of the graph to achieve possibly better results. By results is meant, individuals with higher fitness.

Lastly, this process runs over several generations/populations through a loop till 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 how many colors were 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.

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	8	6	–	2
4	7	4	–	3
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	14	9	–	5
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 found

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	2
4	107	1	107	0	4	3
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	5
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, our algorithms perform well to find the chromatic numbers of several types of graphs:

- For graphs that are special cases, the chromatic numbers can be found quickly, even if 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 special cases testing algorithms are low, as mentioned in chapter 2.
- 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 is an example which proves that the bigger the clique is, the more likely it can be found, as stated in chapter 2.

On the other hand, the algorithms do not give good results on certain graphs that do not have special structures. For example, graph 13 and 15 are fully connected, they are none of the special cases and their biggest clique sizes found are small compared to the graphs' sizes. Therefore, their ranges for chromatic number are higher (where the differences between upper bounds and lower bounds are both 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 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 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 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 results on those graphs. The approach gives decent results 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, complete graph testing have low complexity, thus they can give out the results fast. However, the use of special cases testing algorithms are 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. More enhanced 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*.