Project 1.1 - Block 3

Compute chromatic numbers

Group 10

Tu Anh Dinh
Michal Jarski
Vaishnavi Velaga
Rudy Wessels
Oskar Wielgos

Submited: 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

Submited: Wednesday January 23, 2019

Project coordinator: Prof. Jan Paredis

Preface

This report is a part of the outcomes of our work for project 1.3: Graph Coloring. It can be used as a guideline to partially solve the non-deterministic polynomial-time hard problem of finding the chromatic number of a graph.

Summary

This project gives an answer for the question of finding the smallest range of the chromatic number of a graph. The methods used are graph decomposition, greedy algorithm, identifying special cases of graphs, brute-force algorithm and genetic algorithm. This approach gives decent results on small graphs and graphs with special structures. However, the results are unpredictable for large arbitrary graphs.

Contents

Chapter 1

Introduction

A graph is a set of vertices connected by edges. In this project, the input graphs are undirected graph, where the edges have no orientation. In addition, a vertex must not connected 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 topic of graph theory and is used in real time applications of computer science and it is used in various research areas of computer science such as data mining, networking etc.

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 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. In the last chapter, the whole project will be concluded.

Chapter 2

Methods

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

2.1 Overview

Given the limitation on execution time (2 minutes for each graph in the tournament), methods that give out results fast are executed first. Algorithm 1 describes the general work flow.

First, a greedy algorithm is run on the given graph to calculate the upper bound (line 1). Then, the given graph is decomposed to 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 5 - 24). 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: chromatic number is 3
- Complete graph: chromatic number is the number of vertices

Next, a lower bound and an upper bound are computed by greedy algorithms (line 25 and 26). The reason for running 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 cases to get the better upper bound. However, the greedy algorithm for lower bound normally gives the same results for both cases, so it is run only on the components In line 27, 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 where the chromatic number is below 3. It is then compared to the previous lower bound computed in line 26 to update the lower bound if needed. Next, if the lower bound and upper bound are equal, then the chromatic number can be concluded(line 29 - 32).

Algorithm 1 General work flow

Require: a graph

```
upperbound = greedyUpperbound(graph)
components = decompose(graph)
for all components do
    //Check all special cases
    if component has no vertex then
        chromaticNumber of component = 0
        Go to the next component
    end if
    if component has no edge then
        chromaticNumber of component = 1
        Go to the next component
    end if
    if component is bipartite then
        chromaticNumber of component = 2
        Go to the next component
    end if
    if component is odd cycle then
        chromaticNumber of component = 3
        Go to the next component
    end if
    if component is complete graph then
        chromaticNumber of component = number of vertices
        Go to the next component
    end if
    subUpperbound = greedyUpperbound(component)
    subLowerbound = greedyLowerbound(component)
    newSubLowerbound = 3
    Update subLowerBound if needed
    if subUpperbound = subLowerbound then
        chromaticNumber of component = subUpperbound
        Go to the next component
    end if
    //Run brute-force
    if number of vertices <= 20 then
        chromaticNumber of component = BruteForce(component)
    end if
end for
```

```
38: newUpperbound = max(subUpperbounds)
39: if newUpperbound < upperbound then
40:    //Update upper bound
41:    upperbound = newUpperbound
42: end if
43: lowerbound = max(subLowerbounds)
44: if has found all chromatic numbers of components then
45:    chromatic number = max(chromatic numbers of components)
46: else
47:    lowerbound = max(chromatic numbers of components)
48: end if
49: geneticAlgorithm(graph)
```

If the chromatic number of a component still cannot be calculated then a brute-force algorithm is used (line 34 - 36). However, only components with number of vertices below 20 are proccessed with the brute-force algorithm, since the brute-force algorithm normally takes longer than 2 minutes to execute on bigger graphs. Without the time-restriction of 2 minutes, 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 the upper bound of the original graph (line 38). 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 43).
If the chromatic numbers of all components have been found, then the chromatic number of the original graph is the biggest chromatic number among the components (line 44-45). If it is not the case, then the biggest chromatic number found on the components is another lower bound for the original graph (line 47).
Finally, in line 49, 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). 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.

The algorithm is based on breadth-first search. A unchecked-list stores the vertices whose neighbors are not yet added to the current expanding 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
```

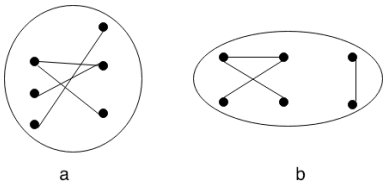

Figure 2.1: A graph (a) before decomposed and (b) after decomposed

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 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 for 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 (Singhal, n.d.) provides an efficient way of coloring a graph. However, it does not guarantee that the coloring is optimal. Therefore, it can be used to calculate a upper bound. Algorithm 3 describes this method.

First, the vertices are sorted based on their constraints (line 1). The constraint

Algorithm 3 Greedy algorithm for upper bound

Require: a graph

```
1:  Sort vertices in non-increasing order of constraints (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
```

of a vertex is the number of other vertices connected to that vertex. Bubble

sort (*Bubble sort*, n.d.) is used in this step. The vertex with more neighbors will be colored first.
A list is used to store available colors. 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.

2.4 Greedy algorithm for lower bound

Greedy algorithm for lower bound finds several cliques in a graph and returns the size of the biggest clique found. However, it does not guarantee to find the maximum clique.
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.

2.5 Special cases

2.5.1 Bipartite

The formal definition of a bipartite graph is:
"A bipartite graph, also called a bigraph, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent." (Weisstein, n.d.)
The chromatic number of a bipartite graph is 2. Algorithm 4 (Barnwal, n.d.) describes the steps to test weather a graph is bipartite, using breadth-first search.

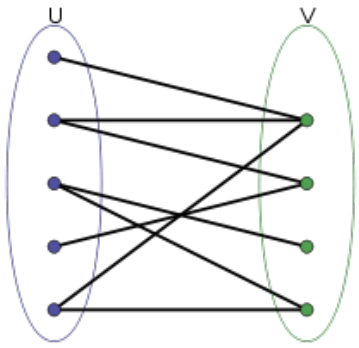

Figure 2.2: An example of bipartite graphs. This figure is from (*Bipartite graph*, n.d.)

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

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
```

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.

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.

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.

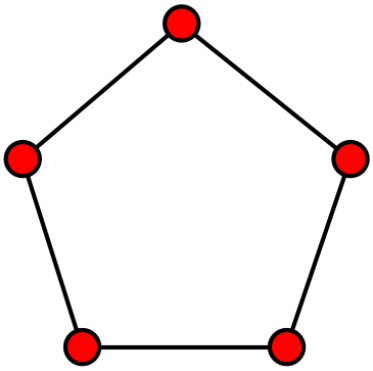

Figure 2.3: An example of odd cycle graphs. This figure is from (*Cycle graph C5*, n.d.)

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.

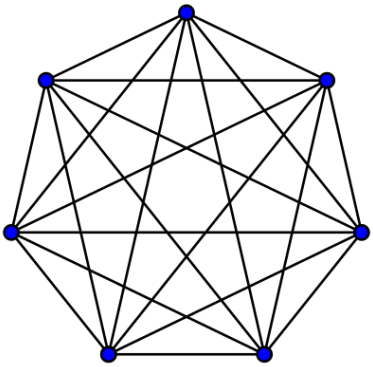

Figure 2.4: An example of complete graphs. This figure is from (*Complete graph*, n.d.)

2.6 Brute-force algorithm

The brute-force algorithm simply generates every possible coloring and checks if it's 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, that iterates through the array representing nodes' colors and searches for a conflict (two adjacent/connected nodes having assigned the same color). If found, false is being returned instantly, 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 nodes, 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 effectivness 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 experiment is set up to run on the given 20 graphs from phase 3 of the project. Each graph is exploited to see its 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

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 4.1: Results on the given 20 graphs from phase 3. The column "Gap" represents the differences 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	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, 25, 29, 26, 25	0	10	1
19	143	1	143	0	11	0
20	387	1	387	0	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

Chapter 5

Discussion

Chapter 6

Conclusion

References

Barnwal, A. (n.d.). *Check whether a given graph is bipartite or not.*

Bipartite graph. (n.d.). https://en.wikipedia.org/wiki/Bipartite_graph. (Accessed: 2019-01-16)

Bubble sort. (n.d.). https://en.wikipedia.org/wiki/Bubble_sort. (Accessed: 2019-01-15)

Complete graph. (n.d.). https://en.wikipedia.org/wiki/Complete_graph. (Accessed: 2019-01-16)

Cycle graph c5. (n.d.). https://commons.wikimedia.org/wiki/File:Cycle_graph_C5.png#filelinks. (Accessed: 2019-01-16)

Singhal, A. (n.d.). *Graph coloring algorithm — how to find chromatic number.*

Weisstein, E. W. (n.d.). *Bipartite graph.*