Maastricht University
Department of Knowledge Engineering

Bouillonstraat 8-10, Maastricht, the Netherlands

Project 1.1.3 Report

A Multi-Algorithm Approach to Graph Coloring

Authors:
Jurriaan Berger (i6142402)
Savvas Emexides (i6128159)
Andrew Gold (i6126154)
Jonas Molz (i6142067)
Nikolas Waniek (i6141814)
Laurin Zecherle (i6147432)

25 Jan. 2017

Supervisors:
Jan Paredis
Steven Kelk
Evgueni Smirnov
Kurt Driessens
Pietro Bonizzi

Preface

This report is a brief explanation of an approach to traversing and coloring graphs. The Graph Coloring Problem (GCP) is a well-known and thoroughly researched topic in the field of mathematics and computer science and its roots can be traced back to mathematician Francis Guthrie, who proposed the Four Color Theorem in 1852. Efficient coloring algorithms play crucial roles in many engineering fields. The practical applications of these concepts include schedule optimization in operations research, directing automobile and airline traffic, and many other situations where optimal patterns within a system need to be recognized efficiently and accurately.

Abstract

Herein is described a comprehensive approach to finding the chromatic number of a specified graph, using several methods to determine best lower and upper bounds and, if possible, the exact chromatic number. Special types of graphs such as bipartite and wheel graphs are initially tested using inexpensive algorithms, and if it is indeterminable that a graph is of a special structure, more computationally expensive methods are used. Well-known graph-traversing algorithms DSATUR, Welsh-Powell Greedy, and Maximum Clique are used concurrently to create accurate upper and lower bounds. The principle methodology of these algorithms is explored, and the motivations behind their results are given. Additionally, the results from a systematic case study of multiple sets of graphs using the above methods are given, and the efficacy and accuracy of each algorithm is motivated.

Contents

1 Lists

1.1 Tables

1.2 Figures

1.3 Appendices

1.4 Abbreviations

- GCP: Graph Coloring Problem
- V: set of vertices
- v: vertex within a given set of vertices V
- E: set of edges
- e: edge within a given set of edges E
- *N(v)*: the set of all neighbours of vertex v
- G: Graph
- Δ: The degree (A.K.A. "valence") of a graph or subgraph (denoted ΔG) is the number of connected vertices to a given vertex v
- $\Delta'(v)$: The vertex v with the highest degree within a given graph or subgraph.
- G' *(prime)*: denotes an induced subgraph within a given graph G
- n: the number of edges within a given graph or subgraph
- k: the number of colors within a (proper) coloring of graph G
- LB: The lower bound estimate of a given graph or subgraph
- UB: The upper bound estimate of a given graph or subgraph
- $\chi(G)$: The chromatic number of a given graph G
- $\alpha(G)$: The independence number of a given graph G
- BT: Backtracking algorithm
- MC: Maximum Clique algorithm
- WP: Welsh-Powell algorithm
- $DSATUR$: Degree of Saturation algorithm
- RLF: Recursive: Large First algorithm

2 Introduction

The *graph coloring problem* (GCP) is informally described as: Given an undirected graph *G(V,E)* such that V is the number of vertices and E is the number of edges connecting the vertices in G, assign a color to vertex $v \in V$ such that no two vertices connected by an edge e have the same color, using as few colors as possible.

The goal of this project is to determine the most viable methods of traversing and coloring a graph, assuming that all graphs are undirected. Special cases will arise, for example bipartite, wheel, and planar graphs. The algorithms employed in this project must be capable of identifying these special cases, as well as being able to traverse and color more traditional graph formats.

Coloring a graph optimally requires a few things to be considered. First, the *degree* of a vertex v describes the number of adjacent vertices. The degree of graph G is denoted by ΔG, which is defined as the maximum degree of its vertices V. A *clique* is a subset of vertices $C \in V$ such that every two distinct vertices are connected. The *maximum clique* $\max\{C\}$ is the largest set of connected vertices in V. G' denotes an *induced subgraph* of G, where induced subgraph is defined as any given subset of vertices and their connecting edges within a graph G. A *walk* of a graph is an ordered, alternating traversal of vertices and edges beginning and ending with vertices, such that each edge is incident with the vertex immediately preceding and following said vertex.

2.1 Project Phase 3

Phase 3 of Project 1.1 focuses on a given set of 20 new undirected graphs, which are more computationally challenging. Certain graphs may be "specially" structured in a way that will need to be identified and tested for. Exploiting these special structures is essential as the computation time required to detect these graph structures is trivial, though will require special attention in identifying.

Furthermore, an imposed run-time limit of 180 seconds was given, meaning that the program needed to be efficient even if this came at the cost of accuracy. Given an unlimited time frame, the obvious choice would be coloring the graph via brute force, but that proves unfeasible within the given constraints. The main focus was to create reliable lower and upper bounds, and to continuously improve upon these bounds considering that the difference between these bounds would be given as a penalty.

2.2 Problem Description

The main challenge of this phase was designing a program capable of both identifying special structures of graphs and their respective chromatic numbers, as well as computing accurate lower and upper bounds for all other graphs. Therefore, further research into the types of special graph structures was necessary. Identifying special graphs is relatively trivial computationally speaking, though the implementation of more complex heuristics in this project required the identification (or inability to identify) specific graph structures to be taken into account.

2.3 Report Structure

This report will first expand upon the concepts of special graph structures in chapter 3, giving definitions and brief motivations behind the complexity of each. Then, the structure of the program and its functions will be described. In chapter 4 the individual algorithms implemented will be described in detail, including any unique modifications made specific to this project. In chapter 5, details regarding the intricacies of the GCP are explored, as well as some of their implications on the efficiency of obtaining a chromatic number. A case study on the 20 graphs given in this project phase and 50 DIMACS benchmark graphs will be explored in chapter 6, including the exact results and relevant statistics for each algorithm. In this case study, the results will be shown for each algorithm along with their runtime for comparison. Graphs that were of a special structure, as well as those that proved more difficult to color, will be examined in depth following the results. Finally, the conclusion in chapter 7.

3 Project Overview

3.1 Types of Graphs

Here a set of typecal graphs is listed, as well has a briefly description of how they can be detected.

- **Complete Graphs:** Graphs whose vertices are all connected to each other. The number of edges in graph G should be equal to: $v(v-1)/2$. [1]
- **Bipartite:** Graphs whose vertices can be divided into two disjoint sets, such that there is no edge between any two members of the same set. Regardless of the number of edges between vertices, these graphs are always two-colorable.
- **Trees:** Undirected graphs in which any two vertices are connected by exactly one path. These types of graphs are always two-colorable.
- **Wheel Graphs:** Graphs formed by connecting a single vertex v_1 to all vertices v in a cycle. The number of edges $E = 2(v-1)$ and all vertices except one must have $\Delta = 2$, the other vertex must have $\Delta = v-1$.
- **Planar Graphs:** Graphs whose edges never cross one another when drawn on a plane, i.e. a graph that can be drawn within a given plane without any edges crossing each other. By assumption of the four color theorem, the chromatic number of a planar graph is at most 4 [7]. However, algorithms that detect planar graphs in linear time are "difficult to implement and understand. A simple and efficient algorithm for testing the planarity of a graph... would be a significant contribution." [6], and therefore such an algorithm is neither discussed in this report nor implemented in the program itself.
- **Undirected Graphs:** Graphs whose vertices are connected by edges that are *bidirectional*; that is, that the edges have no given direction in their connection to the vertices.

3.2 Structure of Program

(see Appendix)

The initial step is to check the graph for trivial cases, i.e. complete graphs and wheel graphs. If identified, the chromatic number is immediately known. Otherwise, the next step is to check if the graph is bipartite, which gives the chromatic number. If the chromatic is still not found by these trivial searches, both the Maximum Clique and heuristic algorithms begin to determine lower and upper bounds, updating themselves as they determine more accurate results. By comparing the results for identical lower and upper bounds, either the chromatic number or the range in which it falls is given. If at this time no solution has yet been found, the graph is then analyzed to determine if it can be split into smaller subgraphs. If this is possible, the coloring algorithms are performed on the individual parts for increased accuracy and efficiency. Finally, a brute-force backtracking algorithm will begin its coloring sequence regardless of projected run time. The coloring process times out after 180 seconds.

4 Algorithms in Depth

The most straightforward algorithmic approach towards serially coloring a graph is to color each vertex sequentially, assigning a new color to each vertex as it is presented. This will result in a coloring equal to the number of vertices, and is hardly the most accurate method. Improving upon this method is one of the most simple heuristics, the greedy search.

Let V be the set of all vertices to be colored within graph G, and let K=$\{v_1, v_2, \ldots v_n\}$ such that K is an arbitrary ordering of vertices within V [2].

Greedy search:

```
for $v$ : $v_1$ to $v_n$
color $v$ with the smallest possible color available not used by the neighbors of $v_i$.
endfor
```

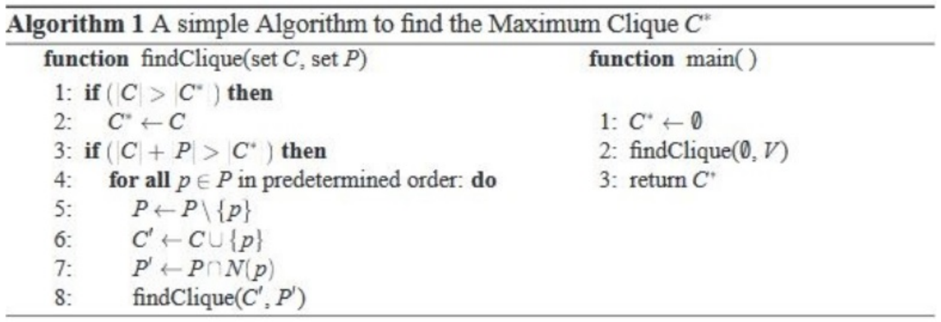
This simple approach analyzes each step sequentially without regard for future impact, and is generally unfeasible for use with complex graphs. On a small scale, however, the greedy search method can be refined, and several more complicated algorithms utilize the greedy structure by constraining it within certain heuristic boundaries. In these more complex algorithms, both the accuracy of the coloring and the computational expense can be dramatically improved upon.

4.1 Two-Colorability Algorithm

For checking whether a graph is bipartite a simple breadth-first search is performed. While traversing the graph each vertex that is encountered is marked with one of two colors, alternatively. This will always produce a proper 2-coloring of the graph, provided it is bipartite.

4.2 Maximum Clique

Finding the Maximum Clique in a graph is one of the most well-known NP-Complete problems in the field of graph theory. For the scale of complexity this project is designed to tackle, a sensible choice was to use an exact, recursive algorithm with pruning capabilities in order to identify the largest clique in a given graph. The choice of implementation was a simple algorithm to find the Maximum Clique.



Algorithm 1 A simple Algorithm to find the Maximum Clique C^*

```
function findClique(set $C$, set $P$)
  1: if ( $|C| > |C^*|$ ) then
  2:     $C^* \leftarrow C$
  3: if ( $|C| + |P| > |C^*|$ ) then
  4:     for all $p \in P$ in predetermined order: do
  5:         $P \leftarrow P \setminus \{p\}$
  6:         $C' \leftarrow C \cup \{p\}$
  7:         $P' \leftarrow P \cap N(p)$
  8:         findClique($C'$, $P'$)

function main( )
  1: $C^* \leftarrow \emptyset$
  2: findClique($\emptyset$, $V$)
  3: return $C^*$
```

Figure 1: Pseudocode for a simple algorithm to find the Maximum Clique C' in a graph, by Fahle T. [8].

Let C be an empty set acting as a container for the clique, and P, a set composed of all the candidates for the clique, initially $C = \emptyset$, and, consequently, $C = V$. The algorithm removes one vertex v from the candidate set, adds it to a new set C' and then runs recursively with the new set C' and an updated candidate set P', specifically the intersection of P and $N(v)$, the neighborhood of the newest member of the clique. It proceeds to do so for all the vertices in the initial set of candidates P, therefore searching through the set all possible maximum cliques. The Cliques are stored inside a global (w.r.t. the scope of the method), variable C^* , which represents the maximum clique.

There is also some pruning involved, namely, the algorithm only searches when the size of the currently explored clique, together with its candidates, is bigger than the largest clique already found.

4.3 Degree of Saturation (DSATUR)

The DSATUR algorithm uses a particular heuristic to change the ordering of nodes, updating this ordering each time a best candidate vertex has been selected and colored. The central concept is the degree of saturation, defined for vertex v as the number of vertices adjacent to v which have already been colored. The vertex with the highest saturation $\Delta' v$ is selected for coloring. Whenever a tie needs to be broken – for example in the beginning, when all vertices have saturation zero – priority is given to higher degree vertices.

The coloring process itself functions like it does for the Greedy algorithm – after the best candidate vertex is chosen, the list of available color classes is traversed in order, and the vertex is added to the lowest numbered color class that does not cause a conflict; if there is no such color class, a new color class is created. After the first vertex is colored, the heuristic rule is applied, and all vertices adjacent to the one just colored have their saturation increased by one.

Similarly to Welsh-Powell, the goal is to prioritize the vertices with the most constraints on them. However, where Welsh-Powell simply considers the degree of a vertex, DSATUR considers only that part of the degree that creates actual constraints at any given point in time in the first stage of its selection process, ignoring uncolored adjacent vertices unless a tie needs to be broken. This additional level of complexity makes the DSATUR algorithm marginally more computationally expensive than the Welsh-Powell. DSATUR is proven to be exact for a set of special graph topologies including bipartite graphs, wheel graphs, and cycle graphs [2]. Its worst-case time complexity is $\mathcal{O}(n^2)$.

4.4 Welsh-Powell

The Welsh-Powell algorithm is a variant of the greedy algorithm that utilizes a particular ordering of the vertices yet to be colored [5]. Let K=$\{v_1...v_n\}$ such that K is an ordered set of vertices arranged by degree, with v_1 being the vertex with the highest degree, descending towards the vertex of the lowest degree, v_n. Assign v_1 the first color, and walk through K. For every v_i the neighboring vertices are checked to see if any existing coloring conflicts with the current color in the queue. If no conflicts are found, v_i is colored accordingly and the walk through K continues. Once v_n is reached, if there remain any uncolored vertices within K, the next color in the queue is selected and K is traversed again. This continues until all vertices are colored.[5]

The reasoning behind this ordering is that vertices with a high degree can become subject to a larger number of constraints than low-degree vertices; by coloring the high-degree vertices first while they are still subject to few constraints, the algorithm reduces the likelihood of conflicts in later stages. Welsh-Powell is polynomially bounded and relatively inexpensive computationally, at a worst-case time complexity of $\mathcal{O}(n^2)$.

4.5 Recursive Large-First (RLF)

The RLF algorithm is different from DSATUR in that, like Welsh-Powell, it creates the coloring one color at a time rather than vertex by vertex. Structurally, it consists mainly of two while-loops.

Let K be an arbitrarily ordered set of vertices. While G contains uncolored vertices, create a new color class C. For each individual color class, the vertex with the highest degree $\Delta'(v)$ is added to C and all neighbors of $\Delta'(v)$ are added to a separate graph Y. Then both $\Delta'(v)$ and all corresponding neighbors are removed from G.

This ensures that all vertices left in G are still viable candidates to be added to the color class. While G contains uncolored vertices, a new vertex to be added to the color class is chosen;

from this second vertex onward, vertices are chosen in order of their degree in relation to already existing members of Y. As with Welsh-Powell, the objective is to prioritize vertices with a high number of constraints. The new vertex is added to C, and all its neighbors are added to Y; both are removed from G. This process continues until G is empty, there are no more possible candidates to be added to C, as every uncolored vertex is now adjacent to at least one vertex $v \in C$. All the vertices from Y are now returned to G, and the outer while-loop begins again, repeating the process for a new color class; this continues until there are no uncolored vertices left.

The worst-case time complexity for RLF is $\mathcal{O}(n^3)$, so it is considerably more computationally expensive than Welsh-Powell and DSATUR. However, RLF has been proven to produce better results than both Welsh-Powell and DSATUR. RLF is also proven to be exact for bipartite graphs, cycle graphs, and wheel graphs [3].

4.6 A Backtracking-based Exact Algorithm

This algorithm combines Greedy with Backtracking to find the a coloring with minimal possible colors and thus $\chi(G)$. It will be called Backtracking Greedy (BG) in the following. A Java implementation can be found in the appendix. As well as Greedy, BG takes in a vertex permutation to determine the order in which the vertices are colored.

It operates differently than Greedy in the sense that it keeps track of all available colors for a vertex, which are all already used colors which are feasible for a vertex, and a new color. For each of those possibilities, the vertex is colored and then for the next vertex in the permutation the same processed is applied. This results in a series of forwards and backwards steps, going forwards if a color for a vertex is picked, and going backwards to the last vertex having any remaining "untried" available color, whenever the last vertex of the permutation is reached or pruning can be applied. A variable *best*, initially being $best = n$, is used to store the minimum number of colors used in all colorings. It is updated whenever the last vertex of the permutation is reached and a lower result is obtained. This can be used for pruning by applying the backwards step, whenever the number of used colors reaches *best*.

On the termination of the algorithm, $best = \chi(G)$, as $\chi(G) = min\{k \in \mathbb{N}$: there is a proper k-coloring$\}$ and BG browses the whole space of all proper colorings, only excluding those who cannot result in a decrease of *best*.

5 Reducing Problem Size

As the correlation of all exact algorithms for finding $\chi(G)$ to the number of nodes is at least exponential, and ultimate goal of this report is to find the exact chromatic number for as many cases possible, it is of great interest to reduce n as much as possible without interfering with $\chi(G)$. Also depth-first search can be applied to compute connected components, which is done in our implementation.

5.1 Use of Articulation Points For Vertex Coloring

A connected Graph G is said to be 1-connected if the removal of one vertex, called an Articulation Point (AP), or cut vertex, induces a subgraph with p, $p \geq 2$, connected components, each denoted as K_i, with $1 \leq i \leq p$. In the following let $|V(G)| \geq 2$, implying $\chi(G) \geq 2$, as G is connected and let $a \in V(G)$ be an AP. Then each $K_i + a$ can be colored separately, using as many common color classes as possible. With permuting the color classes such that $c(a)$ has the same value in each $K_i + a$, they can be merged back into G while resulting in a proper coloring. Also, all proper colorings of G induce a coloring of all $K_i + a$. Thus, $UB(G) = max\{UB(K_i + a) : 1 \leq i \leq p\}$, where $UB(H)$ denotes a upper bound for $\chi(H)$ for some Graph H. Therefore, $\chi(G) = max\{\chi(K_i + a) : 1 \leq i \leq p\}$, meaning that computing the chromatic number of $K_i + a$ for some i results in a lower bound. This also implies that all $K_i + a$ whose upper bound, gained by running the heuristics, is less or equal then a lower bound on $\chi(G)$, can excluded from further examination as they cannot contribute in computing the chromatic number of G. The same can be done when running BG is applied to some component, exiting the search earlier when the value of *best* reaches some lower bound of $\chi(G)$.

5.2 Tarjan's algorithm

The APs can be computed using a algorithm done by Tarjan and Hopcroft [9]. It uses depth first search (DFS), assigning each $v \in V(G)$ the depth it was was discovered with, denoted $depth(v)$ and a *low point* (LP), denoted $lp(v)$, initially being $lp(v) = depth(v)$ and a $parent(v)$, being the vertex visited before. When reaching the end of the DFS, it traverses in reversed order over the DFS tree, recursively assigning each v a new LP, with $lp(v) = min\{lp(u) \in N(v) : u \neq parent(v)\}$, starting at the leaves. After arriving at the root node, the LP of each vertex represents the vertex with the lowest depth that can be reached via path, which does not include any ancestor nodes in the search tree. This represents a path which would persist even after removing any ancestor, which is not the one with depth equal to the lower point of the examined node. Removing a node which has a child with a lower point larger or equal than the depth of that node from the Graph would therefore result in multiple components, thus $v \in V(G)$ is AP $\Leftrightarrow \exists c \in V(G) : depth(v) \leq lp(c)$ [9,10]. This has been implemented in Java. Due to given time constraints, this algorithm couldn't be elaborated to compute all K_i, merged with their adjacent APs, but it would most likely improve the performance of the overall program, as some components would be small enough to run BG on. Also the heuristics seem to perform better on smaller graphs, and thus also on the components.

6 Results: A Case Study

6.1 DIMACS Benchmark Graphs

[4]

Table 1: Results from DIMACS benchmark graphs. "Time" is in milliseconds. A time limit of 180 seconds is imposed.

Graph	v	e	χ	**Density**	**Max Clique**		**Welsh-Powell**		**DSATUR**		**RLF**	
					LB	Time	UB	Time	UB	Time	UB	Time
anna	138	986	11	0.10430	11	8	11	0	11	15	11	16
david	87	812	11	0.21705	11	0	11	0	11	0	11	0
fpsol2.i.1	561	3258	65	0.09493	65	2004	65	47	65	219	65	109
fpsol2.i.2	74	602	30	0.08564	30	4	31	0	30	47	30	47
fpsol2.i.3	80	508	30	0.09642	30	0	31	0	30	31	30	31
games120	496	11654	9	0.17871	9	0	9	0	9	0	9	0
homer	451	8691	13	0.02074	13	0	14	15	13	31	13	47
huck	425	8688	11	0.22288	11	4	11	0	11	0	11	0
inithx.i.1	120	1276	54	0.05017	54	372	54	16	54	225	54	140
inithx.i.2	864	18707	31	0.06730	31	8	31	0	31	94	31	125
inithx.i.3	645	13979	31	0.07256	31	4	31	15	31	78	31	110
jean	621	13969	10	0.16075	10	0	10	0	10	0	10	0
le450_15a	450	8168	15	0.08085	15	4	22	0	19	32	17	46
le450_15b	450	8169	15	0.08086	15	4	22	0	18	32	16	47
le450_15c	450	16680	15	0.16510	15	16	30	16	27	46	23	94
le450_15d	450	16750	15	0.16580	15	16	30	16	26	47	23	94
le450_25a	450	8260	25	0.08176	25	4	28	0	25	47	25	31
le450_25b	450	8263	25	0.08179	25	4	28	15	25	47	25	42
le450_25c	450	17343	25	0.17167	25	20	36	0	31	47	29	94
le450_25d	450	17425	25	0.17248	25	16	36	16	29	47	28	78
le450_5a	450	5714	5	0.05656	5	4	13	0	12	31	7	63
le450_5b	450	5734	5	0.05675	5	4	14	0	11	31	7	47
le450_5c	450	9803	5	0.09703	5	4	16	16	12	32	5	78
le450_5d	450	9757	5	0.09658	5	4	18	0	13	31	6	109
miles1000	128	6432	42	0.79134	42	52	45	0	43	15	42	0
miles1500	128	10396	73	1.27904	73	388	76	0	73	0	73	0
miles250	128	774	8	0.09522	8	0	10	0	8	0	8	0
miles500	128	2340	20	0.28789	20	4	21	0	20	0	20	0
miles750	128	4226	31	0.51993	31	0	33	0	31	0	32	0
mulsol.i.1	197	3925	49	0.20330	49	4	49	0	49	16	49	16
mulsol.i.2	188	3885	31	0.22101	31	0	31	15	31	0	31	0
mulsol.i.3	184	3916	31	0.23259	31	0	31	0	31	0	31	0
mulsol.i.4	185	3946	31	0.23184	31	0	31	0	31	0	31	0
mulsol.i.5	186	3973	31	0.23092	31	0	31	0	31	16	31	15
myciel3	11	20	4	0.36364	2	0	4	0	4	0	4	0
myciel4	23	71	5	0.28063	2	0	5	0	5	0	5	0
myciel5	47	236	6	0.21831	2	0	6	0	7	0	6	0
myciel6	95	755	7	0.16909	2	0	7	0	7	0	7	0
myciel7	191	2360	8	0.13006	2	0	8	0	8	0	8	16
queen11_11	121	3960	11	0.54545	11	4	17	0	18	0	14	0
queen13_13	169	6656	13	0.46886	13	0	21	0	22	0	16	0
queen5_5	25	320	5	1.06667	5	0	8	0	7	0	5	0
queen6_6	36	580	7	0.92063	6	0	11	0	10	0	8	0
queen7_7	49	952	7	0.80952	7	0	10	0	12	0	9	0
queen8_12	96	2736	12	0.6	12	0	15	0	15	15	13	0
queen8_8	64	1456	9	0.72222	8	0	13	0	15	0	10	0
queen9_9	81	2112	10	0.65185	9	0	16	0	15	0	11	0
zeroin.i.1	211	4100	49	0.18506	49	8	50	0	49	0	49	0
zeroin.i.2	211	3541	30	0.15983	30	0	32	0	30	0	30	16
zeroin.i.3	206	3540	30	0.16765	30	0	32	0	30	16	30	0

6.2 DIMACS Results

In 37 out of the 50 DIMACS graphs, an exact chromatic number was found. Of these, the RLF algorithm was responsible for the most accurate boundaries, always obtaining the correct upper bound as the chromatic number in all cases except for one (*miles750*). DSATUR and Welsh-Powell operated more efficiently than RLF, though almost never more accurately. In evaluating the lower bound, the results demonstrate that the Maximum Clique algorithm performed extremely reliably such that $LB{=}\chi(G)$ in well over half of the cases.

6.3 Phase 3 Test Graphs

Table 2: Results from test graphs given as a component of project phase 3. Time is denoted in milliseconds. "-" denotes when a specific coloring method was not performed. Backtracking (BT) was performed only when the exact chromatic number was not found via any other methods. A time limit of 180 seconds was imposed. *t.out* denotes when a coloring algorithm times out. Backtrack greedy is run until the time limit is reached.

Graph	v	e	χ	$K{=}2$	**Max Clique**		**Welsh-Powell**		**DSATUR**		**RLF**		**BT**
				Time	LB	Time	UB	Time	UB	Time	UB	Time	$\chi(G)$
1	161	886	15	-	15	2	15	46	15	15	15	12	-
2	90	4004	89	-	89	1	89	47	89	26	89	2	-
3	738	738	2	6	2	13	3	61	2	161	2	161	-
4	214	1263	[6-7]	-	6	4	9	45	8	31	7	22	*t.out*
5	209	249	3	-	3	3	3	47	3	13	3	43	-
6	138	439	11	-	11	2	11	2	11	45	11	9	-
7	81	1056	9-12	-	9	3	15	46	16	11	12	3	*t.out*
8	101	191	8	-	8	1	8	46	8	7	8	5	8
9	520	262	2	1	2	8	2	49	2	60	2	90	-
10	383	2498	8	-	8	10	9	53	9	64	9	102	8
11	200	955	3-5	-	3	3	6	47	6	23	5	18	*t.out*
12	100	509	5	-	4	2	7	44	6	10	5	5	5
13	450	1022	3-3	-	3	8	5	51	4	51	4	56	*t.out*
14	901	1802	3-4	-	3	15	5	61	5	211	4	273	*t.out*
15	160	191	3	-	2	2	3	44	3	11	3	16	-
16	161	320	4	-	2	3	4	49	4	12	4	12	4
17	210	1673	5-7	-	5	6	10	47	10	32	7	21	*t.out*
18	125	1110	10	-	10	4	11	61	12	15	11	8	10
19	36	522	12	-	8	16	12	45	12	4	12	1	12
20	4000	1198926	2	75	2	1675	4	212		*t.out*		*t.out*	-

6.4 Phase 3 Test Graph Results

All in all, 14 chromatic numbers were identified for the 20 graphs from the phase 3 test graphs. Checking for bipartite graphs resulted in the chromatic number for three graphs. Six other chromatic numbers have been found by comparing the upper and lower bounds and checking for equivalence. After searching graphs for connected components, one more chromatic number has been found. For the remaining ten graphs of which the chromatic number has not yet been found, the Backtrack Greedy algorithm has been able to find the chromatic number for four of them. For the lower bound, MC was able to distinguish $LB{=}\chi(G)$ in 9 out of 20 graphs, and never had a margin of error > 3, except for graph 19 where the margin of error was 4.

6.5 Discussion

6.5.1 RLF analysis

The RLF algorithm was clearly the most accurate algorithm in determining a proper coloring for G, in both the DIMACS and Phase 3 graph sets. Unsurprisingly, due to its accuracy RLF is one of the most popular greedy heuristics for the GCP. It sequentially builds color classes on the basis of greedy choices. In particular the first vertex placed in a color class C is one with a maximum number of uncolored neighbors, and the next vertices placed in C are chosen so that they have as many uncolored neighbors which cannot be placed in C. These greedy choices can have a significant impact on the performance of the algorithm, which justifies the choice of alternative selection rules [1].

Compared to the other heuristic methods employed, RLF was able to match the upper bounds in 11 out of 20 cases from Phase 3 and improved upon these upper bounds in an additional 8 cases. Furthermore, the increased run-time due to higher time-complexity was trivial, as for every graph except for graph 20 the RLF was able to optimally color a graph in under 300 milliseconds. For the DIMACS graphs, the RLF was able to match the upper bounds given by the other heuristics for over half of the cases. The remaining upper bounds were further improved, demonstrating RLF's exceptional accuracy.

For graphs exceedingly large or dense, it can be assumed that the exponential complexity nature of RLF will result in slower run-times, but this was not shown in either the DIMACS or Phase 3 cases.

6.5.2 DSATUR and Welsh-Powell analysis

While the Welsh-Powell algorithm was outperformed by DSATUR and RLF, it is the most computationally inexpensive of the algorithms used. It has been proven to perform significantly better than regular Greedy coloring with random ordering. DSATUR was outperformed by RLF as well, but the fact that it has a lower worst-case time complexity of $\mathcal{O}(n^2)$, compared to $\mathcal{O}(n^3)$ for RLF, may make it the superior choice in some cases. Generating the order in which DSATUR traverses the graph - without regard for the actual coloring - is also a way of creating orderings for other algorithms, such as a backtracking Greedy implementation.

6.5.3 Maximum Clique analysis

The MC algorithm gives proper results s.t. it finds the maximum clique in every case. Due to the fact that graphs from the test set were not very dense, the candidate set shrinks fast enough with the size of the examined clique while running the MC. Therefore, finding the maximum clique is relatively inexpensive and fast.

However, some graphs do have higher chromatic numbers than their maximum clique size. A MC will never find these higher LB's. In order to find higher LB's other algorithms and approaches are necessary.

One option considered was to run the maximum clique algorithm on the complement graph in order to find the independence number α. This might have improved the LB, since $\chi(G) \geq v/\alpha(G)$ of graph G [1]. Unfortunately this approach takes too much computational time. Since the complement of a graph that is not dense, is very dense and the MC is not able to find solutions for these very dense graphs in a short amount of time.

No further approaches of increasing the LB have been considered, nor implemented.

6.5.4 Connected Components of a Graph

If the heuristic algorithms and the MC are incapable of determining $\chi(G)$, there is one more option before starting the BT algorithm; namely, searching for inner connected components and run the heuristic algorithms again. By searching for these connected components, it becomes possible to reduce the problem size and significantly reduce the complexity of the search. Smaller components are easier to traverse and color, and because these components are not connected to each other, independent colorings will not affect the chromatic number. Heuristics performed on connected components resulted in slightly more accurate lower boundaries, and in several cases closing the gap between LB and UB to determine the chromatic number without the need for performing Backtracking Greedy.

6.5.5 Backtracking Greedy analysis

The Backtracking Greedy algorithm is only employed if a discrepancy between the LB and UB exists. Backtracking is run for the duration until a chromatic number is found, or the time limit of 180 seconds is reached. For the Phase 3 test graphs, backtracking was performed on 12 graphs, and in 50% of the cases Backtracking is used, it finds the exact chromatic number. The other 50% of the graphs timed out. For these graphs, the search timed out due to graph complexity and the inability to break the graph down into smaller components within the given time frame.

The results shown above are from the BT that uses the vertex permutation from the heuristics. However when this permutation is not given to the BT it is not able to find proper colorings for any of the graphs. Because the BT is greedy-based, the performance is highly dependent on the permutation that is used. By having a permutation from the heuristic algorithms, more accurate pruning methods can be performed in the BT to reduce problem size and increase efficiency, though this led to improved results in only a few of the test graphs.

7 Conclusions

In this project, several methods for traversing and coloring graphs are explored. By using exact algorithms to determine specific graph structures combined with heuristic and exact algorithms for computing accurate lower and upper bounds, the chromatic numbers for a significant portion of given test graphs were obtained. Accurate lower and upper bounds were also obtained, and the margin for error remained slim throughout.

Due to the the NP-Complete nature of the GCP, creating unique algorithms from scratch was impossible within the given time frame. Therefore the goal was to research the most efficient algorithms already in existence and to implement them according to the project guidelines. Furthermore, by combining both special graph structure searches with modern heuristic coloring methods, each algorithm needed to be rigorously examined on a case-by-case basis to ensure that the actual chromatic number did not lie outside of the given bounds.

Testing on a set of 50 complex benchmark graphs given by DIMACS with known chromatic numbers showed that the margins for error were relatively minor. When the methods are tested on the graphs from the Phase 3 test set many chromatic numbers have been found. For the graphs were no chromatic number has been found, the margin between the LB and UB is small. These performances lead to the conclusion that the methods employed in this project are rigorous and effective.

References

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

[2] Segundo, Pablo S. (2012). *A new DSATUR-based Algorithm for exact vertex coloring.* Computers & OR, vol 39(7): 1724-1733

[3] Adegbindin, et al. (2015). *A new efficient RLF-like Algorithm for the Vertex Coloring Problem.* University Ecole Polytechnique de Montreal.

[4] 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.

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

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

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

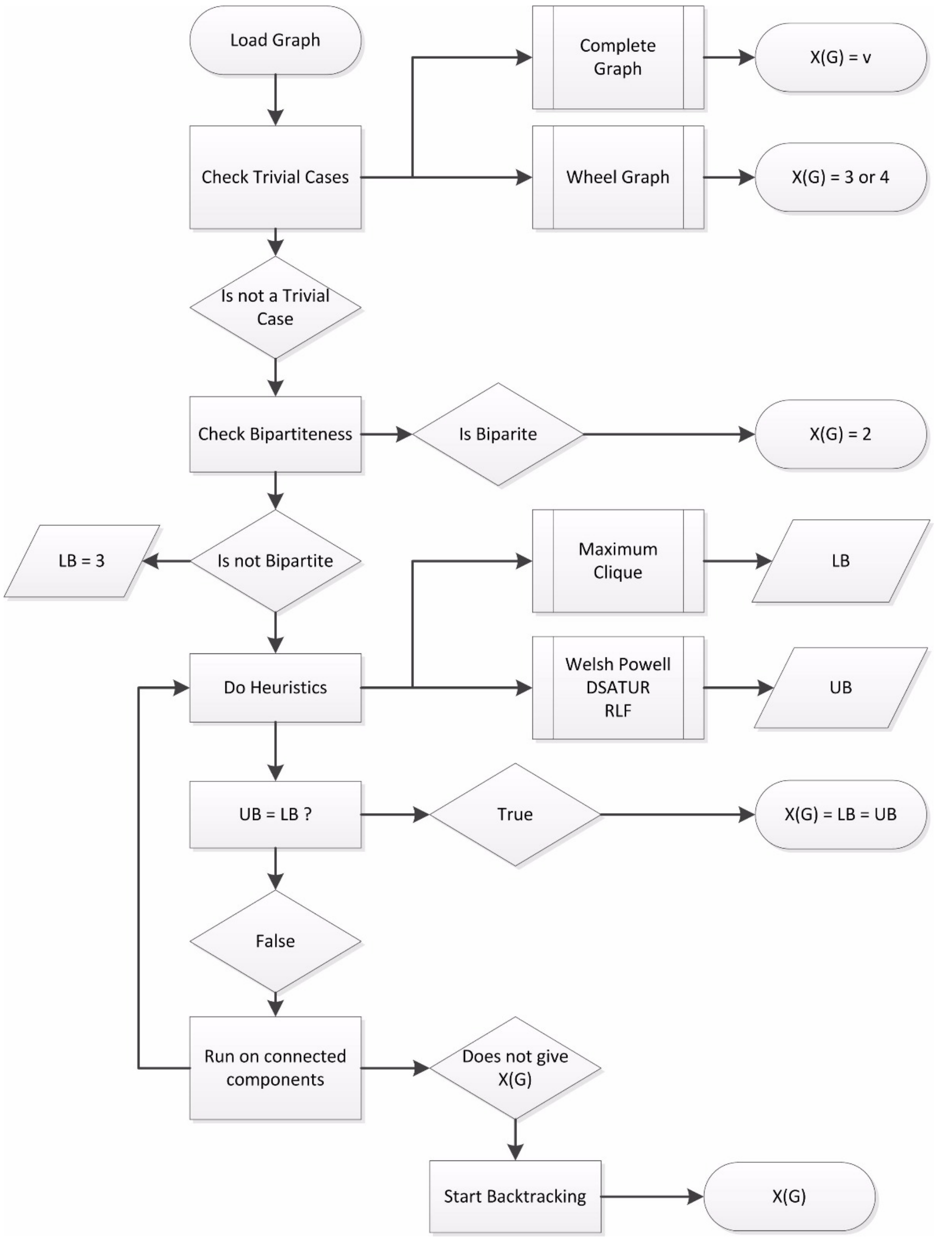
[8] Fahle, Torsten *Simple and Fast: Improving a Branch-And-Bound Algorithm for Maximum Clique.* (21-1-2017) http://www.dcs.gla.ac.uk/ pat/jchoco/clique/indSetMachrahanish/papers/fahle.pdf

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

[10] Anonymous. *Finding Articulation Points in a Graph.* (24-1-2017) http://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/

Appendix

Appendix: Flowchart of Program Structure


Load Graph
Complete Graph
X(G) = v
Check Trivial Cases
Wheel Graph
X(G) = 3 or 4
Is not a Trivial Case
Check Bipartiteness
Is Biparite
X(G) = 2
LB = 3
Is not Bipartite
Maximum Clique
LB
Do Heuristics
Welsh Powell DSATUR RLF
UB
UB = LB ?
True
X(G) = LB = UB
False
Run on connected components
Does not give X(G)
Start Backtracking
X(G)