BACHELOR'S THESIS

Degree in Mathematics

Title: An Experimental Study of the Minimum Linear Colouring Arrangement Problem.

Author: Montserrat Brufau Vidal

Advisor: Maria José Serna Iglesias

Department: Computer Science Department

Academic year: 2016-2017

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

An experimental study of the Minimum Linear Colouring Arrangement Problem

Author:
Montserrat Brufau Vidal

Advisor:
Maria José Serna Iglesias

Facultat de Matemàtiques i Estadística
Universitat Politècnica de Catalunya
26th June 2017

Als meus pares;
a la Maria pel seu suport durant el projecte;
al meu poble, Menàrguens.

Abstract

The Minimum Linear Colouring Arrangement problem (MinLCA) is a variation from the Minimum Linear Arrangement problem (MinLA) and the Colouring problem. The objective of the MinLA problem is finding the best way of labelling each vertex of a graph in such a manner that the sum of the induced distances between two adjacent vertices is the minimum. In our case, instead of labelling each vertex with a different integer, we group them with the condition that two adjacent vertices cannot be in the same group, or equivalently, by allowing the vertex labelling to be a proper colouring of the graph.
In this project, we undertake the task of broadening the previous studies for the MinLCA problem. The main goal is developing some exact algorithms based on backtracking and some heuristic algorithms based on a maximal independent set approach and testing them with different instances of graph families. As a secondary goal we are interested in providing theoretical results for particular graphs. The results will be made available in a simple, open-access benchmarking platform.

Keywords: MinLCA, Graph Colouring, Backtracking, Maximal Independent Set, Parameterised Problems, Benchmarking.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Problem formulation

Mathematical modelling is a largely studied field in mathematics. Many problems in real life can be translated into the language of mathematics. A significant group of these problems, especially in engineering, can be modelled as graph problems. Among these problems we find *graph layout problems*. They are a particular class of combinatorial optimisation problems whose goal is to find a labelling of the vertices of a graph with distinct consecutive integers, such that a certain objective cost is optimised.

One particular graph layout problem which has been largely studied by the computer science community is the *minimum linear arrangement problem* (MinLA). The objective of this problem is to find the best way of labelling the vertices of a graph in such a manner that the sum of the induced distances between two adjacent nodes is the minimum.
The problem we study is a variation of MinLA. Instead of labelling each node with a different integer, we group them under the condition that two adjacent vertices cannot be in the same group. More specifically, the layout will be a proper colouring of the graph. This problem is known as the *minimum linear colouring arrangement problem* (MinLCA). The problem can also be seen as a variation of the Colouring problem.

1.2 Interest of the problem

The interest in studying this problem comes from different points of view. First of all, the study of complexity for computational problems is a very important field in computer science and mathematics. As shown in later chapters, the problem we study is computationally hard, so theoretical and practical results on it can be very useful to the computer science community. More specifically, the *Algorithmics, Bioinformatics, Complexity and formal Methods* (ALBCOM) research group from the *Computer Science* department (CS) of UPC - BaracelonaTech is interested in this problem. It is a generalisation of the *minimum linear arrangement problem*, which has been thoroughly studied by Jordi Petit among others, see [3] and [9] for a full list of results and references for this and other graph layout problems.

Moreover, this problem is highly related to two other well known graph problems, *graph colouring* and *graph homomorphism*. If we look at it as a special case of the latter, it is possible to find a real world application related to computer science. We see the different nodes of the graph as different tasks of a computer programme and the edges as incompatibility rules between them (for example, two tasks that need to be executed on different computers but that share some information). Considering that the computers are sequentially numbered and physically distributed in the order of the sequence, the layout would be the assignment of the tasks to the different computers which minimises the physical distances between incompatible tasks sharing information. With this considered, the model might be a way to minimise the time and energy spent in information exchange.

1.3 State of the art

Even though the minimum linear arrangement problem has been largely studied for many years (see [3] and [9] for a complete list of results and references for this and other layout problems), our problem has only been studied by I. Sánchez in his thesis, see [14].
In said thesis, he developed some algorithms based upon those used for the MinLA. The difference between MinLA and MinLCA is in the solution: in the original problem it has to be a bijective mapping between the set of vertices and a set of integers, but for our problem we only need to have a proper colouring. For this reason, some of the algorithms developed for MinLA can give ideas for developing the algorithms for MinLCA, but due to their differences, the algorithms need to be different.

In his project, I. Sánchez started presenting the problem, establishing the problem complexity and giving some theoretical results. After that he developed three types of algorithms: exact, greedy and local search algorithms.
For the exact algorithms he developed an integer linear programming algorithm, and it was concluded that the algorithm took too much time to execute and it did not get to an optimal solution except for very few cases.
The greedy algorithms are based on a breadth-first search. Two different approaches were considered. The first one uses a nearest colour approach. When the algorithm has to assign a colour to a vertex v, it checks the colour of its neighbours and increases the preference of the one colour before and the one colour after the colour of each neighbour. The second greedy algorithm uses a least cost approach. In this case, the colour assigned to a vertex by the algorithm is the one with the least increment of the cost at each step.
Finally, for the local search algorithms the simulated annealing algorithm is used. This algorithm works by making small modifications to an initial solution with the aim to get a better solution at each step. For this algorithm two different approaches were considered. First, an approach based on the greedy recolourings was taken into account, for which the two greedy algorithms described above are used to colour the vertices. Secondly, an approach based on changing the colours was used. In this case, instead of reordering the vertices and recolouring them, we change the colour of the vertices and see what happens.

In order to test the algorithms, different graph instances were tested. These graphs include the instances used for the MinLA (see Table 4.1), binomial random graphs, graphs with cliques, random geometric graphs and outerplanar graphs.

1.4 Goals and scope of the project

The goal of this project is to continue the study of the *minimum linear colouring arrangement problem*, started by Isaac Sánchez in his bachelor's thesis (See [14]).
First of all we study the MinLCA problem and some of its parameterised versions. We start surveying its computational complexity and then some particular graph classes for which results for the problem are known. This includes bipartite, complete, planar and outerplanar graphs.

We complement those results with some new ones for the families of k-trees, complete k-partite graphs and bounded treewidth graphs. The second ones provide the tools to disprove a conjecture made in [14] about the number of colours used in the MinLCA problem and the latter allows us to give a result for the problem parameterised by the treewidth of the input graph.
Afterwards, we develop some exact and heuristic algorithms and evaluate their performance in different types of graphs which include the graphs used in the previous study of the MinLCA and also two new graph families, k-trees and almost 3-complete graphs.
For the exact algorithms we develop a backtracking approach based on a combinatorial search and test it in instances for big and small orders in order to determine for which size and type of graph it is feasible to use this approach. For the heuristic algorithms we develop a maximal independent set approach and compare the results with the heuristic algorithms studied in [14].
Finally, we plan to build an online benchmarking platform with all the obtained results, and add it to the benchmarking platform built by I. Sánchez in his thesis (See [15]). This way all the results will be available for further study of the problem.

Chapter 2

Preliminaries

In order to understand the problem presented, it is necessary to have a basic knowledge about graph theory and computational complexity. In this chapter we present a brief introduction to these fields, focusing on the concepts and properties that help make the next chapters self-contained and easier to follow.

2.1 Graphs

We start providing a brief introduction on graph theory. We give some basic definitions about this field, focusing on some specific computational problems which help in order to understand our problem. Through all of this section we follow the notations of [4].

2.1.1 Graphs and notation

We start by defining the concept of a graph, its components and the corresponding notation.

Definition 2.1.1. (Graph) A *simple undirected graph* is a pair (V, E) of sets such that V is the set of *vertices* (or nodes) of the graph and $E \subseteq \{\{u, v\}|u, v \in V, u \neq v\}$ is the set of *edges*. The vertex set of a graph is referred to as $V(G)$ and the edge set as $E(G)$.
The *order* of a graph is its number of nodes ($|V(E)|$) and the *size* is the number of edges ($|E(G)|$). Usually we set $n = |V(G)|$ and $m = |E(G)|$.
A vertex v is *incident* with an edge e if $v \in e$; then e *is an edge at* v. Two vertices incident with an edge are called its *endvertices* or ends, and an edge *joins* its ends. If u, v are the endvertices of an edge, we can write the edge as uv. Two vertices u, v of G are *adjacent*, or *neighbours*, if uv is an edge of G. We will denote two adjacent vertices as $u \sim v$. We are considering simple graphs, so there will be no loops (edges joining a vertex with itself) nor multiple edges (more than one edge joining the same vertices).
We say that $G' = (V', E')$ is a *subgraph* of G when $V' \subseteq V$ and $E' \subseteq E$. If $G' \subseteq G$ and G' contains all the edges $xy \in E$ with $x, y \in V'$ then G' is and *induced subgraph* of G. We use the notation $G[V']$ to represent the subgraph induced by $V' \subseteq V$.

Once we have these basic definitions in mind, we state the relationship between neighbours in a graph.

Definition 2.1.2. (Neighbourhood and Degree) Let $G = (V, E)$ be a graph. The set of *neighbours* of a vertex v in G is denoted by $\Gamma_G(v) = \Gamma(v)$.
The *degree* $deg_g(v) = deg(v)$ of a vertex v is the number of edges at v. This is equal to the number of neighbours of v.

2.1.2 Graph types

After this brief introduction we give a taste of different types of graph which are going to be mentioned in the following chapters.

We start with two basic subgraphs on every graph, the paths and cycles.

Definition 2.1.3. (Paths and Cycles) A *path* on n vertices is a graph (P_n) of the form $V(P) = \{x_1, ..., x_n\}$ and $E(P) = \{x_1x_2, ..., x_{n-1}x_n\}$. Such a path has $n-1$ edges and thus has length $n-1$.
A *cycle* on n vertices is a graph (C_n) of the form $V(P) = \{x_1, ..., x_n\}$ and
$E(P) = \{x_1x_2, ..., x_{n-1}x_n, x_nx_1\}$.

Next, we are going to introduce different graphs for which we know particular results for our problem.
First, we define a graph where all its nodes are connected. We can see that in this case the degree of all the nodes is equal to $|V(E)| - 1$.

Definition 2.1.4. (Complete Graph) A *complete* graph $K_n = (V, E)$ is a graph where $|V| = n$ and $E = \{uv|u, v \in V, u \neq v\}$.

Following the definition of a complete graph, we define bipartite graphs which, as we see in the next chapters, also has a closed result for our problem.

Definition 2.1.5. (Bipartite graph) A graph $G = (V, E)$ is bipartite if V can be divided into two disjoint sets A, B, i.e., $V = A \cup B$ and $A \cap B = \varnothing$, such that all edges have one endvertex in A and the other in B.
A *complete bipartite* graph is a bipartite graph where every edge from a set is connected with all the edges of the other set.

For the following graphs we do not know closed results for our problems, but we can deduce simple upper bounds for them. We have planar and outerplanar graphs and k-trees.

Definition 2.1.6. (Planar graphs) A graph $G = (V, E)$ is planar if it can be drawn in the plane and have its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other.

Definition 2.1.7. (Outerplanar graphs) A graph $G = (V, E)$ is outerplanar if the vertices can be drawn in a cycle so that the edges not in the cycle drawn as chords result in a planar graph.

Definition 2.1.8. (k-tree) A k-tree is a graph formed by starting with a complete graph of size k and repeatedly adding nodes in such a way that each node has exactly $k-1$ neighbours, so the k nodes form a clique.

2.1.3 Problems on graphs

Now we state different computational problems which are directly related to our problem. We explore the different statements for the property or parameters defining the problem, its decisional version and its parameterised version if it exists.

We start with the GRAPH_HOMOMORPHISM problem. Before stating the problem, we need to define the concept of graph homomorphism.

Definition 2.1.9. (Graph homomorphism) Let $G = (V, E)$ and $H = (V', E')$ be two graphs. Then the application $\varphi : V \to V'$ is an *homomorphism* if, for any $uv \in E$ we have $\varphi(u)\varphi(v) \in E'$. If, in addition, we require φ is bijective and that when $uv \notin E$ then $\varphi(u)\varphi(v) \notin E$ we say that *varphi* is an *isomorphism*.

With this we can state the problem in question.

Problem 2.1.1. (Graph_Homomorphism) Given two graphs $G = (V, E)$ and $G' = (V', E')$ find (if it exists) an homomorphism φ between them.

The decisional version for this problem is as follows.

Problem 2.1.2. (Graph_Homomorphism-Dec) Given two graphs $G = (V, E)$ and $G' = (V', E')$ decide if there is a graph homomorphism between them.

This is a very well known problem. It is NP-Complete. The equivalent problem for Graph isomorphism, i.e., given two graphs decide if there is a graph isomorphism between them is not known to be solvable in Polynomial time nor to be NP-Complete. It is known that this problem is in the low hierarchy of the NP-class and therefore if it is NP-Complete then the Polynomial Hierarchy (PH) collapses into its second level. (See [16]).

Next we define the concept of linear layout, which is basic in order to define the MinLA problem.

Definition 2.1.10. (Linear Layout) Let $G = (V, E)$ and $n = |V|$. A *linear layout* of G is a bijective function $\varphi : V \to [n] = \{1, ..., n\}$.
We call $\Phi(G)$ the set of all layouts of a graph G.
Given a layout φ of G and an edge $uv \in E$ the *length* of uv on φ is

$$\lambda(uv, \varphi, G) = |\varphi(u) - \varphi(v)| \tag{2.1}$$

If we sum the length of all the edges, we get the cost of the linear arrangement,

$$LA(\varphi, G) = \sum_{uv \in E} \lambda(uv, \varphi, G) \tag{2.2}$$

Now we can define a problem largely studied and defined in [3], the Minimum Linear Arrangement problem. Our problem is a variation of this one.

Problem 2.1.3. (MinLA) Given a graph $G = (V, E)$, find a layout $\varphi^* \in \Phi(G)$ such that $LA(\varphi^*, G)$ is the minimum,

$$MinLA(G) = \min_{\varphi \in \Phi(G)} LA(\varphi, G) \tag{2.3}$$

Problem 2.1.4. (MinLA-Dec) Given a graph $G = (V, E)$ and an integer ℓ, decide if there exists a layout $\varphi^* \in \phi(G)$ such that $LA(\varphi^*, G) < \ell$.

This problem is NP-Complete. It has been studied in numerous occasions, for example [10], [11]. To make our Benchmarking for our problem we use the instances J. Petit used in [11].

In the definition of our problem, we have two problems which are highly related to ours, the MinLA and Graph_Colouring problems. We have already seen the MinLA, so now we give the basic concepts in order to define the Graph_Colouring problem. We give some basic definitions that lead us to the problem in question.

Definition 2.1.11. (Vertex colouring) Given $G = (V, E)$ and $\varphi : V \to [k]$ for some $k \in \mathbb{N}$. We say that φ is a *vertex colouring* if for any $uv \in E$ then $\varphi(u) \neq \varphi(v)$. If such colouring exists, we say that G is k-*colourable* and φ is a k-*colouring*.
Observe that a k-colouring is a homomorphism from G to K_n for $n = |V(G)|$.

Definition 2.1.12. (Colouring family) Given $G = (V, E)$ and $k \in \mathbb{N}$, the *colouring family* F_k of G is defined as the set of colourings $\varphi : V(G) \to [k]$. That is

$$F_k(G) = \{\varphi : V(G) \to [k] \mid uv \in E(G) \Rightarrow \varphi(u) \neq \varphi(v)\} \tag{2.4}$$

Then we can define the family of all colourings

$$F(G) = \bigcup_{k \in \mathbb{N}} F_k(G) = \{\varphi : V \to \mathbb{N} \mid uv \in E(G) \Rightarrow \varphi(u) \neq \varphi(v)\} \tag{2.5}$$

Note that by labelling each node with a different colour we get a valid colouring. Therefore $F_n(G)$ is non-empty thus $F(G) \neq \emptyset$. This property allows us to set the following definition.

Definition 2.1.13. (Chromatic number) Given $G = (V, E)$ and k the smallest integer such that G has a k-colouring, then k is the *chromatic number* of G. It is denoted as $\chi(G)$. A graph with $\chi(G) = k$ is called *k-chromatic*.

Definition 2.1.14. (Linear distance of a colouring) Given a graph G and a colouring $\varphi(G) \in F(G)$, we define the *linear distance* of φ as

$$L[\varphi](G) = \sum_{uv \in E} \lambda(uv, \varphi, G) = \sum_{uv \in E} |\varphi(u) - \varphi(v)| \tag{2.6}$$

Now we have the tools to define the *graph colouring* problem.

Problem 2.1.5. (Graph-Colouring) Given a graph G and an integer k, find a k-colouring $\varphi \in F_k(G)$.

Problem 2.1.6. (Graph-Colouring-Dec) Given a graph G and an integer k, decide if there exists a k-colouring $\varphi \in F_k(G)$.

Finally, we are going to look at the parameterised version of this problem.

Problem 2.1.7. (Graph-Colouring$_k$) Given a graph G, find a k-colouring $\varphi \in F_k(G)$.

Problem 2.1.8. (Graph-Colouring-Dec$_k$) Given a graph G, decide if there exists a k-colouring $\varphi \in F_k(G)$.

The decisional version of the problem is well known to be NP-Complete except for the cases where $k = 0, 1, 2$. In fact, it is NP-Hard to compute the chromatic number. The 3-colouring problem is NP-Complete on planar graphs. However, for $k > 3$ the k-colouring problem is known to be in P for planar graphs, since every planar graph has a 4-colouring(see [19]).
This problem is highly related to our problem, in fact, if we find a solution for our problem, it is also a proper colouring.

2.1.4 Data structures for graphs

In order to work with graphs in a computer program, we first need to decide how to represent them within the computer. As we can see in [17], there are two commonly used representations and the choice between them depends primarily upon whether the graph is dense or sparse together with other considerations about the operations to be performed.

First of all, we need to map the vertex names to integers between 1 and $|V|$, so we can quickly access information corresponding to each vertex using array indexing.

The most straightforward representation for graphs is the *adjacency matrix* representation. We have a $|V| \times |V|$ array of boolean values, with position $[x, y]$ set to true if there is an edge from vertex x to vertex y and false otherwise.
The adjacency matrix representation is satisfactory only if the graphs to be processed are dense, because the matrix requires $|V|^2$ bits of storage and $|V|^2$ steps just to initialize it. If the number of edges is proportional to $|V|^2$, then this is acceptable because $|V|^2$ steps are required to read the edges in any case.

Now let us look at a representation that is more suitable for sparse graphs. In the *adjacency-structure* representation all the vertices connected to each vertex are listed on an *adjacency list* for that vertex. We have $|V|$ empty lists, one for each node. To add an edge connecting x to y we add x to y's adjacency list and y to x's adjacency list.
For this representation, the order in which the edges appear in the input is quite important because it determines the order in which the vertices appear on the adjacency list. This order affects, in turn, the order in which edges are processed by algorithms.
Some simple operations are not supported by this representation. For example, to delete a node x, and all the edges corresponding to it, it is not sufficient to delete nodes from the adjacency list, each node on the adjacency list specifies another vertex whose adjacency list must be searched to delete a node corresponding to x.

The adjacency matrix representation is more suitable when the basic operations are queries about the existence or not of potential edges. The second is useful when we operate processing the neighbours of a node one after the other.

In our case, we will mostly be working with sparse graphs, so we pick the adjacency list representation as input to our algorithms.

2.2 Computational Complexity

In order to understand the inherent difficulty of our problem, we study its computational complexity. For this reason we give a basic introduction to this field. We follow the notation of [18].

First of all we need to start with the definition of a Turing machine. A Turing machine can do everything a real computer can do. Nonetheless, even a Turing machine cannot solve certain problems. These problems are beyond the theoretical limits of computation. Here we have the formal definition for a Turing machine.

Definition 2.2.1. (Turing machine) A *Turing machine* is a 7-tuple, $(\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where $\mathcal{Q}$, Σ, Γ are all finite sets and

1. $\mathcal{Q}$ is the set of states,
2. Σ is the input alphabet not containing the blank symbol $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : \mathcal{Q} \times \Gamma \rightarrow \mathcal{Q} \times \Gamma \times \{L, R\}$ is the transition function,

5. $q_0 \in \mathcal{Q}$ is the start state,
6. $q_{accept} \in \mathcal{Q}$ is the accept state, and
7. $q_{reject} \in \mathcal{Q}$ is the reject state, where $q_{reject} \neq q_{accept}$.

The *start configuration* of a Turing machine M on input w is the configuration $q_0 w$, which indicates that the machine is in the start state q_0 with its head at the leftmost position on the tape. In an *accepting configuration* the state of the configuration is q_{accept}. In a *rejecting configuration* the state of the configuration is q_{reject}.
M *accepts* input w if a sequence of configurations $C_1, C_2, ..., C_k$ exists, where:

1. C_1 is the start configuration of M on input w,
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration.

The collection of strings that M accepts is the *language of* M, denoted $L(M)$. We call a language *Turing-decidable* or simply *decidable* if some Turing machine decides it.

Even if a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory. Next, we introduce computational complexity theory, a theory which aims to understand the resources of the time and memory among others required for solving computational problems. We focus on a way of measuring the time used to solve a problem.

Definition 2.2.2. (Time complexity) Let M be a deterministic Turing machine that halts on all inputs. The *time complexity* of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Now we come to some important definitions in complexity theory.

Definition 2.2.3. (P) The class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine is called *P*.

The class P plays a central role in complexity theory and it is important because it is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and it roughly corresponds to the class of problems solvable on nowadays computers.

We can avoid brute-force search in many problems and obtain polynomial time solutions. However, attempts to avoid brute-force in certain problems haven't been successful, and polynomial time algorithms that solve them are not known to exist.

Even so, there are problems which are not known to have a polynomial time solution but, if given a solution, it is easy to verify if it is a correct one. Here follows a formal definition of these concepts.

Definition 2.2.4. (Verifier) A *verifier* for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}. \tag{2.7}$$

We measure the time of a verifier only in terms of the length of w, so a *polynomial time verifier* runs in polynomial time in the length of w. A language A is *polynomially verifiable* if it has a polynomial time verifier.

With this definition in mind, we can define the class NP.

Definition 2.2.5. (NP) The class of languages that have polynomial time verifiers is called *NP*.

The class NP is important because it contains many problems of practical interest. The term NP comes from *nondeterministic polynomial time* and is derived from an alternative characterisation by using nondeterministic polynomial time.
The power of polynomial verifiability seems to be much greater than that of polynomial decidability, but P and NP could be equal. We are unable to prove the existence of a single language in NP that is not in P. The question of whether P=NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics. If these classes were equal, any polynomially verifiable problem would be polynomially decidable.
In the early 1970s Stephen Cook and Leonid Levin discovered certain problems in NP whose individual complexity is related to that of the entire class. If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called *NP-complete.*

In order to give a formal definition of NP-complete problems, we introduce the concept of reducibility. When a problem A is efficiently reducible to a problem B, an efficient solution of B can be used to solve A efficiently.

Definition 2.2.6. (Polynomial time computable function) A function $f : \Sigma^* \to \Sigma^*$ is a *polynomial time computable function* if some polynomial time Turing Machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

Definition 2.2.7. (Polynomial time reducible language) Language A is polynomial time reducible to language B, written $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists, where for every w,

$$w \in A \Leftrightarrow f(w) \in B. \tag{2.8}$$

The function f is called *polynomial time reduction* of A to B.

If one language is polynomial time reducible to a language already known to have a polynomial time solution, we obtain a polynomial time solution to the original language, as in the following theorem.

Theorem 2.2.1. *If $A \leq_P B$ and $B \in P$, then $A \in P$.*

Proof. See [18], pg. 273. □

From all this we reach the definition of NP-Completeness.

Definition 2.2.8. (NP-Completeness) A language B is *NP-complete* if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

Theorem 2.2.2. *If B is NP-complete and $B \in P$, then $P = NP$.*

Proof. This theorem follows directly from the definition of polynomial time reducibility. □

Theorem 2.2.3. *If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.*

Proof. See [18], pg. 276. □

2.3 Fixed-Parameter Algorithms

In order to deal with computational intractability, several methods have been developed, for example approximation algorithms or heuristic methods. Parameterised complexity theory is another proposal in order to cope with computational intractability in some cases. We give a basic introduction to this field, focusing on the concepts and properties which we use later. We follow the notation of [8].
We start giving a formal definition of parameterised problems.

Definition 2.3.1. (Parameterised problems) Given an alphabet Σ to represent the inputs to decisional problems.
A *parameterisation* of Σ^* is a mapping $\kappa : \Sigma^* \to \mathbb{N}$ that can be computed in polynomial time.
A *parameterised problem* is a pair (L, κ) where $L \subseteq \Sigma^*$ and κ is a parameterisation of Σ^*.
Parameterised problems are decisional problems together with a parameterisation. A problem can be analysed under different parameterisations.

Many hard computational problems have the following general form: given an object x and a nonnegative integer k, does x have some property that depends on k? In parameterised complexity theory, k is called *parameter*. If we take $\kappa(x, k) = k$, then k is the *natural parameterisation* of the problem. In many application, the parameter k can be considered to be "small" in comparison with the size $|x|$ of the given object. Hence, it may be of high interest to ask whether these problems have deterministic algorithms that are exponential only in respect to k and polynomial with respect to $|x|$.

Parameterised problems which can be solved in polynomial time in respect to $|x|$ are called *Fixed Parameter Tractable* problems, and we say that they belong to *FPT*. For a more formal definition:

Definition 2.3.2. Given an alphabet Σ and a parameterisation κ, we say that $\mathcal{A}$ is an *FPT* algorithm with respect to κ if there is a computable function f and a polynomial function p such that for each $x \in \Sigma^*$, $\mathcal{A}$ on input x requires time $f(\kappa(x))p(|x|)$.
A parameterised problem (L, κ) *belongs to FPT* if there is an FPT-algorithm with respect to κ that decides L.

FPT contains all polynomial-time computable problems. For a more general class of problems we have the W-hierarchy. This collection of computational complexity classes is defined as follows:

Definition 2.3.3. (W-hierarchy) A parameterised problem is in the class $W[i]$, if every instance (x, k) can be transformed (in FPT-time) to a combinatorial circuit that has weft at most i, such that $(x, k) \in L$ if and only if there is a satisfying assignment to the inputs, which assigns 1 to at most k inputs.

Note that $\text{FPT} = W[0]$ and $W[i] \subseteq W[j]$ for all $i \leq j$. Many natural computational problems occupy the lower levels, in fact the minimum linear arrangement problem (MinLA) parameterised by the treewidth of the graph is in $W[1]$ (see [6]).

Now we explore a graph parameter that measures the closeness of a graph to a tree: treewidth. In order to define this parameter we need some concepts first.

Definition 2.3.4. (Tree decomposition) A *tree decomposition* of a graph G is a tuple (T, X) where T is a tree and $X = \{X_v \mid v \in V(T)\}$ is a set of subsets of $V(G)$ such that:

- For every $xy \in E(G)$, there is a $v \in V(T)$ with $\{x, y\} \subseteq X_v$.

- For every $x \in V(G)$, the subgraph of T induced by $X^{-1}(x) = \{v \in V(T) \mid x \in X_v\}$ is non-empty and connected.

To distinguish between vertices of G and T, the vertices of T are called *nodes*. The sets X_v are called the *bags* of the tree decomposition.

Definition 2.3.5. (Treewidth) The *width* of a tree decomposition (T, X) for G is defined as $\max_{v \in V(T)} |X_v| - 1$.
The *treewidth* ($tw(G)$) of a graph G is the minimum width over all tree decompositions of G.

Deciding if a graph has treewidth w is NP-complete, but computing a tree decomposition with width at most w (if it exists) takes $O(f(w)n)$ time.

We make T into a rooted tree by choosing a root $r \in V(T)$, and replacing edges by arcs in such a way that every node points to its parent. For $v \in V(T)$, $R_T(v)$ denotes the nodes in the subtree rooted at v (including v). For $v \in V(T)$, $V(v) = X(R_T(v))$, and $G(v) = G[V(v)]$. Thus, we have an induced subgraph associated to each node.

A tree decomposition with a particularly simple structure is given by the following definition. It is useful to have such tree decomposition when solving problems by dynamic programming on tree decompositions, as we see in Subsection 3.4.3.

Definition 2.3.6. (Nice tree decomposition) A rooted decomposition (T, X) is *nice* if for every $u \in V(T)$

- $|X_u| = 1$ (start node)
- u has one child v
  a) with $X_u \subseteq X_v$ and $|X_u| = |X_v| - 1$ (forget node)
  b) with $X_v \subseteq X_u$ and $|X_u| = |X_v| + 1$ (introduce node)
- u has two children v and w with $X_u = X_v = X_w$ (join node)

Lemma 2.3.1. *Computing a rooted nice tree decomposition with width at most w, given a tree decomposition of width at most w takes $O(wn)$ time.*

Every node of the nice tree decomposition has a graph associated to it. First of all, for the start node we have an isolated vertex. Secondly, the graph associated with the forget node is the same as the graph associated to its child without the edges connecting the vertex x, which is not in the node. For the introduce node we have the same graph associated to its child adding one vertex and its edges. Finally, for the join node, the two graphs associated to its children only share the vertices of the join node and there are not any edges between vertices outside of the node, so the graph associated to the join node is the graph resulting of joining the two subgraphs.

With this results we have the tools to prove that a solution of MinLCA$_k$ for bounded treewidth graphs can be computed in $O(f(k, w)n)$ time.

2.4 Experimental platform

To test our algorithms on the instances we have chosen and compare our results to those in [14] we make some experiments using the same computational platform.
Before showing the results for every algorithm, it is important to have in mind the implementation details and the technical description to the cluster that we use to execute our code.

2.4.1 Implementation details

All the algorithms in this project have been implemented using C++11 and an open-source graph library called LEMON (Library for Efficient Modelling and Optimization Network [5]) version 1.3.1, which is part of the COIN-OR initiative [2].
The code has been written using C++ templates and inheritance and it should work on any system with a C++ compiler. The plots for the results have been made with MATLAB R2016b.

2.4.2 The cluster at RDLab

To obtain the results for our algorithms we are using the cluster at RDLab [13]. When we talk about clusters in computer science, we refer to a set of connected computers through high speed networks which work together in problem solving.
The cluster at RDLab works with the Oracle Grid Engine, a queue system that allows for assigning dedicated resources to each task. It runs on a Ubuntu 12.04.2 LTS system with an x86_64 architecture with Xeon X5670 CPUs running at 2.93 GHz.
Unless stated otherwise, the executions have been made using a single thread and a maximum of 2GB of RAM.
Our code has been compiled using C++ compiler from the GNU Compiler Collection (GCC) with version 4.6.3 and the CMake build tool version 2.8.7.

Chapter 3

The Minimum Linear Colouring Arrangement problem

In this chapter we define and analyse basic properties of the Minimum Linear Colouring Arrangement problem (MinLCA). First of all we introduce the problem and its variations. Then, we look at the computational complexity of the problem and some upper and lower bounds. Finally, we present results for some particular classes of graphs studied in [14] and we give some results for the parameterised version of the problem.

3.1 Definition and examples

3.1.1 The Minimum Linear Colouring Arrangement problem

We begin with some simple definitions which will help us define the MinLCA problem.

Definition 3.1.1. Given a graph G, we define

$$LCA_k(G) = \begin{cases} \min_{\varphi \in F_k(G)} L[\varphi](G), & \text{if } F_k(G) \neq \emptyset \\ +\infty, & \text{if } F_k(G) = \emptyset \end{cases} \tag{3.1}$$

Definition 3.1.2. ($MinLCA(G)$) Given a graph G, we define

$$MinLCA(G) = \min_{\varphi \in F(G)} L[\varphi](G) = \min_{k \in \mathbb{N}} LCA_k(G) \tag{3.2}$$

With this two definitions, we can state the general formulation of our problem as follows.

Problem 3.1.1. (MinLCA) Given a graph G, find a colouring $\varphi \in F(G)$ so that $L[\varphi](G) = MinLCA(G)$.

In Figure 3.1 we can see the solution for the MinLCA problem for a simple graph.

We can also consider the decisional version of the MinLCA problem.

Problem 3.1.2. (MinLCA-Dec) Given a graph G and an integer L, decide if there exists a colouring $\varphi \in F(G)$ so that $L[\varphi](G) \leq L$.

Once we have this general definitions of our problems we state the parameterised formulation by the number of colours, for a fixed integer k.

Problem 3.1.3. (MinLCA$_k$) Given a graph G, find a colouring $\varphi \in F_k(G)$ so that $L[\varphi](G) = LCA_k(G)$.

Figure 3.1: Original graph and optimal layout for MinLCA with cost 4.

As done before, we now can consider the parameterised decisional version of the problem.

Problem 3.1.4. (MinLCA-Dec$_k$) Given a graph G and an integer L, decide if there exists a colouring $\varphi \in F_k(G)$ so that $L[\varphi](G) \leq L$.

3.2 Computational complexity

In the previous section we introduced the MinLCA problem. In order to make a deep analysis of our problem, we need to look at the complexity of the problem. First we see that the problem is NP-complete and afterwards we give some upper and lower bounds for the problem. These results were first presented in [14].

3.2.1 NP-Completeness

Before proving the NP-completeness of our problem we need to prove a basic lemma in which we give an upper bound for $L[\varphi](G)$.

Lemma 3.2.1. *Given $G = (V, E)$ a graph and $k \in \mathbb{Z}$ an integer. Then,*

$$L[\varphi](G) \leq |E|(k-1) \quad \forall \varphi \in F_k(G) \tag{3.3}$$

Proof. We have $\max_{uv \in E} |\varphi(u) - \varphi(v)| \leq k-1$, because $\max_{v \in V} \varphi(v) = k$ and $\min_{v \in V} \varphi(v) = 1$. Then,

$$L[\varphi](G) = \sum_{uv \in E} |\varphi(u) - \varphi(v)| \leq \sum_{uv \in E} (k-1) = m(k-1) = |E|(k-1) \tag{3.4}$$

□

Now we have all the tools to prove the NP-completeness of the MinLCA problem.

Theorem 3.2.2. *The decisional version of* MinLCA$_k$ *is NP-Complete for any $k > 2$ and P for $k \leq 2$.*

Proof. First of all we prove the basic cases. For $k = 1$, we only have a proper colouring for a graph with no edges. Then, given a graph $G = (V, E)$ and an integer L, there only exists a solution if $|E| = 0$ and $L \geq 0$.

For $k = 2$, given $G = (V, E)$ and an integer L, we only have a proper colouring if G is bipartite (see Lemma 3.3.2). For bipartite graphs we know that $MinLCA(G) = |E|$ (see Theorem 3.3.3). Then we only need to see if $|E| \leq L$. Therefore, for $k \leq 2$ MinLCA-Dec$_k$ is in P.

Now we prove the general case. First of all we see that MinLCA-Dec$_k$ is in NP. We have to find an algorithm to verify a solution in polynomial time. In Algorithm 3.1 we provide the code for a verifier for MinLCA-DEC$_k$ in polynomial time.
We have two loops, in line 3.1.3 we are doing a loop linear on $n = |V|$ and in line 3.1.8 we are doing a loop with at most $m = |E|$ steps. So then, the cost of our algorithm is $\mathcal{O}(n + m)$. If we consider that the costs of the sums and comparisons are not constant, then we only have to add a $log(n)$ factor to the cost function.
Now we need to prove that our algorithm is NP-Hard. To do so, we will reduce it from GRAPH-COLOURING-DEC$_k$. Suppose that $\mathcal{A}(G, k, L)$ decides MinLCA-Dec$_k$ and let $L = |E|(k-1)$. If G is k-colourable then $F_k(G) \neq \emptyset$. Because of Lemma 3.2.1 we have that $\mathcal{A}(G, k, L)$ returns true. If we know that it returns true for the values (G, k, L), then we know that there exists a $\varphi \in F_k(G)$ (whatever the value of L).
So then we have that MinLCA-Dec$_k$ is NP-Complete. □

Algorithm 3.1 Verifier for MinLCA-Dec$_k$

Input:
$G = (V, E)$, $V = \{1, ..., n\}$, $|E| = m$ presented in an adjacency list
$k, L \geq 0$ integers
φ possible colouring of the problem

Output:
$\varphi \in F_k(G)$ and $L[\varphi](G) \leq L$

```
 1: function MinLCA-verif(G, k, L, φ)
 2:     LCA ← 0
 3:     for v ∈ V do
 4:         if φ(v) > k then                          ▷ If true then φ ∉ F_k(G)
 5:             return false
 6:         end if
 7:     end for
 8:     for uv ∈ E do
 9:         if φ(u) = φ(v) then         ▷ If true then φ is not a proper colouring
10:             return false
11:         end if
12:         LCA ← LCA + |φ(u) − φ(v)|
13:     end for
14:     if LCA ≤ L then                           ▷ If true then φ is a solution
15:         return true
16:     else                                      ▷ Else φ is not a solution
17:         return false
18:     end if
19: end function
```

3.2.2 Upper and lower bounds

Once we have proved the NP-completeness of the problem, we look at some basic upper and lower bounds for the MinLCA.
It is important to notice that every linear arrangement is a linear colouring arrangement in particular. This means that all known upper bounds for the MinLA are also upper bounds (although rough) for the MinLCA.

Now we present some bounds which come directly from the definition of the problem. These bounds were first stated in [14].

Proposition 3.2.3. *(Lower bound for* MinLCA*). Given a graph $G = (V, E)$, the number of edges is a lower bound for* MinLCA*(G). If $|E| = m$, then*

$$m \leq MinLCA(G) \tag{3.5}$$

Proof. The graph has m edges, and each edge uv has a cost $|\varphi(u) - \varphi(v)| \geq 1$, because $\varphi(u) \neq \varphi(v)$. □

Proposition 3.2.4. *(Upper bound for* MinLCA*). Given a graph $G = (V, E)$, the chromatic number of G and the number of edges m give us the following upper bound:*

$$MinLCA(G) \leq (\chi(G) - 1)m \tag{3.6}$$

And equality holds only when $\chi(G) \leq 2$.

Proof. Let $k = \chi(G)$ and apply Lemma 3.2.1. Considering $m \geq 1$, there is always at least one edge with cost 1, therefore equality holds only when all edges have cost 1. □

3.3 Previous results for particular graphs

In the next two section we look at some particular graphs for which the MinLCA problem has a closed results or a more adjusted upper bound. This is the case of bipartite graphs, complete graphs, complete balanced k-partite graphs, planar graphs, outerplanar graphs and k-trees. First of all we state the results previously studied in [14] for binomial, complete, planar and outerplanar graphs.

3.3.1 Bipartite graphs

We defined the concept of bipartite graphs in Chapter 2, now we are going to look at some results which lead us to a closed result stated in [14] for the MinLCA.

Theorem 3.3.1. *A graph G is bipartite if and only if it contains no odd cycles as subgraphs.*

Proof. See [4], pg. 9. □

Lemma 3.3.2. *A graph G is bipartite if and only if $\chi(G) \leq 2$.*

Proof. We just need to see the disjoint sets A, B from the definition as colours 1 and 2 for φ and vice-versa. □

Now we can enunciate the closed result of MinLCA for bipartite graphs.

Theorem 3.3.3. *(*MinLCA *for bipartite graphs) A graph $G = (V, E)$, with $|E| = m$, is bipartite if and only if $MinLCA(G) = m$.*

Proof. $\Rightarrow$ We have $\chi(G) \leq 2$. We just need to apply Propositions 3.2.3 and 3.2.4.
$\Leftarrow$ We show that G needs to be 2-colourable.
Suppose $\chi(G) = k \geq 3$. Let $\varphi \in \mathcal{F}$ be a colouring of G such that $L[\varphi](G) = MinLCA(G) = m$. Let $v \in V$ such that $\varphi(v) = \max_{u \in V} \varphi(u) = r \geq k$, which exists because we need at least k

colours.
Given that $L[\varphi](G) = m$, all vertices adjacent to v must have colour $r-1$. But then, v could be coloured with colour $r-2$ which is a contradiction.
So $\chi(G) \leq 2$ and G is bipartite. □

As we can see, when we have a bipartite graph we have an equality in the lower bound for the MinLCA problem that we saw in Proposition 3.2.3.

In Figure 3.2 we have an example of a bipartite graph and its result for the MinLCA problem.

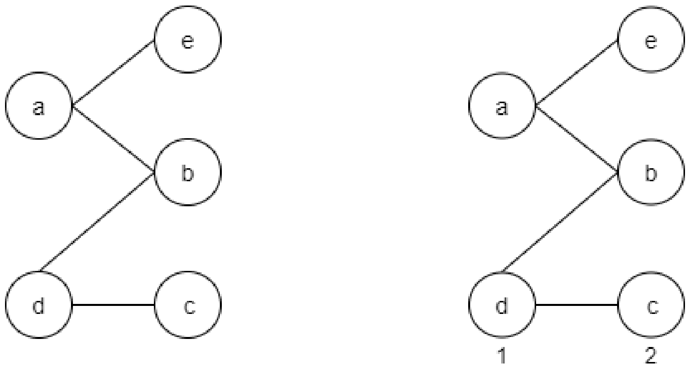

Figure 3.2: Bipartite graph and optimal layout for MinLCA with cost 4.

As a direct result from this theorem we have a result for the parameterised version of the problem.

Corollary 3.3.4. *(*MinLCA$_k$ *for bipartite graphs) Given $G = (V, E)$ a bipartite graph,*

$$MinLCA_k(G) = m \quad \forall k \geq 2 \tag{3.7}$$

3.3.2 Complete graphs

For complete graphs we also have a closed result for the MinLCA which was given in [14]. In this case we have a more elaborate proof than in the case of bipartite graphs.

Theorem 3.3.5. *(*MinLCA *for complete graphs) If $G = (V, E)$ is the complete graph K_n, then*

$$MinLCA(G) = \frac{n^3 - n}{6} \tag{3.8}$$

Proof. Since G is complete, all vertices are adjacent to each other and we need to use exactly n colours. Without loss of generality, suppose $V = [n]$ and that we choose the colouring $\varphi(v) = v$ for every $v \in V$. Then, we have

$$L[\varphi](G) = \sum_{uv \in E} |\varphi(u) - \varphi(v)| = \sum_{uv \in E} |u - v|$$

If we consider the edges uv with $u < v$, since uv and vu are the same edge,

$$L[\varphi](G) = \sum_{u=1}^{n-1} \sum_{v=u+1}^{n} (v - u)$$

changing v for $v - u$,

$$L[\varphi](G) = \sum_{u=1}^{n-1} \sum_{v=1}^{n-u} v$$

for which the inner sum can be seen as the cost of the edges incident to u which are on its right. Now change u for $n-u$,

$$L[\varphi](G) = \sum_{u=1}^{n-1}\sum_{v=1}^{u} v$$

The inner sum is an arithmetic series from 1 to $n-1$,

$$L[\varphi](G) = \sum_{u=1}^{n-1} \frac{u(u+1)}{2} = \frac{1}{2}\left(\sum_{u=1}^{n-1} u^2 + \sum_{u=1}^{n-1} u\right) = \frac{1}{2}\left(\sum_{u=1}^{n-1} u^2 + \frac{n(n-1)}{2}\right)$$

and using the identity for the series of the first $n-1$ squares,

$$L[\varphi](G) = \frac{1}{2}\left(\frac{n(n-1)(2n-1)}{6} + \frac{n(n-1)}{2}\right) = \frac{n(n-1)(n+1)}{6} = \frac{n^3-n}{6}$$

□

We can observe that the solution of a complete graph K_n is exactly the same for MinLCA and MinLA.

In figure 3.3 we have an example of a complete graph and its result for the MinLCA problem.

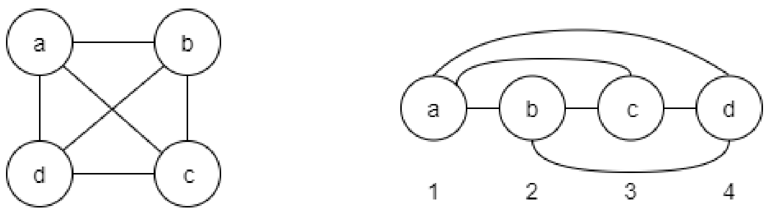

Figure 3.3: Complete graph and optimal layout for MinLCA with cost 10.

From this theorem we can state a result for the parameterised version of the MinLCA problem.

Corollary 3.3.6. *(MinLCA$_k$ for complete graphs) If $G = (V,E)$ is the complete graph K_n, then*

$$MinLCA_k(G) = \frac{n^3-n}{6} \quad \forall k \geq n \tag{3.9}$$

3.3.3 Planar graphs

In the case of planar graphs, we have a very important result which helps us give an upper bound for the MinLCA problem.This result is really difficult to prove. Indeed, all accepted proofs make use of computers and were not widely accepted at first. For more information on this topic, including the story of the theorem and a complete list of references, you can refer to [19].

Theorem 3.3.7. *(Four colour theorem) Every planar graph is 4-colourable.*

From this result, in [14] an upper bound for the MinLCA problem for planar graphs was deduced.

Corollary 3.3.8. *If $G = (V,E)$ is planar, with $|E| = m$, then $MinLCA(G) < 3m$*

3.3.4 Outerplanar graphs

In the case of outerplanar graphs, we also have an important result given in [14] from which we can extract an upper bound for the MinLCA.

Theorem 3.3.9. *If G is an outerplanar graph, then $\chi(G) \leq 3$.*

Corollary 3.3.10. *If $G = (V, E)$ is outerplanar, with $|E| = m$, then $MinLCA(G) < 2m$.*

3.4 New results for particular graphs

In this section we look at some new results found for particular graphs such as complete balanced k-partite graphs, k-trees and bounded treewidth graphs. Looking into complete balanced k-partite graphs we find a counter-example to the conjecture made in [14] that the optimal value for the MinLCA can be obtained using as many colours as the chromatic number.

3.4.1 Complete balanced k-partite graphs

Now that we have looked at the results for bipartite and complete graphs, we can consider the class of multipartite graphs and prove a closed result for the MinLCA problem.

Definition 3.4.1. (Complete balanced k-partite graphs) A *k-partite graph* is a graph whose vertices are or can be partitioned into k different independent sets.
A *complete k-partite graph* is a k-partite graph in which there is an edge between every pair of vertices from different independent sets.
A *complete balanced k-partite graph* is a complete k-partite graph where every set has the same cardinality.

It is immediate from the definition that every k-partite graph G has chromatic number $\chi(G) = k$. With this result we can deduce an exact result for the MinLCA problem for complete balanced k-partite graphs.

Proposition 3.4.1. *If $G = (V, E)$ is a complete balanced k-partite graph with $|V| = kn$, then*

$$MinLCA_k(G) = n^2 \frac{k^3 - k}{6} \tag{3.10}$$

Proof. As we have a complete balanced k-partite graph, we have k sets S_i with $i = \{1, ..., k\}$. Because the graph is balanced, every k-colouring has the same result for the MinLCA problem, so we choose to colour the nodes from set S_i with colour i.
Every node from a set has n edges with the nodes of every other set, so the number of edges between two sets are n^2. Then, for set S_1, the sum of the distances of all the edges with a node in S_1 is the sum of the distance between colours by the number of edges:

$$1n^2 + 2n^2 + ... + (k-2)n^2 + (k-1)n^2$$

Then if we sum the distances for every set we have:

$$\begin{array}{rl} S_1: & 1n^2 + 2n^2 + ... + (k-2)n^2 + (k-1)n^2 + \\ S_2: & 1n^2 + 2n^2 + ... + (k-2)n^2 + \\ & . \\ & . \\ S_{k-2}: & 1n^2 + 2n^2 + \\ S_{k-1}: & 1n^2 = \\ \hline \end{array}$$

$$= 1(k-1)n^2 + 2(k-2)n^2 + ... + (k-2)2n^2 + (k-1)1n^2$$
$$= n^2 \sum_{i=1}^{k-1} i(k-i)$$

If we divide the sum into two series we have $MinLCA(G) = n^2 \left(k \sum_{i=1}^{k-1} i - \sum_{i=1}^{k-1} i^2\right)$. As we know the two series have a closed sum, we can compute the result:

$$MinLCA(G) = n^2 \left(k\frac{(k-1)k}{2} - \frac{(k-1)k(2k-1)}{6}\right) = n^2 \frac{k^3 - k}{6}$$

□

This result for the parameterised formulation on the chromatic number leads us to question whether it is also true for the general formulation. In the next proposition we see that indeed it is.

Proposition 3.4.2. *If $G = (V, E)$ is a complete balanced k-partite graph with $|V| = kn$, then*

$$MinLCA(G) = MinLCA_k(G)$$

Proof. We have k sets S_i where every node from a set is connected to all the nodes from the other sets, and nodes from the same set are not connected.
Suppose we have an optimal solution where the nodes of some set S_i are distributed in m different colours.
Let S_{i_j} be the different subsets of the set S_i obtained by grouping into the same subset the nodes with the same colour, with $j \in \{1, ..., m\}$ and let ℓ_j be the sum of the distances between one node from S_{i_j} and all its adjacent vertices (note that for all vertices of S_{i_j}, ℓ_j has the same value as they all have the same adjacent vertices).
We can compute the sum of the distances of all the edges with a node in S_i as

$$|S_{i_1}|\ell_1 + |S_{i_2}|\ell_2 + ... + |S_{i_m}|\ell_m$$

Then, as we have an optimal solution, ℓ_j needs to have the same value for all subsets, otherwise we could recolour the nodes using the colour of the subset which has the minimum value of ℓ_j (it is possible as we do not have any edges between nodes from the same set and there cannot be any vertex from other sets with the same colour) and we would have a better solution, which is contradictory to the supposition that we have the optimal solution.
As we have the same value for all subsets, we can choose the colour of an arbitrary subset and assign it to the other subsets and the value for $MinLCA(G)$ will not change. This proves that the optimal solution of the parameterised version of the problem with the chromatic number is the same as the general version of the problem. □

In [14] the author made the conjecture that the optimal value for the MinLCA problem could be obtained using the chromatic number as the number of colours. More specifically:

Conjecture 3.4.3. *(Number of colours in an optimal linear colouring arrangement) The minimum linear colouring arrangement can be obtained using as many colours as the chromatic number. In other words, if G is a graph and $\chi(G) = k$, then*

$$MinLCA(G) = LCA_k(G)$$

We can disprove this conjecture with a counterexample using two complete balanced 3-partite graphs. First of all, to ensure the correctness of the proof we give an easy combinatorial result.

Lemma 3.4.4. *Let $\pi : [k] \to [k]$ be a* derangement*, i.e., a permutation without fixed points. Then there is some $i \in \{1, ..., k\}$ for which $|\pi(i) - i| > 1$.*

Proof. Suppose that for all integers i with $i \in \{1, ..., k\}$ we have $|\pi(i) - i| = 1$ (as it is a derangement, we cannot have $|\pi(i) - i| = 0$). Then, the only possible values of $\pi(i)$ are $i + 1$ and $i - 1$. If we look at $i = 1$, there is only one possible value for the permutation, $\pi(1) = 2$. As we have a bijection, for every $i \in \{1, ..., k\}$ we must have $\pi(i) = i + 1$. But if we look at $i = k$, we find that the permutation can only have one value, $\pi(k) = k - 1$. We reach a contradiction under the assumption that for all integers $i \in \{1, ..., k\}$ we have $|\pi(i) - i| = 1$.
This means that there has to be at least one integer i for which $|\pi(i) - i| > 1$. □

We can now show an example of a graph for which a solution computed with the chromatic number is worse than the optimal value, which uses more colours.

Exemple 3.4.1. Given two complete balanced 3-partite graphs, we join them adding edges between two sets in a way that for every set of one of the grapshs, all nodes from a set are connected to all the nodes from a set of the other graph, see Figure 3.4.

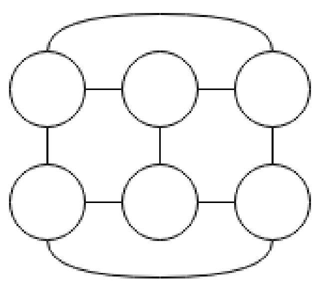

Figure 3.4: Counterexample for the conjecture of the number of colours in an optimal linear colouring arrangement.

This graph can be coloured using 3 colours, assigning different colours to the pairs of sets from the two graphs which are connected. In a more mathematical way, we assign 3 colours to one graph and then we make a derangement to the other graph, where we identify two sets from different graphs if they are connected with each other. As we have seen in Lemma 3.4.4, if we do it this way at least one element of the derangement will have distance greater than 1 for all possible solutions with three colours. In Figure 3.5 we can see the distances of colouring the graph with three and four colours. As changing the ordering of the colours does not change the cost of the MinLCA problem for complete balanced 3-partite graphs, we only need to look at the cost of the edges that join the two graphs. As we see, if we do it with four colours, we can obtain distance 1 between all joined sets, so we have a better solution using 4 colours than

using 3.
This proves that for this type of graph we have $MinLCA(G) < MinLCA_{\chi(G)}(G)$ and with this we can disprove Conjecture 3.4.3.

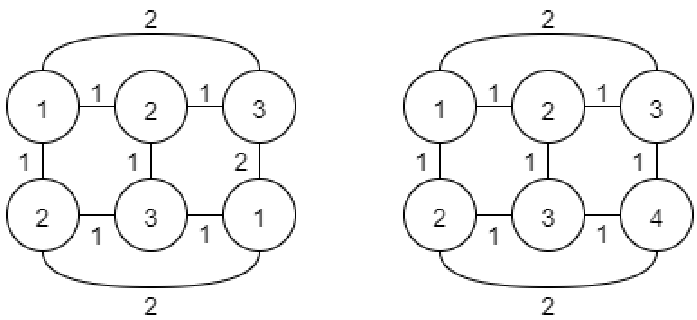

Figure 3.5: Comparison between using 3 colours and 4 colours for two complete balanced 3-partite graphs.

3.4.2 k-tree

These graphs are interesting because they are the maximal graphs with a given treewidth, i. e., they are graphs to which no more nodes can be added without increasing their treewidth.
For k-trees we have an exact result for the chromatic number, which leads us to an upper bound for the MinLCA problem.

Proposition 3.4.5. *If G is a k-tree, then $\chi(G) = k$*

Proof. It is immediate by the construction of the graph. We assign k colours to the initial complete graph of size k. Then for every node that we add, we connect it with $k-1$ nodes, so there is at least one node which is not connected to the node and has a different colour than the nodes connected. We assign this colour to the new node, so in the end we have k colours. □

Proposition 3.4.6. *If G is a k-tree with $k+t$ nodes, then*

$$MinLCA(G) \leq \frac{k^3-k}{6} + \left(k^2t - 2kt + t\right) \tag{3.11}$$

Proof. As we have seen before, for the initial complete graph of size k we have $MinLCA(G) = \frac{k^3-k}{6}$. Therefore, we only need to compute the partial sum of the distance for the t nodes added to the initial graph. For every node added, we connect it with $k-1$ nodes. Assuming we colour the graph with the chromatic number, the cost for every edge added is at maximum $k-1$, so then we can compute the upper bound:

$$MinLCA(G) \leq \frac{k^3-k}{6} + \sum_{j=1}^{t}\sum_{i=1}^{k-1}(k-1)$$

If we develop the series we get:

$$MinLCA(G) \leq \frac{k^3-k}{6} + \sum_{j=1}^{t}(k-1)^2 = \frac{k^3-k}{6} + (k-1)^2t = \frac{k^3-k}{6} + k^2t - 2kt + t$$

□

3.4.3 Bounded treewidth graphs

k-tree subgraphs have interesting properties. In fact, partial k-trees are defined either as a subgraph of a k-tree or as a graph with treewidth at most k. These graphs have the property that many combinatorial problems on graphs are solvable in polynomial time when restricted to them, for bounded values of k. If a family of graphs has bounded treewidth, then it is a subfamily of the partial k-trees, where k is the bound on the treewidth. Families of graphs with this property include outerplanar graphs, cactus graphs, pseudoforests, series-parallel graphs, Hallin graphs and Apollonian networks (see [1]).

We want to see that the parameterised version of our problem with bounded treewidth graphs can be solved in polynomial time. As we see in Section 2.3, given a bounded treewidth graph with treewidth w we can compute a rooted nice tree decomposition of the graph with the same width. We use dynamic programming on the tree decomposition.

Theorem 3.4.7. *Let (T, X) be a rooted nice tree decomposition of width w of a graph G with n vertices. Then* $\textsc{MinLCA}_k(G)$ *can be computed in time $f(w)n^{\mathcal{O}(1)} = k^{\mathcal{O}(w)+\mathcal{O}(1)}n$.*

Proof. We use dynamic programming on the rooted nice tree decomposition. For each node $v \in V(T)$ we keep a table $P_v(\varphi)$ for each $\varphi \in F_k(G[X_v])$ holding:

$$P_v(\varphi) = \begin{cases} \min_{\substack{\varphi' \in F_k(G(v)) \\ \varphi'|_{X_v} = \varphi}} L[\varphi'](G(v)) & \text{if some } \varphi' \text{ exists} \\ +\infty & \text{otherwise} \end{cases}$$

Then the value for the root of the tree decomposition $P_r(\emptyset)$ is the value for $\textsc{MinLCA}_k(G)$.

In order to ensure the correctness of our algorithm we use the properties for nice tree decompositions defined in 2.3. We deal with each type of node separately:

- Start node: In this case, for a node u we have $X_u = \{x\}$, where $x \in V(G)$. We have k possible colourings for the vertex: $\varphi \in F_k(\{x\})$, $\varphi(x) = i$, with $i \in \{1, ..., k\}$.
  As we only have one isolated node in $G(u)$, the cost is zero for all colourings, $P_u(\varphi) = 0$, $i \in \{1, ..., k\}$.

- Introduce node: In this case, for a node u and its only child v, we have $X_u = X_v + \{x\}$, where $x \in V(G)$. For every proper colouring φ of $G(v)$ we have k possible colourings for X_u. For $i \in \{1, ..., k\}$, we have

  $$\varphi_i(y) = \begin{cases} \varphi(y) & y \in X_v \\ i & y = x \end{cases}$$

  Then the cost for the subtree is

  $$P_u(\varphi) = \min_{\substack{i \in \{1,...,k\} \\ \varphi_i \in F_k(G(u))}} \left(P_v(\varphi) + \sum_{y \in X_v} |i - \varphi(y)| \right)$$

  This sum is correct because the subgraph induced by node u is the same as the subgraph induced by node v adding the node x and the edges between vertices of node v and x.

- Forget node: Now, for a node u and its only child v, we have $X_u = X_v - \{x\}$. For every proper colouring φ of $G(v)$ we subtract the cost of the vertex x as follows:

  $$P_u(\varphi) = P_v(\varphi) - \sum_{y \in X_v} (|\varphi(x) - \varphi(y)|)$$

In this case the formula is correct because the subgraph induced by node u is the same as the subgraph induced by node v without the edges connecting x.

- Join node: In this case we have a node u with two children v_1 and v_2, with $X_u = X_{v_1} = X_{v_2}$. Every proper colouring φ for $G(u)$ is also a proper colouring for $G(v_1)$ and $G(v_2)$. Then, we have

$$P_u(\varphi) = P_{v_1}(\varphi) + P_{v_2}(\varphi) - \sum_{xy \in E(X_u)} |\varphi(x) - \varphi(y)|$$

  This formula is correct because there are not any edges between $G(v_1)$ and $G(v_2)$, except for those in X_u. This way, the value for $P_u(\varphi)$ is the sum of the cost for the two children subtracting the cost of the edges of X_u, which are duplicate.

Finally, the root node r has $X_r = \emptyset$, so then we have $P_r(\emptyset) = MinLCA(G)$.
We compute the value for MinLCA$_k$ tracing back through the tree decomposition. Therefore, we have cost $k^{\mathcal{O}(w)+\mathcal{O}(1)}n$ as we wanted to prove.

□

The parameterised version of the Minimum linear arrangement problem (MinLA) has been largely studied by the community. In fact, the standard parameterisation of MinLA is fixed-parameter tractable (FPT). Although most parameterised problems are FPT parameterised by the treewidth of the input graph, graph layout problems are a notable exception. The MinLA problem is, in fact W[1]-hard for bounded treewidth graphs (see [6]).
This result contrasts with our result, for we have found that the MinLCA problem with a fixed number of colours and bounded treewidht is, in fact, fixed parameter tractable (FPT).
This leaves an open problem for the MinLCA problem parameterised only by the treewidth of the graph.

Open problem 3.4.1. Finding whether the minimum linear colouring arrangement problem parameterised by the treewidth of the input graph is in FPT or W[1]-hard.

Chapter 4

Algorithms and instances

In this chapter, we present the algorithms developed and the instances used in this project. First of all we define some basic components which help in simplifying the presentation of the algorithms later on. Afterwards, we look at the exact algorithms, which use backtracking with a combinatorial search. Following this we look at the heuristic proposal developed using a maximal independent set approach. Finally, we study the instances used in the previous study of the MinLCA and we also present some new instances influenced by the parameterised cases. All the algorithms have been developed using LEMON. In order to store the graphs, the algorithms use a graph structure from this library which is a simple and fast graph implementation. It is also quite memory efficient but it does not support node and edge deletion. It provides constant time counting for nodes and edges.

4.1 Basic components

First of all, in order to simplify the presentation of our algorithms, we define some basic components that we use repeatedly. We develop two functions that update the cost of the solution computed by our algorithm: `setColour` paints a given node with a given colour and updates the current cost and `unColour` uncolours the given node and updates the current cost. The pseudocode for these functions are given in Algorithm 4.1 and Algorithm 4.2. We also use a decisional function for deciding whether a node can be painted with a certain colour or not: `check_Node_Colour` checks if a given node can be painted with a given colour checking its neighbours. The pseudocode for this function is given in Algorithm 4.3. We also develop a decisional function which returns whether all nodes have been coloured: `check_All_Coloured`. The pseudocode for this is given in Algorithm 4.4. Finally we develop a function that checks whether a colour has been used: `Check_Colour_Used`. This can be found on Algorithm 4.5. Note that all these components can be computed in linear time $\mathcal{O}(n)$, where $n = |V|$.

Algorithm 4.1 Set colour

Global variables/conditions:
$G = (V, E)$, $|V| = n$, the current colouring φ and the current cost lca. v can be coloured by c.

Input:
Given $v \in V$ and a colour c.

Output:
Colours v with the colour c and updates the current cost lca.

```
function setColour(v, c)
    old_Colour ← φ(v)
    φ(v) ← c
    for uv ∈ E do
        u_Colour ← φ(u)
        if u_Colour ≠ Undefined then
            if old_Colour ≠ Undefined then
                lca ← lca - |u_Colour - old_colour|
            end if
            lca ← lca + |c - u_Colour|
        end if
    end for
end function
```

Algorithm 4.2 Uncolouring nodes

Global variables:
$G = (V, E)$, $|V| = n$, the current colouring φ and the current cost lca.

Input:
Given $v \in V$.

Output:
Uncolours v and updates the current cost lca.

```
function unColour(v)
    old_Colour ← φ(v)
    φ(v) ← Undefined
    for uv ∈ E do
        u_Colour ← φ(u)
        if u_Colour ≠ Undefined then
            lca ← lca - |u_Colour - old_colour|
        end if
    end for
end function
```

Algorithm 4.3 Check node colouring

```
    Global variables:
    G = (V, E), |V| = n and the current colouring φ.
Input:
    Given v ∈ V and a colour c.
Output:
    Checks if node v can be painted with colour c.
function CHECK_NODE_COLOUR(v, c)
    for uv ∈ E do
        if φ(u) = c then
            return false
        end if
    end for
    return true
end function
```

Algorithm 4.4 Check all coloured

```
    Global variables:
    G = (V, E), |V| = n and the current colouring φ.
Output:
    Checks if all nodes are coloured
function CHECK_ALL_COLOURED
    for u ∈ V do
        if φ(u) =Undefined then
            return false
        end if
    end for
    return true
end function
```

Algorithm 4.5 Check colour used

```
    Global variables:
    G = (V, E), |V| = n and the current colouring φ.
Input: Colour c
Output:
    Checks whether c has been used
function CHECK_COLOUR_USED(c)
    for u ∈ V do
        if φ(u) =c then
            return true
        end if
    end for
    return false
end function
```

4.2 Backtracking

We develop an exact algorithm using backtracking. A backtracking algorithm enumerates a set of partial candidates that, in principle, could be completed in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of candidate extension steps.
Conceptually, the partial candidates are represented as the nodes of a tree structure, the potential search tree. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further.
The backtracking algorithm traverses this search tree recursively, from the root down, in depth-first order. At each node, the algorithm checks whether the node can be completed to a valid solution. If it cannot, the whole sub-tree rooted at the node is skipped (pruned). Otherwise, the algorithm checks whether the node itself is a valid solution, and if so reports it to the user. For more information on backtracking, see [7].

4.2.1 Basic schema

This algorithm is based on a combinatorial search. It considers searching every possible combination in order to solve our optimization problem. The search tree is pruned to avoid considering cases which will not lead to any optimal solution.
The idea of our algorithm is as follows:

Given a stack with all the uncoloured nodes of our graph we do the following.

- If we have painted all the nodes and the stack is empty we check if it is the best colouring among all the previous colourings we had and we return ignoring the rest of the steps. Otherwise we ignore this step.
- We take the top node v of the stack.
- For every colour c, if v can be coloured with c we paint it, update the current cost, pop it off the stack and go back to the first step.
  Once we return to this step we will uncolour v, update the current cost and push v into the stack again.

This algorithm takes $\mathcal{O}(n^n)$ time in the worst case, which is a graph without edges where we have to check all possible colourings. As we have n vertices and the maximum number of colours is n, we have n^n possible colourings.
The pseudocode is given on Algorithm 4.6.

4.2.2 Improvements

If we analyse this algorithm we notice that we are allowing the possibility of using non-consecutive integers as colours, which clearly increases the cost.
We also notice that when the number of colours needed k are less than the number of nodes n and we use consecutive colours, we repeat each colouring $n-k$ times, because we have the same results whether we use use the colours $\{0, ..., k-1\}$ or $\{1, ..., k\}$.
To avoid this we make two improvements of our algorithm. The idea for this is making the recursive function dependent on the number of colours and the vertices and making sure that if we paint a node with a colour c greater than 0, then there is at least one node painted with the colour $c-1$.

This way we force our algorithm to use consecutive colours and always start with the colour 0. The sketch for this algorithm is as follows:

Given a vector v with all the nodes from the graph, for every colour c and index i of v:

- If the cost is greater than the best cost found or c is greater than the number of nodes and not all nodes are coloured then we don't explore this branch.
- If all the nodes are coloured, then if the cost is better than the best cost found we update the best colouring to the current colouring. Otherwise we return to the previous step.
- If non of the above is true then we take the node u with index i in v. If c is zero or c is greater than zero and there is at least one node painted with colour $c-1$ then, if u hasn't been painted and it is a valid colour we paint it and we go back to the first step for (c, $i+1$) if $i+1$ is a valid index or for ($c+1$, 0) otherwise. Then we uncolour u and we go to the first step with the same values as before without the node painted.

For this improvement we have the same computational cost for the worst case as in the first backtracking algorithm. The algorithm takes $\mathcal{O}(n^n)$ time to give a solution for the worst case. The pseudocode for this modification is given in Algorithm 4.7.

Algorithm 4.6 Backtracking

Global variables:
$G = (V, E)$, $|V| = n$, a stack with the uncoloured nodes S, the maximum number of colours $maxK$, the current cost lca and the best cost $best\_lca$.
Suppose we have a variable Solution = Not_Found

Input:
Let i be the number of coloured nodes.

Output:
Returns either the best solution found or reports that the graph cannot be coloured with $maxK$ colours.

```
function BACKTRACKING(i)
    if lca > best_lca then                         ▷ If the cost is greater than the best, return
        return
    end if
    if i ≥ n and S.empty() then                    ▷ If we have coloured all nodes
        if lca < best_lca then
            Solution ← Found
            best_colouring ← colouring
            best_lca ← lca
        end if
        return
    end if
    v ← S.top()                                    ▷ Top uncoloured node of the stack
    for c ← 0 to maxK − 1 do
        if CHECK_NODE_COLOUR(v, c) then            ▷ Check proper colouring if φ(v) = c
            SETCOLOUR(v, c)
            S.pop()
            BACKTRACKING(i+1)
            UNCOLOUR(v)
            φ(v) ← Undefined
            S.push(v)
        end if
    end for
end function
```

Algorithm 4.7 Backtracking for Colours

Global variables:
$G = (V, E)$, $|V| = n$, a vector V with all the nodes, the maximum number of colours $maxK$, the current cost lca and the best cost $best\_lca$.
Suppose we have a variable Solution = Not_Found

Input:
Let c be a colour and i the index of v.

Output:
Returns either the best solution or reports that the graph cannot be coloured with $maxK$ colours.

```
function BACKTRACKING_COLOURS(c, i)
    if lca > best_lca then                        ▷ If the cost is greater than the best, return
        return
    end if
    if c >= maxK and not CHECK_ALL_COLOURED() then
        return
    end if
    if CHECK_ALL_COLOURED then                    ▷ If we have coloured all nodes
        if lca < best_lca then
            Solution ← Found
            best_colouring ← colouring
            best_lca ← lca
        end if
        return
    end if
    u ← V[i]
    if c = 0 or (c > 0 and CHECK_COLOUR_USED(c-1)) then
        if φ(u) = -1 and CHECK_NODE_COLOURING(u, c) then
            SET_COLOUR(u, c)
            if i+1 < n then
                BACKTRACKING_COLOURS(c, i+1)
            else
                BACKTRACKING_COLOURS(c+1, 0)
            end if
        end if
        if i+1 < n then
            BACKTRACKING_COLOURS(c, i+1)
        else
            BACKTRACKING_COLOURS(c+1, 0)
        end if
    end if
end function
```

4.3 Maximal Independent Set Approach

These algorithms are based on finding maximal independent sets of a given graph and then assigning a colour to each set. First of all we define the concept of maximal independent sets.

Definition 4.3.1. (Maximal Independent Set) In graph theory, an *independent set* is a set of vertices of a graph such that for every two vertices in the set, there is no edge connecting the two.
A *maximal independent set* (MIS) is an independent set which is not a subset of any other independent set.

What we try to do for our algorithm is grouping the vertices of a graph in maximal independent sets and assigning a colour to each set.
The idea for our algorithm is as follows:

First of all, we find maximal independent sets for our graph. To do so, given an array of length $k+1$ with a set on every index of the array we put all the nodes of our graph into the first index. Then, starting with the first index of the array, we iterate through the sets until we have all vertices assigned to a maximal independent set. Let i be the index of the current set.

- If $i = k+1$, then the first k sets of the array have at least one vertex. We check if all the vertices have a set assigned, i.e., we check whether there is any vertex in the set $k+1$. If there is not, we have found our maximal independent sets. Otherwise, we have not found a solution.
- If there is at least a vertex in our set i, we select one vertex from the set erasing it from the set and we assign the set i to this vertex. Then we take all the neighbours of the vertex we have selected which are in the set, we erase them from the set and put them in the next set $i+1$. Then we go back to the first step until there are no more vertices in the set i.
- Otherwise, if the set $i+1$ has no vertices we go back to the first step with $i = k+1$. If the set $i+1$ does have some vertex we go back to the first step with the set i.

This algorithm takes $\mathcal{O}(n^2)$ time to execute in the worst case. This case is when we have a complete graph. We have n iterations of the algorithm in which we assign one node to a set, and on every iteration we look at the neighbours of the node in question. So if we have a complete graph, every iteration costs $n-1$ time because we look at the neighbours of the node in question. Therefore, in the worst case the algorithm takes $n^2 - n$ time to execute, i.e., $\mathcal{O}(n^2)$ time. The pseudocode for this algorithm (Independent_set) is given in Algorithm 4.8.

Once we have our maximal independent sets, we have to assign each of them a colour. Given r the number of sets obtained, we do it in 3 different ways:

- Combinations without repetition (Set_Combination): We look into all possible combinations for the colours without repeating them and we stay with the one which gives us the best cost. This takes $\mathcal{O}(r!)$ time.
- Best of Two (Best_of_two): We assign a colour c to one of the sets and then for the next sets we look the cost of assigning the colour $c+1$ and $c-1$ and we paint it with the one that gives us the best cost. This takes us $\mathcal{O}(r)$
- Random colouring (Rand_selection): We assign a colour randomly to every set. This takes us $\mathcal{O}(r)$ time.

The pseudocode is given in Algorithms 4.9, 4.10 and 4.11.

Algorithm 4.8 Maximal Independent Set

Global variables:

$G = (V, E)$, $|V| = n$, the maximum number of colours $maxK$, the number of colours used $k = maxK$, a vector of sets indep_sets with size $maxK$ +1, a map of nodes node_set where we will store the set index for each node.

Suppose we have a variable Solution = Not_Found

Input:

Let c be the index of the set we are looking into.

Output:

Returns either the vector of sets with our independent sets or there is no solution for $maxK$ colours.

```
function Independent_set(c)
    if c ≥ maxK then                                  ▷ If c is not a valid set position
        if indep_sets[maxK] is empty then             ▷ If there aren't any nodes left
            for v ∈ V do
                index ← node_set[v]
                indep_sets[index].insert(v)
            end for
        else
            Solution ← Not_Found
        end if
        return
    end if
    if indep_sets[c] is not empty then
        v ← indep_sets[c].begin()
        node_set[v] = c
        for uv ∈ E do
            if u is in indep_sets[c] then
                Erase u from indep_sets[c]
                Insert u in indep_sets[c+1]
            end if
        end for
        Independent_set(c)
    else
        if indep_sets[c+1] is empty then
            k ← c
            Independent_set(maxK)
        else
            Independent_set(c+1)
        end if
    end if
end function
```

Algorithm 4.9 Colour Sets with Set Combinations

Global variables:

$G = (V, E)$, $|V| = n$, the maximum number of colours $maxK$, the number of colours used (number of sets) k, the vector of sets indep_sets with size k, the current cost lca and the best cost $best\_lca$.

Suppose we have a variable Solution = Not_Found

Input:

Let i be the set index.

Output:

Returns the best colouring and the cost.

```
function SET_COMBINATION(i)
    if lca > best_lca then                 ▷ If the cost is greater than the best, return
        return
    end if
    if i ≥ k then
        if lca < best_lca then
            best_lca ← lca
            Solution ← Found
            best_colouring ← colouring
        end if
        return
    end if
    for c ← 0 to k do
        if c not used then
            for v ∈ indep_sets[i] do
                SETCOLOUR(v, c)
            end for
            SET_COMBINATION(i+1)
            for v ∈ indep_sets[i] do
                UNCOLOUR(v)
            end for
        end if
    end for
end function
```

Algorithm 4.10 Colour Sets with Best of Two

Global variables:
$G = (V, E)$, $|V| = n$, the maximum number of colours $maxK$, the number of colours used (number of sets) k, the vector of sets indep_sets with size k, the current cost lca and the best cost $best\_lca$.
Suppose we have a variable Solution = Not_Found

Output:
Returns the best colouring and the cost.

```
function Best_of_two
    max ← k
    min ← k
    for v ∈ indep_sets[0] do
        φ(v) ← k
        for uv ∈ E do
            if φ(u) ≠ Undefined then
                lca ← lca + |c − φ(u)|
            end if
        end for
    end for
    for i ← 1 to k do
        for v ∈ indep_sets[i] do
            setColour(v, max + 1)
        end for
        lca1 ← lca
        for v ∈ indep_sets[i] do
            unColour(v)
        end for
        for v ∈ indep_sets[i] do
            setColour(v, min − 1)
        end for
        lca2 ← lca
        if lca1 < lca2 then
            for v ∈ indep_sets[i] do
                unColour(v)
            end for
            for v ∈ indep_sets[i] do
                setColour(v, max + 1)
            end for
            max ← max+1
        else
            min ← min-1
        end if
    end for
    best_lca ← lca
    best_colouring ← colouring
end function
```

Algorithm 4.11 Colour Sets with Random Selection

Global variables: For every algorithm we have the following implicit parameters: $G = (V, E)$, $|V| = n$, the maximum number of colours $maxK$, the number of colours used (number of sets) k, the vector of sets indep_sets with size k, the current cost lca and the best cost $best\_lca$.

Suppose we have a variable Solution = Not_Found

Output:

Returns the best colouring and the cost.

```
function RAND_SELECTION
    set_colours ← -1                    ▷ Vector with size k and with -1 in every entry
    for i ← 0 to k do
        set_colours[i] ← i
    end for
    for i ← k − 1 to 1 do
        j ← random%i                    ▷ random integer between 0 and i-1
        old ← set_colours[i]
        set_colours[i] ← set_colours[j]
        set_colours[j] ← old
        for v ∈ indep_sets[i] do
            SETCOLOUR(v, set_colours[i])
        end for
    end for
    for v ∈ indep_sets[0] do
        SETCOLOUR(v, set_colours[0])
    end for
    best_lca ← lca
    best_colouring ← colouring
end function
```

4.4 Benchmarking platform

Now we show the different instances we use for our algorithms. We work with different types of graphs which might give interesting results for the MinLCA problem. Although the minimum linear arrangement problem was known before, in [11], the author used some particular graphs to do some benchmarks. Many articles about the minimum linear arrangement problem have used these particular instances since then (see Table 4.1 for the complete list). In [14], Isaac Sánchez used them for the minimum linear colouring arrangement problem, so we are also going to use them in order to compare the results. In the MinLCA Benchmarking, 4 different types of random graphs were chosen to study. We will use the same graphs in order to compare the results with our algorithms. We also present two new instances which we think might be interesting to study. This two instances are k-trees and almost 3-complete graphs.

Table 4.1: MinLA instances

For each graph, its name, number of nodes, number of edges, degree information (minimum/average/maximum), diameter and family.

Name	Nodes	Edges	Degree	Diameter	Family
`randomA1`	1000	4974	1/9.95/21	6	$\mathcal{G}_{n=1000,p=0.01}$
`randomA2`	1000	24738	28/49.47/72	3	$\mathcal{G}_{n=1000,p=0.05}$
`ramdomA3`	1000	49820	72/99.64/129	4	$\mathcal{G}_{n=1000,p=0.1}$
`randomA4`	1000	8177	4/16.35/29	4	$\mathcal{G}_{n=1000,p=0.0164}$
`randomG4`	1000	8173	5/16.34/31	23	$\mathcal{G}_{n=1000}(r = 0.075)$
`hc100`	1024	5120	10/10/10	10	10-hypercube
`mesh33x33`	1089	2112	2/3.88/4	64	33 × 33-mesh
`bintree10`	1023	1022	1/1.99/3	18	10-bintree
`3elt`	4720	13722	3/5.81/9	65	FE
`airfoil1`	4253	12289	3/5.78/10	65	
`crack`	10240	30380	3/5.93/9.00	121	
`whitaker3`	9800	28989	3/5.91/8	161	
`c1y`	828	1749	2/4.22/304	10	VLSI
`c2y`	980	2102	1/4.29/327	11	
`c3y`	1327	2844	1/4.29/364	13	
`c4y`	1366	2915	1/4.26/309	14	
`c5y`	1202	2557	1/4.25/323	13	
`gd95c`	62	144	2/4.65/15	11	GD
`gd96a`	1076	4676	1/3.06/111	20	
`gd96b`	111	193	2/3.47/47	18	
`gd96c`	65	125	2/3.84/6	10	
`gd96d`	180	228	1/2.53/27	8	
`small`	5	8	2/3/4	2	

4.4.1 Binomial random graphs

Definition 4.4.1. (Binomial random graph) G is a binomial random graph from the family $\mathcal{G}(n, p)$ if $|V(G)| = n$ and every edge of G exists with probability p.

A reason to study these graphs is that all graphs are potentially in them. In fact, $\mathcal{G}(n, p = 0.5)$ contains all graphs (with isomorphism) with equiprobability.

In the MinLA benchmarking p is chosen small because the interest is in sparse graphs, but it is large enough to ensure connected graphs.
For the MinLCA benchmarking instances are generated with p such that the expected number of edges is not within a constant factor of n^2. This way we generate not-too-dense graphs.
To choose an adequate probability p we put the number of edges as a function fo the number of vertices n, that is $|E(G)| = f(n) = \binom{n}{2}$. Then we consider a random variable M for the number of edges. This variable follows a binomial distribution, $M \sim Binom\left(\binom{n}{2}, p\right)$, so if we want to have $f(n)$ edges on average, we can use the formula for the expectation of M:

$$f(n) = E(M) = p\binom{n}{2} = p\frac{n(n-1)}{2} \approx p\frac{n^2}{2} \tag{4.1}$$

so then we can set $p = \dfrac{2f(n)}{n^2}$.

4.4.2 Random geometric graphs

Definition 4.4.2. (Random geometric graphs) G is a random geometric graph of the family $G(n; r)$ if it has n vertices uniformly distributed in some metric space and two $u \sim v$ if $d(u, v) \leq r$ for some distance d.

In the MinLA benchmarking, a random geometric graph with 1000 nodes and radius 0.075 (`randomG4`) is chosen.

For the MinLCA benchmarking, for implementation reasons the unit disc distance is chosen. These graphs are not too dense for a small radius r which may depend on n and usually they have some interesting properties.

For choosing the radius we choose a small radius $r > 0$. Since the distribution is uniform in the unit disc $D_1(0)$, which has an area of π, the average proportion of vertices adjacent to a vertex v (those in the disc $D_r(v)$) approaches the ratio between both disks, $\dfrac{\pi r^2}{\pi} = r^2$. Considering this, the degree of v is $deg(v) = nr^2$ on average. And since, by double counting, we have $2|E(G)| = \sum_{v \in V(G)} deg(v) \approx 2n^2r^2$, if we want $|E(G)| = f(n)$ we can take $r^2 = \dfrac{2f(n)}{n^2}$.

4.4.3 Graphs with cliques

These graphs were chosen in the MinLCA benchmarking in order to have more or less symmetric graphs for which we can know their optimal cost analytically in an easy way. That way we can know if our algorithms behave well for *easy* instances.

We generate this graphs in two ways: cycles of cliques and interconnected cliques.

Definition 4.4.3. (Cycles of cliques) G belongs to the family of cycles of cliques if given two integers s and t, G has t cliques isomorphic to K_s and there is a cycle connecting all the cliques (choosing two nodes from each clique).

Corollary 4.4.1. *If $G = (V, E)$ is a cycle of t cliques of order s, then*

$$MinLCA(G) = t\left(1 + \frac{s^3 - s}{6}\right) \tag{4.2}$$

Proof. We just have to notice that each clique has cost $\dfrac{s^3 - s}{6}$ by applying Theorem 3.3.5, and that the cycle connecting the cliques has length $2t$, with t edges from a different clique each.

If we colour the cycle using colours 1 and 2 (the subgraph is bipartite; it is an even cycle) the cliques can be optimally coloured because they have at least one edge with cost 1, which can be the one which is part of the cycle. □

Definition 4.4.4. (Interconnected cliques) G belongs to this family if given two integers s and t, G has t cliques each one isomorphic to K_s and two cliques are adjacent with probability p chosen (It is the same idea as the $\mathcal{G}(t,p)$ family with the end-vertices in each clique chosen at random).

The expected cost for this kind of graph is around $t\dfrac{s^3-s}{6} + MinLCA(H)$, where $H \in \mathcal{G}(t,p)$.

4.4.4 Outerplanar graphs

Generating random outerplanar graphs has been studied in different publications. In this case, for the MinLCA benchmarking, an algorithm was used which works by breaking the outer cycle in two parts, randomly adding chords from one part to the other and then adding more chords between the vertices of the same side. You can find the algorithms in [14], pg. 38.

4.4.5 k-tree

In order to test the algorithms with k-trees we have developed an algorithm to generate them. First of all, if we want to generate a k-tree of order $k+t$, the algorithm generates a complete graph of order k. Then it randomly chooses $k-1$ nodes from the graph and adds an edge between these nodes and a new node. From the resulting graph it randomly chooses $k-1$ nodes again and adds edges from these nodes to a new one until we have $k+t$ nodes.
In order to choose the $k-1$ nodes randomly we use the random class from LEMON with a random seed.

4.4.6 Almost 3-complete graphs

This family of graphs is a variation for the complete balanced 3-partite family. We take a complete balanced 3-partite graph and add a few new edges between nodes from the same set. More formally, we have:

Definition 4.4.5. (Almost 3-complete) G belongs to this family ($K_{\alpha_1,\alpha_2,\alpha_3}$) if it has three sets (S_1, S_2, S_3) of n nodes each where every node from each set is connected with all the nodes from the other sets, and there are α_i edges between nodes from set S_i ($i = \{1,2,3\}$).

The reason to study this graphs is because the chromatic number should be small and we want to see if the backtracking algorithm will have good results with this type of graphs.
In fact, we can give a very simple upper bound for the chromatic number.

Proposition 4.4.2. *If $K_{\alpha_1,\alpha_2,\alpha_3}$ is an almost 3-complete graph, then $\chi(G) \leq 3 + \alpha_1 + \alpha_2 + \alpha_3$*

Proof. It is immediate by the construction of the graph. We assign a different colour to every set, so we have three colours. Then for every edge that we add between two nodes of a set we assign to one of the nodes a new colour, and we do it for all new edges. Then we have $\alpha_1 + \alpha_2 + \alpha_3$ new colours. Note that in some cases when assigning the colour to a node from a new edge we could reuse the same colour used for another edge, hence the inequality. So then we have $\chi(G) \leq 3 + \alpha_1 + \alpha_2 + \alpha_3$. □

In order to test this instances with the algorithms we develop an algorithm to generate almost 3-complete graphs. This algorithm generates three sets of n nodes and connects every node from a set with all the nodes from the other sets. Then for every set i, it randomly chooses α_i pairs of nodes and adds an edge between them.
In order to choose the new edges added randomly, we use the random class from LEMON.

4.4.7 Trees, meshes and hypercubes

These three instances are graphs with optima solutions known for the minimum linear arrangement problem. They are included in the MinLA benchmarking.
A tree is an undirected graph in which any two vertices are connected by exactly one path. We have a binary tree with 10 levels (`bintree10`).
A mesh or grid graph, is a graph whose drawing, embedded in some Euclidean space $\mathbb{R}^n$, forms a regular tilling. This implies that the group of bijective functions that send the graph to itself is a lattice in the group-theoretical sense (see [4]). We have a 33x33 grid graph (`mesh33x33`).
Finally, an hypercube graph is a graph formed from the vertices and edges of an n-dimensional hypercube. In this case we have a 10-hypercube (`hc10`).

4.4.8 Other graphs

For the MinLA benchmarking, some other graphs were chosen. This instances come from different engineering applications and can be classified in graphs from finite element discretisation, graphs from VLSI design and graphs from graph-drawing competitions.
The VLSI family come from different circular layouts, and are `c1y` to `c5y`.
The graphs from finite element discretisation come from different fields such as fluid dynamics (`arifoil1` and `3elt`), earthquake wave propagation (`whitaker3`) and structural mechanics (`crack`).
Finally we have graphs from graph-drawing competitions. This graphs are mostly planar (`gd95c` and `gd96a` to `gd96d`).

Chapter 5

Experimental Results for Backtracking

In this chapter we analyse and present the results of the backtracking algorithm for the chosen instances. First of all, we detail the experiment design, afterwards we look at the results and we compare them with the results of [14].

5.1 Experiment design

The backtracking algorithm takes too much time to execute even in small graphs, so in order to analyse the results obtained with this algorithm we are going to do two types of restrictions. On the one hand, as done in [14] for the integer linear programming algorithm, we broaden the memory used in every execution to 8 threads and 8GB of RAM instead of 2, and we limit the time to 5h for each instance.
On the other hand, we limit the maximum number of colours for our algorithm, starting with two colours and increasing the number. In this case, if we find the first number of colours for which our algorithm has a solution, we find the chromatic number of the graph. For graphs that we already know the chromatic number, we limit the colours for this number directly (for example, bipartite or outerplanar graphs) and then we limit the number of colours to some slightly higher integer to see whether it finds the best result with the chromatic number or not.

Apart from this, for the different types of graphs chosen for the MinLCA problem, we execute the backtracking algorithm for small orders. This way we can determine the maximum order for which it is feasible to obtain an exact result with the backtracking algorithm for every type of graph.

5.2 Results

5.2.1 MinLA instances

If we look at the results obtained with the 5 hour limitation with the MinLA instances (see Table 4.1), we see that the algorithm has not finished for any graph except for the `small` graph and the bipartite graphs (`bintree`, `gd96b`, `gd96d`, `hc10` and `mesh33x33`), so we only get an exact result for these graphs. Despite this, we have found an upper bound for every graph, i.e., some value for the MinLCA problem was found for every instance, although it is not the optimal value.

Comparing our results to those of [14] for the integer linear programming, we see that the results found don't improve the results we had, however there were three graphs which did not have an upper bound (`crack`, `randomA3` and `whitaker3`), and for this graphs we have found one with our algorithm. In Table 5.1, we can see our results compared with those found with integer linear programming.
If we look at the results, in some cases even though the cost does not improve, the number of colours used is lower, so we have a better upper bound for the chromatic number for these graphs and we see that the cost seems not to decrease.

Now if we look at the results obtained by limiting the number of colours for each graph, we see that we don't have better results for the MinLCA than those that we found before. However, with these results we can deduce a better upper bound for the chromatic number of this graphs. In Table 5.2 we have a summary for the upper bounds known for the MinLCA problem and we have highlighted those for which we have improved the upper bound for the chromatic number.

5.2.2 MinLCA instances

In [14], the author chose not to execute the new MinLCA instances (binomial random graphs, random geometric graphs, graphs with cliques and outerplanar graphs) with integer linear programming, because of the amount of time needed for each execution compared to the poor results obtained. In our case, we have executed the backtracking algorithm for these instances with the restrictions specified.
For the restriction of 5 hours, we see that the local optimals found do not improve the results for the greedy algorithms, and the number of colours used is quite similar to those results.
For the restriction of the maximum number of colours, we tried limiting the number of colours to the lower number of colours used in other algorithms. For some instances there was no result found. For the outerplanar graphs we limited the colours to the chromatic number which we know is 3, but no result was found for any instance with 3 colours, so we executed the algorithm again with 4 colours.
In Tables 5.3, 5.5, 5.6 and 5.4 we can see the results from this two limitations compared with the greedy algorithm solutions. As we can see, for the executions in which we do get a result for the colour restriction, we find that the cost is less or equal than the result found for the 5 hour limitation, although it does not improve the greedy algorithm results. In Figure 5.1 we can see the results for the four types of graph.
Apart from this, we have ran the backtracking algorithm for the MinLCA problem choosing graphs with small orders from the MinLCA instances. We choose graphs of orders 10, 15, 20, 25 and 30. In Tables 5.9, 5.7, 5.8 and 5.10 we can see the results for these graphs and if the algorithm delivered an exact solution or not. As we can see, the backtracking algorithm only gave the optimal solution for graphs of order 10. Even so, it is possible that the algorithm has found the optimal solution even if it has not finished.
For the results for graphs with cliques, we have cliques of order 5 with two cliques adjacent with probability p. The cost of MinLCA for the cliques is $\dfrac{5^3-5}{6} = 20$. If t is the number of cliques, the cost for all cliques is $5t$. Then, if e is the number of edges between cliques, the minimum cost possible for these graphs is $5t + e$. If we look at the results we can see that for all graphs of order 15 the cost is the minimum possible, so the algorithm has found the optimal solution, although it has not finished. For graphs of order 20 we have the minimum cost for all graphs except for one in which we have $5t + e + 1$. In this graph we have 4 extra edges for 4 cliques, so at least one edge has to have distance 2. We conclude that for graphs of order 20 we also have the optimal result. In Table 5.9 we can see for which graphs the algorithm returned the

optimal solution even though it did not finish.
As we know, outerplanar graphs can be coloured using 3 colours. The upper bound stated in Proposition 3.2.4 gives us $MinLCA(G) \leq 2|E|$ for outerplanar graphs. If we look at the results obtained in Table 5.10, we can see that in all cases even though we do not know if we have optimal solutions for all the instances, we have that cost$/|E|$ is less than 1.5. So experimentally we have found a better upper bound for outerplanar graphs, which is that $MinLCA(G) \leq \frac{3}{2}|E|$. In [14], the author made the conjecture that this upper bound holds for all graphs with $\chi(G) \leq 3$.

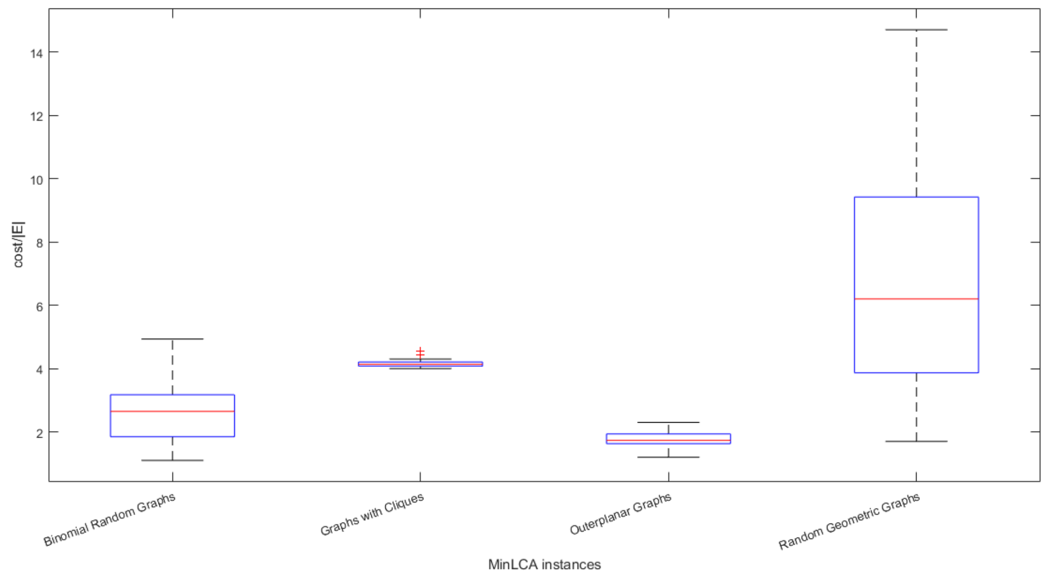

Figure 5.1: Backtracking results for MinLCA instances with small orders.

We have chosen to do a boxplot of the results for every instance with the cost$/|E|$ because this way we can compare the results for the different families of graphs used and we can also see whether the cost compared with the number of edges of the algorithm is similar for all instances as for graphs with cliques, or on the contrary, this cost is very disparate for the different instances as for random geometric graphs.

5.2.3 k-tree instances

We have done executions for graphs of orders 1000, 5000, 10000, 50000 and 100000 starting with a clique of size ten and adding nodes and edges using five different seeds for each order.
In Table 5.11 we can see the results for this instances with the two restrictions. As we can observe, restricting the colours to the chromatic number we have obtained equal or better result for k-trees than those found with the 5 hour restriction.

We also run executions for graphs of orders 10, 15, 20, 25 and 30 with 5 different random seeds. In Table 5.12 we can see the results for this instances and whether the algorithm reached the optimal solution or not. As we can see, for graphs of orders up to 15, the algorithm found an optimal solution. Comparing this with the results for the MinLCA instances, we can say that the algorithm works better with k-trees, because it has ended for bigger orders than the other instances. We also notice that some instances which have found the optimal solution have not found it with the chromatic number. Even so, there could be a solution with the chromatic number and the same cost as the optimal solution, so this does not prove Conjecture 3.4.3.

5.2.4 Almost 3-complete instances

In this case we make executions for graphs of orders 90, 450, 900, 4500 and 9000 for five different seeds with three sets of equal order and with every set having a reduced number of extra edges. In Table 5.13 we can see the results for these instances with the two restrictions. As we can observe, in this case we obtain the same result with the two restrictions. We also notice that the number of colours used is 6, which is the chromatic number, for we have added at least one edge on every set, and this forces to have at least one more colour for every set. This also means that we do not have any cycle of order 3, because if so, we would need more colours to have a proper colouring. Because of this, we can see that for every order of almost 3-complete instances we have the same cost, even though we have different graphs.

We also run the algorithm for graphs of orders 9, 15, 21, 27 and 33 for five different random seeds. These results can be found in Table 5.14. As we can see, for graphs of order 9 we obtain the optimal solution and the same cost for all graphs. That is because in this instance, only one edge was added on every set, therefore, the cost of MinLCA is the same for the different graphs. We notice that for all graphs we have used the chromatic number for the result as in the results for bigger orders. Also like in the results for bigger orders, for every instance of the same order we have the same cost (except for order 15).

Table 5.1: BACKTRACKING results for MINLA instances

For each graph, its name, number of nodes, number of edges, execution time, number of colours used and cost for integer linear programming(ILP) and backtracking and if one of the algorithms has found the optimal solution. Highlighted we have the instances for which we have found an upper bound.

			ILP			Backtracking			
Name	$\|V\|$	$\|E\|$	exec. time	colours	cost	exec. time	colours	cost	optimal?
3elt	4720	13722	18006.4151	7	21371	18000	5	22501	
airfoil1	4253	12289	18001.7815	9	19067	18000	5	20059	
bintree10	1023	1022	0.0970	2	1022	0.0008	2	1022	Back/ILP
c1y	828	1749	18000.2448	5	2130	18000	5	2540	
c2y	980	2102	18000.0842	5	2464	18000	5	2985	
c3y	1327	2844	18000.3257	5	3346	18000	6	4097	
c4y	1366	2915	18001.1299	5	3451	18000	5	4127	
c5y	1202	2557	18000.2934	5	2983	18000	6	3668	
crack	**10240**	**30380**	**inf**	–	–	**18000**	**6**	**48119**	
gd95c	62	144	18000.3073	6	198	18000	6	251	
gd96a	1076	4676	18000.1531	5	1875	18000	6	2633	
gd96b	111	193	0.0126	2	193	0.0001	2	193	Back/ILP
gd96c	65	125	39.5810	4	145	18000	4	185	ILP
gd96d	180	228	0.0132	2	228	18000	3	325	ILP
hc10	1024	5120	0.1747	2	5120	0.0011	2	5120	Back/ILP
mesh33x33	1089	2112	0.1838	2	2112	0.0005	2	2112	Back/ILP
randomA1	1000	4974	18001.5969	8	8925	18000	9	11334	
randomA2	1000	24738	18003.4304	19	133106	18000	19	140083	
ramdomA3	**1000**	**49820**	**inf**	–	–	**18000**	**31**	**455530**	
randomA4	1000	8177	18001.1336	10	21411	18000	11	23467	
randomG4	1000	8173	18001.3234	18	30193	18000	17	34614	
small	5	8	0.0103	3	10	0.0001	3	10	Back/ILP
whitaker3	**9800**	**28989**	**inf**	–	–	**18000**	**6**	**47758**	

Table 5.2: Chromatic number for MinLA instances

For each graph, its name, number of nodes, number of edges and chromatic number. Highlighted we have the instances for which we have found a better upper bound for the chromatic number

Name	$\|V\|$	$\|E\|$	$\chi(G)$
3elt	**4720**	**13722**	**5**
airfoil1	**4253**	**12289**	**5**
bintree10	1023	1022	2
c1y	828	1749	5
c2y	980	2102	5
c3y	1327	2844	5
c4y	1366	2915	5
c5y	1202	2557	5
crack	**10240**	**30380**	**6**
gd95c	62	144	6
gd96a	1076	4676	5
gd96b	111	193	2
gd96c	65	125	4
gd96d	180	228	2
hc10	1024	5120	2
mesh33x33	1089	2112	2
randomA1	1000	4974	8
randomA2	1000	24738	19
ramdomA3	1000	49820	31
randomA4	1000	8177	10
randomG4	**1000**	**8173**	**17**
small	5	8	3
whitaker3	**9800**	**28989**	**5**

Table 5.3: Backtracking results for binomial random graphs compared with greedy algorithm results

For each graph, its name, number of nodes, number of edges, edge probability, number of colours and the cost for each algorithm.

				5 hours limit		Colour limit			Greedy algorithm	
Instance	$\|V\|$	$\|E\|$	edge prob.	colours	cost	colour limit.	colours	cost	colours	cost
gr-1000-84458055	1000	4946	0.0100	8	11235	8	8	11235	8	9856
gr-1000-97953427	1000	4826	0.0100	8	10452	8	8	10452	8	9568
gr-1000-256996916	1000	5131	0.0100	7	11612	8	7	11612	8	10313
gr-1000-360678409	1000	4932	0.0100	9	10979	8	8	10961	8	9511
gr-1000-671420410	1000	5063	0.0100	8	11333	8	8	11321	8	10435
gr-5000-84458055	5000	31583	0.0025	10	78814	9	-	-	9	71418
gr-5000-97953427	5000	31376	0.0025	9	78334	9	9	78334	9	68967
gr-5000-256996916	5000	31325	0.0025	9	78423	9	9	78423	9	69809
gr-5000-360678409	5000	31424	0.0025	9	78312	9	9	78312	9	70272
gr-5000-671420410	5000	31478	0.0025	10	78863	9	-	-	9	70320
gr-10000-84458055	10000	65292	0.0013	9	166461	9	9	166461	9	157781
gr-10000-97953427	10000	64919	0.0013	10	164955	9	-	-	9	156565
gr-10000-256996916	10000	65254	0.0013	10	166185	10	10	166185	10	159034
gr-10000-360678409	10000	65049	0.0013	9	165291	10	9	165291	10	156714
gr-10000-671420410	10000	65052	0.0013	9	164875	10	9	164875	10	156899
gr-50000-84458055	50000	374759	0.0003	10	1022622	11	10	1022622	11	1016164
gr-50000-97953427	50000	374834	0.0003	11	1022367	11	11	1022367	11	1014582
gr-50000-256996916	50000	375072	0.0003	10	1024575	11	10	1024575	11	1014565
gr-50000-360678409	50000	374904	0.0003	11	1026871	11	11	1026871	11	1017139
gr-50000-671420410	50000	374802	0.0003	10	1027240	11	10	1027240	11	1015434
gr-100000-84458055	100000	1000917	0.0002	12	3212999	11	-	-	11	2246695
gr-100000-97953427	100000	999643	0.0002	12	3202271	11	-	-	11	2235742
gr-100000-256996916	100000	999734	0.0002	12	3192467	11	-	-	11	2240223
gr-100000-360678409	100000	1000601	0.0002	12	3210010	11	-	-	11	2238494
gr-100000-671420410	100000	999990	0.0002	12	3207353	11	-	-	12	2245051

Table 5.4: Backtracking results for random geometric graphs compared with greedy algorithm results

For each graph, its name, number of nodes, number of edges, radius, number of colours and the cost for each algorithm.

				5 hours limit		Colour limit			Greedy algorithm	
Instance	$\|V\|$	$\|E\|$	edge prob.	colours	cost	colour limit.	colours	cost	colours	cost
rg-gr-1000-84458055	1000	4660	0.0998	13	13709	12	-	-	12	12069
rg-gr-1000-97953427	1000	4875	0.0998	15	15094	14	-	-	14	13268
rg-gr-1000-256996916	1000	4770	0.0998	12	14214	12	12	14214	12	12739
rg-gr-1000-360678409	1000	4829	0.0998	12	14707	11	-	-	11	12846
rg-gr-1000-671420410	1000	4724	0.0998	12	13929	11	-	-	11	12298
rg-gr-5000-84458055	5000	29968	0.0496	19	141944	17	-	-	17	90318
rg-gr-5000-97953427	5000	29779	0.0496	16	142428	14	-	-	14	89082
rg-gr-5000-256996916	5000	30115	0.0496	17	145868	14	-	-	14	90458
rg-gr-5000-360678409	5000	30326	0.0496	17	146176	14	-	-	14	91792
rg-gr-5000-671420410	5000	30222	0.0496	17	145414	14	-	-	14	91486
rg-gr-10000-84458055	10000	65107	0.0365	13	150305	15	13	150305	16	205455
rg-gr-10000-97953427	10000	65292	0.0365	14	148930	15	14	148930	15	204612
rg-gr-10000-256996916	10000	65285	0.0365	14	149874	16	14	149874	16	206989
rg-gr-10000-360678409	10000	65611	0.0365	15	150315	17	15	150315	17	207278
rg-gr-10000-671420410	10000	65413	0.0365	14	151059	16	14	151059	16	207120
rg-gr-50000-84458055	50000	387828	0.0177	19	1479229	19	-	-	19	1362039
rg-gr-50000-97953427	50000	386336	0.0177	18	1467954	19	18	1467954	19	1350596
rg-gr-50000-256996916	50000	387865	0.0177	18	1477193	18	-	-	18	1358626
rg-gr-50000-360678409	50000	387382	0.0177	18	1474964	18	18	1474964	18	1354981
rg-gr-50000-671420410	50000	387836	0.0177	18	1475961	19	18	1475961	19	1358186
rg-gr-100000-84458055	100000	827038	0.0129	23	4833587	19	-	-	19	3019130
rg-gr-100000-97953427	100000	824534	0.0129	22	4797402	19	-	-	19	3002759
rg-gr-100000-256996916	100000	826747	0.0129	24	4819850	23	-	-	23	3022070
rg-gr-100000-360678409	100000	825219	0.0129	23	4810507	20	-	-	20	3009759
rg-gr-100000-671420410	100000	827318	0.0129	23	4840174	19	-	-	19	3030202

Table 5.5: Backtracking results for graphs with cliques compared with greedy algorithm results

For each graph, its name, number of nodes, number of edges, edge between cliques probability, number of colours and the cost for each algorithm.

				5 hours limit		Colour limit			Greedy algorithm	
Instance	$\|V\|$	$\|E\|$	edge prob.	colours	cost	colour limit.	colours	cost	colours	cost
qg-gr-1000-84458055	1000	7817	0.664	12	28937	11	-	-	11	17133
qg-gr-1000-97953427	1000	7812	0.664	12	28710	11	-	-	10	17129
qg-gr-1000-256996916	1000	7746	0.664	12	28485	11	-	-	11	17251
qg-gr-1000-360678409	1000	7752	0.664	12	28984	11	-	-	11	17059
qg-gr-1000-671420410	1000	7834	0.664	12	28872	11	-	-	11	17174
qg-gr-5000-84458055	5000	24719	0.0179	11	90232	11	11	90232	11	87780
qg-gr-5000-97953427	5000	24639	0.0179	11	90085	11	11	90085	12	87411
qg-gr-5000-256996916	5000	24737	0.0179	11	90311	11	11	90311	11	87877
qg-gr-5000-360678409	5000	24719	0.0179	11	90450	11	11	90450	11	87669
qg-gr-5000-671420410	5000	24752	0.0179	12	90327	11	-	-	11	87795
qg-gr-10000-84458055	10000	951216	0.0100	11	182581	11	11	182581	12	176750
qg-gr-10000-97953427	10000	949535	0.0100	11	181883	11	11	181883	11	176319
qg-gr-10000-256996916	10000	950836	0.0100	11	183246	11	11	183246	12	177538
qg-gr-10000-360678409	10000	950247	0.0100	11	182624	11	11	182624	11	176959
qg-gr-10000-671420410	10000	949938	0.0100	11	182707	11	11	182707	11	177179
qg-gr-50000-84458055	50000	5379451	0.0025	11	921485	11	11	921485	13	903204
qg-gr-50000-97953427	50000	5375529	0.0025	12	923648	11	-	-	12	901713
qg-gr-50000-256996916	50000	5374279	0.0025	11	923485	11	11	923485	12	901998
qg-gr-50000-360678409	50000	5375581	0.0025	11	924856	11	11	924856	12	902827
qg-gr-50000-671420410	50000	5377553	0.0025	12	921485	11	-	-	12	902669
qg-gr-100000-84458055	100000	11002715	0.0013	11	1995423	11	11	1995423	12	1820144
qg-gr-100000-97953427	100000	11001280	0.0013	11	1984125	11	11	1984125	13	1818598
qg-gr-100000-256996916	100000	10998321	0.0013	11	1993254	11	11	1993254	12	1819245
qg-gr-100000-360678409	100000	10998015	0.0013	12	1991567	11	-	-	13	1818732
qg-gr-100000-671420410	100000	11001453	0.0013	11	1982542	11	11	1982542	13	1819298

Table 5.6: Backtracking results for outerplanar graphs compared with greedy algorithm results

For each graph, its name, number of nodes, number of edges, edge between cliques probability, number of colours and the cost for each algorithm.

			5 hours limit		Colour limit			Greedy algorithm	
Instance	$\lvert V\rvert$	$\lvert E\rvert$	colours	cost	colour limit.	colours	cost	colours	cost
ro-gr-1000-84458055	1000	1455	4	1848	4	4	1848	4	1741
ro-gr-1000-97953427	1000	1391	4	1751	4	4	1750	4	1622
ro-gr-1000-256996916	1000	1549	4	2048	4	4	2048	4	1907
ro-gr-1000-360678409	1000	1398	4	1758	4	4	1756	4	1656
ro-gr-1000-671420410	1000	1416	4	1771	4	4	1769	4	1638
ro-gr-5000-84458055	5000	7189	4	9052	4	4	9052	4	8493
ro-gr-5000-97953427	5000	7237	4	9222	4	4	9222	4	8571
ro-gr-5000-256996916	5000	7263	4	9318	4	4	9318	4	8658
ro-gr-5000-360678409	5000	7273	4	9257	4	4	9257	4	8635
ro-gr-5000-671420410	5000	7381	4	9525	4	4	9525	4	8892
ro-gr-10000-84458055	10000	14108	4	17610	4	4	17610	4	16478
ro-gr-10000-97953427	10000	14331	4	18105	4	4	18105	4	16907
ro-gr-10000-256996916	10000	14589	4	18554	4	4	18554	4	17422
ro-gr-10000-360678409	10000	14289	4	17954	4	4	17954	4	16869
ro-gr-10000-671420410	10000	14364	4	18222	4	4	18222	4	16977
ro-gr-50000-84458055	50000	71633	4	85147	4	4	85147	4	84689
ro-gr-50000-97953427	50000	71629	4	86123	4	4	86123	4	84466
ro-gr-50000-256996916	50000	69876	4	83156	4	4	83156	4	81338
ro-gr-50000-360678409	50000	69687	4	81956	4	4	81956	4	80884
ro-gr-50000-671420410	50000	70576	4	84123	4	4	84123	4	82442
ro-gr-100000-84458055	100000	140777	4	175566	4	4	175566	4	164657
ro-gr-100000-97953427	100000	140496	4	174380	4	4	174380	4	163809
ro-gr-100000-256996916	100000	138842	4	170990	4	4	170990	4	161082
ro-gr-100000-360678409	100000	141429	4	176539	4	-	-	4	165695
ro-gr-100000-671420410	100000	141733	4	176911	4	4	176911	4	166229

Table 5.7: Backtracking results for binomial random graphs of small orders

For each graph, its name, number of nodes, number of edges, edge probability, number of colours, cost, cost/$|E|$ and if it is an optimal solution.

Instance	$\|V\|$	$\|E\|$	edge prob.	colours	cost	optimal?
gr-10-84458055	10	15	0.3322	3	17	YES
gr-10-97953427	10	17	0.3322	4	21	YES
gr-10-256996916	10	12	0.3322	3	14	YES
gr-10-360678409	10	10	0.3322	3	11	YES
gr-10-671420410	10	17	0.3322	5	21	YES
gr-15-84458055	15	35	0.2605	4	46	
gr-15-97953427	15	22	0.2605	3	26	
gr-15-256996916	15	25	0.2605	4	28	
gr-15-360678409	15	24	0.2605	4	27	
gr-15-671420410	15	28	0.2605	4	32	
gr-20-84458055	20	39	0.2161	4	53	
gr-20-97953427	20	29	0.2161	3	33	
gr-20-256996916	20	44	0.2161	4	58	
gr-20-360678409	20	40	0.2161	4	58	
gr-20-671420410	20	45	0.2161	4	61	
gr-25-84458055	25	67	0.1858	5	104	
gr-25-97953427	25	43	0.1858	4	56	
gr-25-256996916	25	57	0.1858	4	80	
gr-25-360678409	25	52	0.1858	4	68	
gr-25-671420410	25	61	0.1858	4	88	
gr-30-84458055	30	89	0.1636	5	148	
gr-30-97953427	30	55	0.1636	4	78	
gr-30-256996916	30	73	0.1636	4	111	
gr-30-360678409	30	65	0.1636	4	95	
gr-30-671420410	30	72	0.1636	4	103	

Table 5.8: Backtracking results for random geometric graphs of small orders

For each graph, its name, number of nodes, number of edges, radius, number of colours, cost and if it is an optimal solution.

Instance	$\|V\|$	$\|E\|$	radius	colours	cost	optimal?
`rg-gr-10-84458055`	10	17	0.5764	3	22	YES
`rg-gr-10-97953427`	10	16	0.5764	3	21	YES
`rg-gr-10-256996916`	10	16	0.5764	5	27	YES
`rg-gr-10-360678409`	10	17	0.5764	5	28	YES
`rg-gr-10-671420410`	10	12	0.5764	4	17	YES
`rg-gr-15-84458055`	15	36	0.5104	5	59	
`rg-gr-15-97953427`	15	36	0.5104	6	55	
`rg-gr-15-256996916`	15	41	0.5104	6	71	
`rg-gr-15-360678409`	15	44	0.5104	7	93	
`rg-gr-15-671420410`	15	32	0.5104	6	60	
`rg-gr-20-84458055`	20	57	0.4649	6	108	
`rg-gr-20-97953427`	20	61	0.14649	6	121	
`rg-gr-20-256996916`	20	69	0.4649	6	135	
`rg-gr-20-360678409`	20	72	0.4649	7	165	
`rg-gr-20-671420410`	20	49	0.4649	6	93	
`rg-gr-25-84458055`	25	85	0.431	6	180	
`rg-gr-25-97953427`	25	90	0.431	7	213	
`rg-gr-25-256996916`	25	116	0.431	8	302	
`rg-gr-25-360678409`	25	105	0.431	8	272	
`rg-gr-25-671420410`	25	85	0.431	7	198	
`rg-gr-30-84458055`	30	112	0.4044	7	268	
`rg-gr-30-97953427`	30	123	0.4044	8	332	
`rg-gr-30-256996916`	30	153	0.4044	9	441	
`rg-gr-30-360678409`	30	136	0.4044	9	398	
`rg-gr-30-671420410`	30	120	0.4044	8	327	

Table 5.9: Backtracking results for graphs with cliques of small orders

For each graph, its name, number of nodes, number of edges, edge between cliques probability, number of colours, cost and if it is an optimal solution.

Instance	$\|V\|$	$\|E\|$	edge prob.	colours	cost	optimal?
qg-gr-10-84458055	10	21	0.5	5	41	YES
qg-gr-10-97953427	10	21	0.5	5	41	YES
qg-gr-10-256996916	10	20	0.5	5	40	YES
qg-gr-10-360678409	10	20	0.5	5	40	YES
qg-gr-10-671420410	10	20	0.5	5	40	YES
qg-gr-15-84458055	15	32	0.5283	5	62	YES
qg-gr-15-97953427	15	33	0.5289	5	63	YES
qg-gr-15-256996916	15	31	0.5283	5	61	YES
qg-gr-15-360678409	15	31	0.5283	5	61	YES
qg-gr-15-671420410	15	32	0.5283	5	62	YES
qg-gr-20-84458055	20	42	0.5	5	82	YES
qg-gr-20-97953427	20	43	0.5	5	83	YES
qg-gr-20-256996916	20	41	0.5	5	81	YES
qg-gr-20-360678409	20	42	0.5	5	82	YES
qg-gr-20-671420410	20	44	0.5	5	85	YES
qg-gr-25-84458055	25	54	0.4644	5	105	
qg-gr-25-97953427	25	55	0.4644	5	105	YES
qg-gr-25-256996916	25	52	0.4644	5	102	YES
qg-gr-25-360678409	25	52	0.4644	5	103	
qg-gr-25-671420410	25	58	0.4644	5	111	
qg-gr-30-84458055	30	66	0.4308	5	127	
qg-gr-30-97953427	30	67	0.4308	5	129	
qg-gr-30-256996916	30	63	0.4308	5	127	
qg-gr-30-360678409	30	64	0.4308	5	125	
qg-gr-30-671420410	30	71	0.4308	5	137	

Table 5.10: Backtracking results for outerplanar graphs of small orders

For each graph, its name, number of nodes, number of edges, number of colours, cost and if it is an optimal solution.

Instance	$\lvert V \rvert$	$\lvert E \rvert$	colours	cost	optimal?
`ro-gr-10-84458055`	10	13	3	14	YES
`ro-gr-10-97953427`	10	16	3	20	YES
`ro-gr-10-256996916`	10	15	3	18	YES
`ro-gr-10-360678409`	10	14	4	16	YES
`ro-gr-10-671420410`	10	13	3	15	YES
`ro-gr-15-84458055`	15	23	4	28	
`ro-gr-15-97953427`	15	23	4	28	
`ro-gr-15-256996916`	15	18	4	20	
`ro-gr-15-360678409`	15	23	3	26	
`ro-gr-15-671420410`	15	17	3	18	
`ro-gr-20-84458055`	20	31	4	39	
`ro-gr-20-97953427`	20	29	3	33	
`ro-gr-20-256996916`	20	29	4	33	
`ro-gr-20-360678409`	20	32	3	39	
`ro-gr-20-671420410`	20	26	4	28	
`ro-gr-25-84458055`	25	36	3	41	
`ro-gr-25-97953427`	25	36	4	41	
`ro-gr-25-256996916`	25	40	3	50	
`ro-gr-25-360678409`	25	39	3	47	
`ro-gr-25-671420410`	25	36	4	43	
`ro-gr-30-84458055`	30	50	4	64	
`ro-gr-30-97953427`	30	51	4	69	
`ro-gr-30-256996916`	30	47	3	58	
`ro-gr-30-360678409`	30	47	3	57	
`ro-gr-30-671420410`	30	42	3	51	

Table 5.11: Backtracking results for k-trees

For each graph, its name, number of nodes, number of edges, number of colours and the cost for each algorithm.

			5 hour limit		Colour limit		
Instance	$\|V\|$	$\|E\|$	colours	cost	colour limit.	colours	cost
kt-gr-1000-84458055	1000	8955	10	23998	10	10	23993
kt-gr-1000-97953427	1000	8955	10	23891	10	10	23891
kt-gr-1000-256996916	1000	8955	10	24046	10	10	24044
kt-gr-1000-360678409	1000	8955	10	24128	10	10	24128
kt-gr-1000-671420410	1000	8955	10	23916	10	10	23915
kt-gr-5000-84458055	5000	41355	10	109940	10	10	109924
kt-gr-5000-97953427	5000	41355	10	110204	10	10	110188
kt-gr-5000-256996916	5000	41355	10	110524	10	10	110515
kt-gr-5000-360678409	5000	41355	10	110159	10	10	110149
kt-gr-5000-671420410	5000	41355	10	109673	10	10	109655
kt-gr-10000-84458055	10000	89955	10	238552	10	10	238536
kt-gr-10000-97953427	10000	89955	10	238676	10	10	238648
kt-gr-10000-256996916	10000	89955	10	238607	10	10	238588
kt-gr-10000-360678409	10000	89955	10	237938	10	10	237909
kt-gr-10000-671420410	10000	89955	10	238792	10	10	238787
kt-gr-50000-84458055	50000	413955	10	1095805	10	10	1095805
kt-gr-50000-97953427	50000	413955	10	1095275	10	10	1095245
kt-gr-50000-256996916	50000	413955	10	1093715	10	10	1093706
kt-gr-50000-360678409	50000	413955	10	1093613	10	10	1093594
kt-gr-50000-671420410	50000	413955	10	1094279	10	10	1094265
kt-gr-100000-84458055	100000	899955	10	2234587	10	10	2234510
kt-gr-100000-97953427	100000	899955	10	2229854	10	10	2229786
kt-gr-100000-256996916	100000	899955	10	2231256	10	10	2231203
kt-gr-100000-360678409	100000	899955	10	2228452	10	10	2228412
kt-gr-100000-671420410	100000	899955	10	2232485	10	10	2232415

Table 5.12: Backtracking results for k-trees of small orders

For each graph, its name, number of nodes, number of edges, number of colours, cost and if it is an optimal solution.

Instance	$\|V\|$	$\|E\|$	colours	cost	optimal?
kt-gr-10-84458055	10	30	5	55	YES
kt-gr-10-97953427	10	30	5	52	YES
kt-gr-10-256996916	10	30	5	50	YES
kt-gr-10-360678409	10	30	6	54	YES
kt-gr-10-671420410	10	30	5	51	YES
kt-gr-15-84458055	15	50	5	81	YES
kt-gr-15-97953427	15	50	6	82	YES
kt-gr-15-256996916	15	50	5	84	YES
kt-gr-15-360678409	15	50	5	83	YES
kt-gr-15-671420410	15	50	5	86	YES
kt-gr-20-84458055	20	135	10	413	
kt-gr-20-97953427	20	135	10	398	
kt-gr-20-256996916	20	135	10	387	
kt-gr-20-360678409	20	135	11	414	
kt-gr-20-671420410	20	135	10	401	
kt-gr-25-84458055	25	180	10	565	
kt-gr-25-97953427	25	180	10	508	
kt-gr-25-256996916	25	180	10	526	
kt-gr-25-360678409	25	180	11	552	
kt-gr-25-671420410	25	180	10	543	
kt-gr-30-84458055	30	315	15	1398	
kt-gr-30-97953427	30	315	15	1358	
kt-gr-30-256996916	30	315	15	1372	
kt-gr-30-360678409	30	315	15	1437	
kt-gr-30-671420410	30	315	15	1408	

Table 5.13: Backtracking results for almost 3-complete graphs

For each graph, its name, number of nodes, number of edges, number of colours and the cost for each algorithm.

			5 hour limit		Colour limit		
Instance	$\|V\|$	$\|E\|$	colours	cost	colour limit.	colours	cost
tk-gr-90-84458055	90	2706	6	7086	6	6	7086
tk-gr-90-97953427	90	2706	6	7086	6	6	7086
tk-gr-90-256996916	90	2706	6	7086	6	6	7086
tk-gr-90-360678409	90	2706	6	7086	6	6	7086
tk-gr-90-671420410	90	2706	6	7086	6	6	7086
tk-gr-450-84458055	450	67512	6	179412	6	6	179412
tk-gr-450-97953427	450	67512	6	179412	6	6	179412
tk-gr-450-256996916	450	67512	6	179412	6	6	179412
tk-gr-450-360678409	450	67512	6	179412	6	6	179412
tk-gr-450-671420410	450	67512	6	179412	6	6	179412
tk-gr-900-84458055	900	270021	6	718821	6	6	718821
tk-gr-900-97953427	900	270021	6	718821	6	6	718821
tk-gr-900-256996916	900	270021	6	718821	6	6	718821
tk-gr-900-360678409	900	270021	6	718821	6	6	718821
tk-gr-900-671420410	900	270021	6	718821	6	6	718821
tk-gr-4500-84458055	4500	6750048	6	17994048	6	6	17994048
tk-gr-4500-97953427	4500	6750048	6	17994048	6	6	17994048
tk-gr-4500-256996916	4500	6750048	6	17994048	6	6	17994048
tk-gr-4500-360678409	4500	6750048	6	17994048	6	6	17994048
tk-gr-4500-671420410	4500	6750048	6	17994048	6	6	17994048
tk-gr-9000-84458055	9000	27000093	6	69542345	6	6	69542345
tk-gr-9000-97953427	9000	27000093	6	69542345	6	6	69542345
tk-gr-9000-256996916	9000	27000093	6	69542345	6	6	69542345
tk-gr-9000-360678409	9000	27000093	6	69542345	6	6	69542345
tk-gr-9000-671420410	9000	27000093	6	69542345	6	6	69542345

Table 5.14: Backtracking results for almost 3-complete graphs of small orders

For each graph, its name, number of nodes, number of edges, number of colours, cost and if it is an optimal solution.

Instance	$\|V\|$	$\|E\|$	colours	cost	optimal?
tk-gr-9-84458055	9	30	6	59	YES
tk-gr-9-97953427	9	30	6	59	YES
tk-gr-9-256996916	9	30	6	59	YES
tk-gr-9-360678409	9	30	6	59	YES
tk-gr-9-671420410	9	30	6	59	YES
tk-gr-15-84458055	15	81	6	150	
tk-gr-15-97953427	15	81	6	150	
tk-gr-15-256996916	15	81	6	160	
tk-gr-15-360678409	15	81	6	160	
tk-gr-15-671420410	15	81	6	150	
tk-gr-21-84458055	21	153	6	312	
tk-gr-21-97953427	21	153	6	312	
tk-gr-21-256996916	21	153	6	312	
tk-gr-21-360678409	21	153	6	312	
tk-gr-21-671420410	21	153	6	312	
tk-gr-27-84458055	27	249	6	509	
tk-gr-27-97953427	27	249	6	509	
tk-gr-27-256996916	27	249	6	509	
tk-gr-27-360678409	27	249	6	509	
tk-gr-27-671420410	27	249	6	509	
tk-gr-33-84458055	33	369	6	753	
tk-gr-33-97953427	33	369	6	753	
tk-gr-33-256996916	33	369	6	753	
tk-gr-33-360678409	33	369	6	753	
tk-gr-33-671420410	33	369	6	753	

Chapter 6

Experimental results for Maximal Independent Set

In this chapter we are going to look at the results with the Maximal Independent Set approach. As done in the previous chapter, first we detail the experiment design and then we look at the results obtained.

6.1 Experiment design

6.1.1 MinLA instances

In order to compare the results of these instances (see Table 4.1) with those given in [14], we follow the same experiment design. We choose 50 randomly distributed integers from the set $\{1,..., 1 \times 10^9\}$ as seeds for the graph reading method. This method does a uniform ordering of the vertices using the *Mersenne twister* random generator from LEMON. As the maximal independent set found for the algorithm is different depending on the first node picked, this approach helps us make a random exploration of the graph.

6.1.2 MinLCA instances

We run the algorithms for graphs of orders 1000, 5000, 10000, 50000 and 100000 using five different seeds for each order and type of graph. As done in [14], we have done one execution of the algorithms per random instance. In this case, we have not ran the Set Combinations algorithm, because the time of execution was too long and as we can see in the results obtained for the MinLA instances, the results are similar to those obtained with the Best of Two approach.

6.1.3 k-tree instances

We run the algorithm for k-trees of orders 1000, 5000, 10000, 50000, 100000 using five different random seeds for each order and with every set having a reduced number of extra edges.

6.1.4 Almost 3-complete instances

We run executions for almost 3-complete graphs of orders 90, 450, 900, 4500, 9000 using five different random seeds for each order and with every set having a reduced number of extra edges. We do not execute the algorithm for instances of greater orders because the graph uses a lot of memory in order to store the edges.

6.2 Results

6.2.1 MinLA instances

The results for this experiment are given in Table 6.1. We observe that for some graphs (`randomA2`, `randomA3` and `randomG4`) we do not have results for the Set Combinations algorithm. This is because the computational cost was too high compared to the other two results.

To analyse and compare the results we look at Figure 6.1. As we can see, we have one plot for every algorithm. It is clear that the Best of Two and Set Combinations algorithms have quite a similar result and they are quite different from the results from the Random Selection algorithm. We notice that the scale for the Random Selection algorithm is different than the other two plots, so to compare them, we look at Figure 6.2.

In this figure, we observe that the Random Selection algorithm has a big interquartile difference and the median is much higher than the median of the Best of Two algorithm, so the results for this algorithm are hihgly dependent on the ordering of the vertices. Then we can conclude that the Random Selection algorithm does not give good results if we compare it with the Best of Two algorithm.

Moreover, as the Best of Two and the Set Combinations algorithms are quite similar, but the Set Combinations has a higher computational cost, we conclude that the Best of Two algorithm is the best option out of the three algorithms presented for these instances.

6.2.2 MinLCA instances

We have executed the algorithm for binomial random graphs, random geometric graphs, random outerplanar graphs and graphs with cliques. The results are given in Table 6.2, Table 6.3, Table 6.4 and Table 6.5.

If we look, for example, at the results for binomial random graphs, we see that the Random Selection results are between a 25% and a 59% worse than those for the Best of Two algorithm. If we look at the results for graphs with cliques, this percentages are a little lower, between 7% and 38% worse. If we look at Figure 6.3, we can see a comparison between the four types of graph and the two algorithms used. As stated before, the Random Selection algorithm is highly dependant on the ordering of the vertices, because the interquartile difference is very big compared with the Best of Two algorithm.

If we compare the four instances, we can see that the best and more adjusted results are for outerplanar graphs, and the worse results are those from the random geometric graphs. In fact, for the latter results, the interquartile difference for the Best of Two algorithm is quite significant if we compare it with the other instances.

6.2.3 k-tree instances

The results for this experiment are given in Table 6.6. As we can clearly see, the number of colours used is always the chromatic number and as stated before, the Best of Two algorithm gives much better results than the Random Selection algorithm. If we compute the value of the upper bound found in Proposition 3.4.6, we can see that our result improves it significantly. For example, for the first instance the upper bound is 80355 and our result is 22270.

In this case the difference between the two algorithms is very similar in all instances. In fact, the Random Selection algorithm is between a 54% and a 56% worse than the Best of Two algorithm.

6.2.4 Almost 3-complete instances

The results for this experiment are given in Table 6.7. As stated in the results for the backtracking algorithm, the chromatic number is used for all instances and we have the same results for all almost 3-complete instances of the same order. In this case we have the worst result for the Random Selection algorithm, we can see that for all instances the cost for Random Selection is more than doubles the cost for the Best of Two algorithm. This is logical because the number of edges for this instances is greater than the number of edges for the other instances, so the cost increases significantly for worse results.

In Figure 6.4 we can see a comparison between all algorithms used in MinLCA benchmarking. We observe that the random geometric graph instances have a big interquartile difference compared with the other instances, so this instances are highly dependant on the ordering of the vertices, because they give very different results for different instances. We notice that the outerplanar graphs and the almost 3-complete graphs are the ones with the best results and the interquartile range is very small. This is due to the fact that the algorithm uses between 4 and 5 colours for outerplanar graphs and 6 colours for almost 3-complete graphs for each instance. This contributes to giving similar results for all instances.

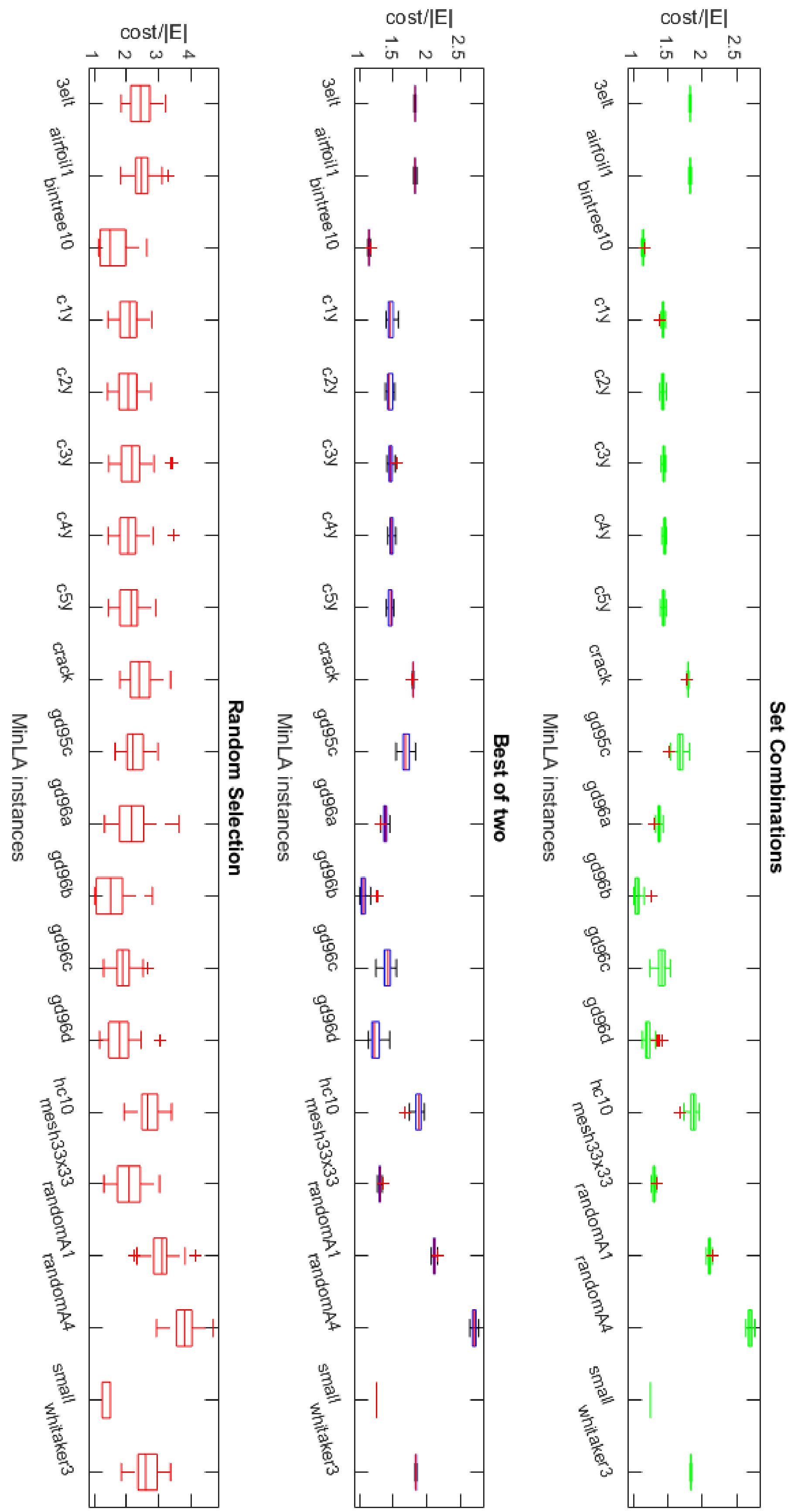

Figure 6.1: Results of Maximal Independent Set for MinLA instances

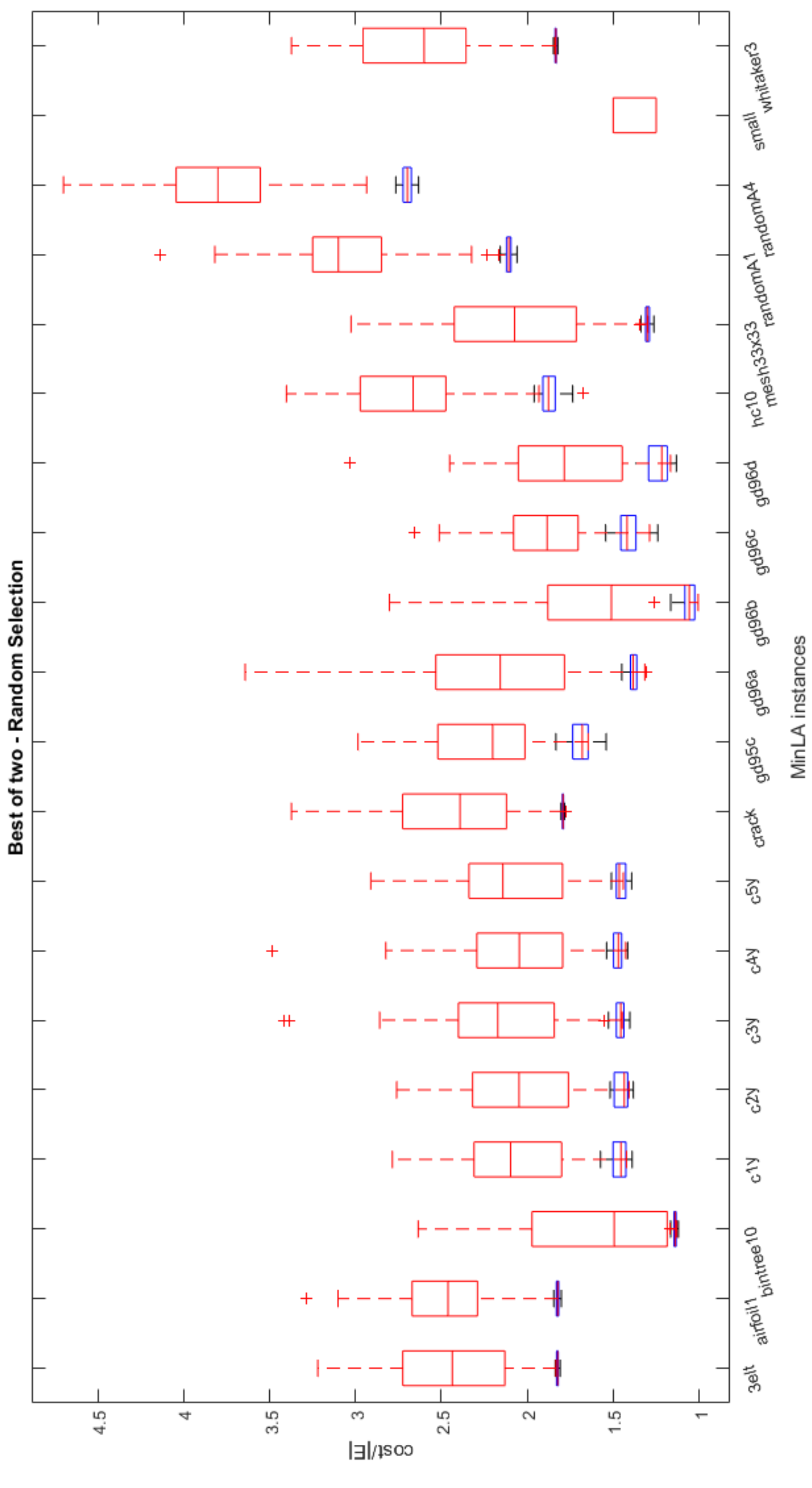

Figure 6.2: Comparison between Best of Two and Random Selection algorithms

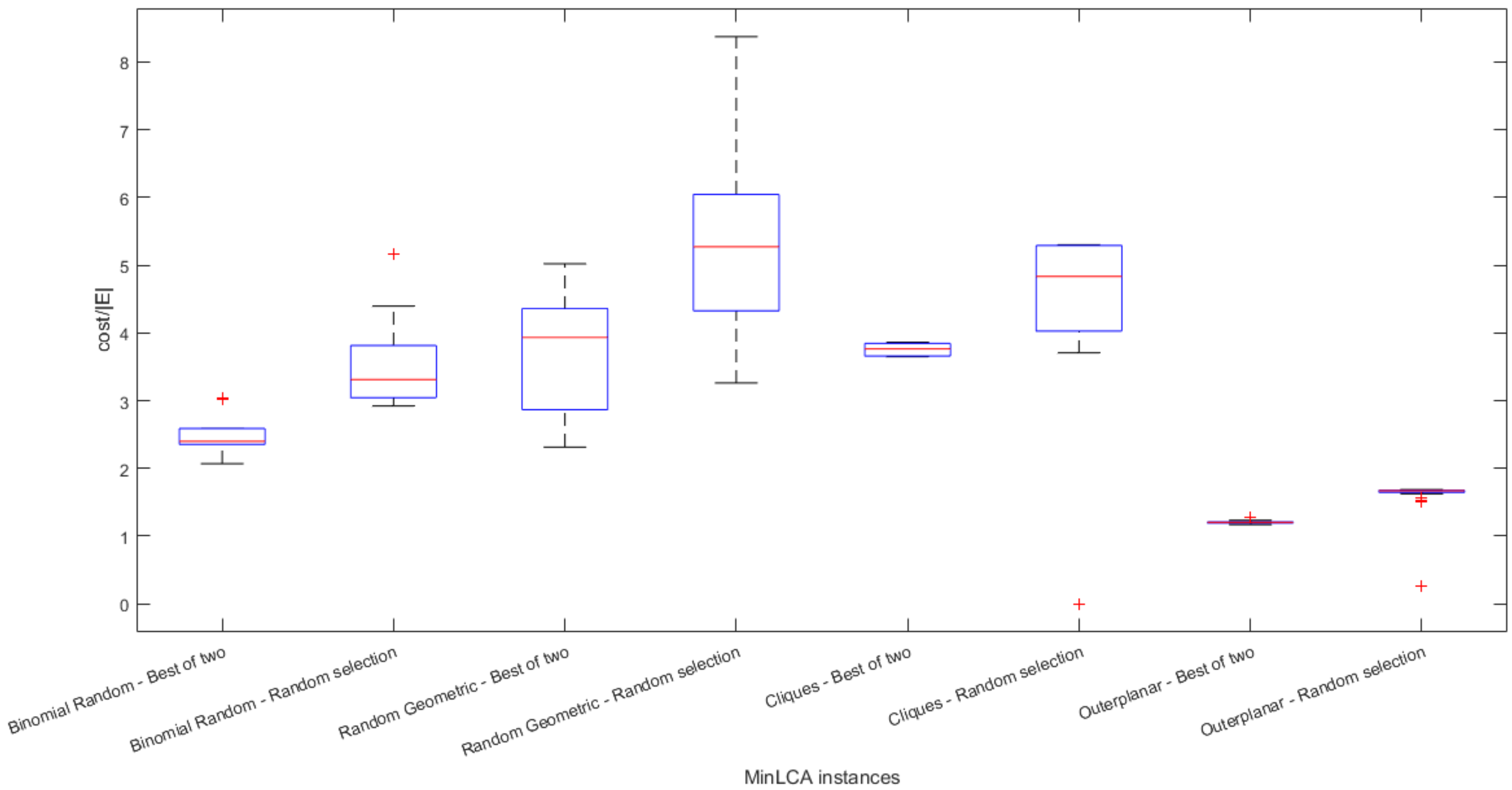

Figure 6.3: Results of Maximal Independent Set for MinLCA instances

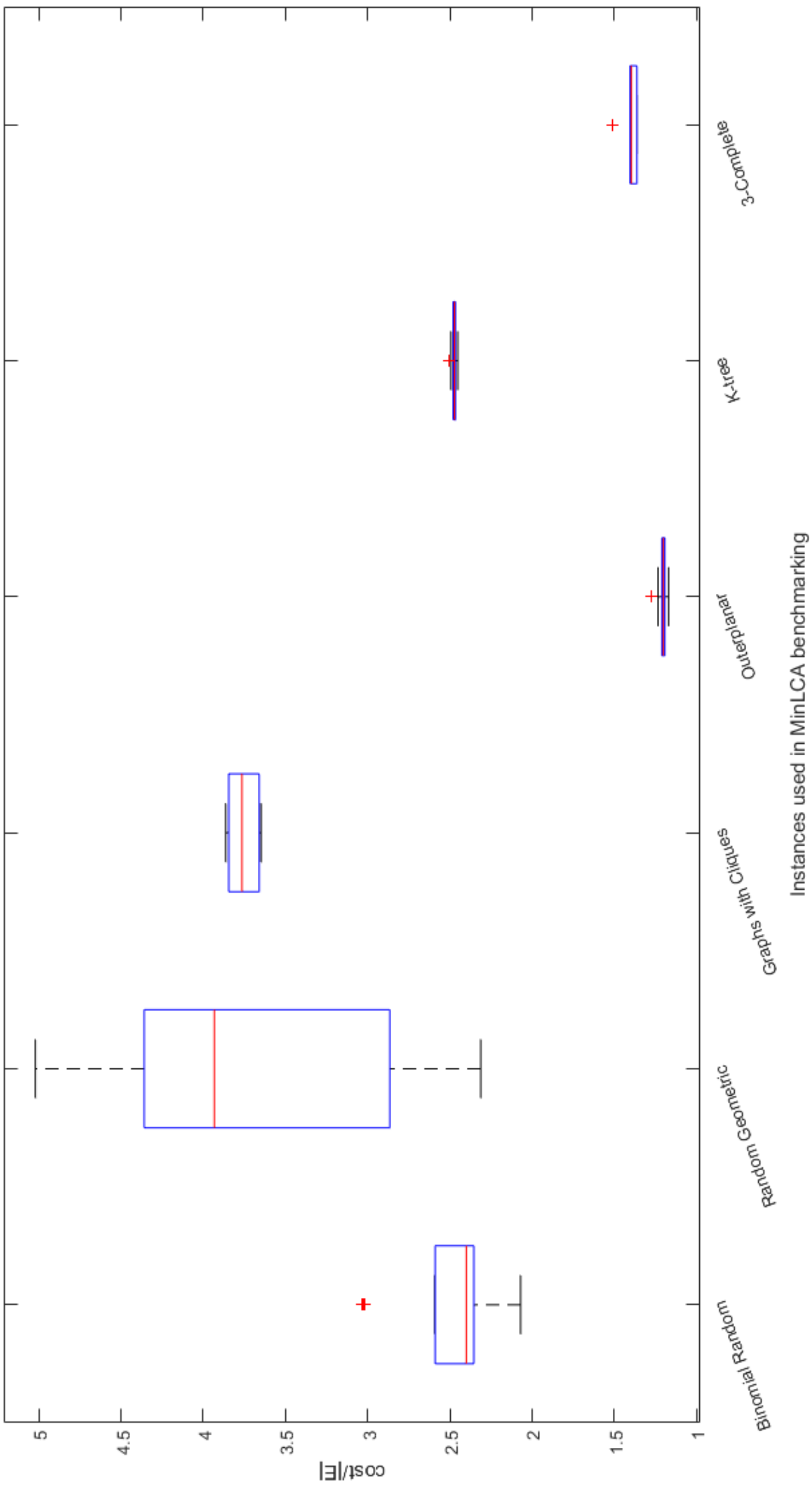

Figure 6.4: Results of Maximal Independent Set for all instances used in MinLCA benchmarking.

Table 6.1: MaxInSet results for MinLA instances

For each graph, its name, number of nodes, number of edges, average number of colours used and cost for every algorithm (minimum/average/median/maximum).

				Set Combinations - cost/$\|E\|$				Best of Two - cost/$\|E\|$				Random Selection - cost/$\|E\|$			
Name	$\|V\|$	$\|E\|$	avg. colours	min.	avg.	median	max.	min.	avg.	median	max.	min.	avg.	median	max.
3elt	4720	13722	6.32	1.81	1.82	1.82	1.84	1.81	1.82	1.82	1.84	1.84	2.38	2.39	3.22
airfoil1	4253	12289	6.31	1.80	1.82	1.82	1.85	1.81	1.82	1.82	1.85	1.85	2.47	2.46	3.29
bintree10	1023	1022	4	1.12	1.14	1.14	1.17	1.12	1.14	1.14	1.17	1.13	1.63	1.48	2.62
c1y	828	1749	5.11	1.39	1.43	1.43	1.47	1.39	1.46	1.45	1.57	1.42	2.03	2.10	2.79
c2y	980	2102	5.07	1.38	1.43	1.43	1.48	1.38	1.45	1.43	1.52	1.41	2.00	2.06	2.76
c3y	1327	2844	5.24	1.40	1.44	1.44	1.47	1.40	1.46	1.46	1.55	1.48	2.12	2.18	3.42
c4y	1366	2915	5.17	1.42	1.46	1.46	1.48	1.42	1.48	1.48	1.54	1.43	2.02	2.03	3.48
c5y	1202	2557	5.09	1.39	1.44	1.44	1.48	1.39	1.46	1.47	1.51	1.47	2.08	2.14	2.91
crack	10240	30380	6.38	1.78	1.79	1.79	1.80	1.78	1.79	1.79	1.80	1.81	2.40	2.37	3.31
gd95c	62	144	5.60	1.51	1.67	1.67	1.81	1.54	1.67	1.67	1.81	1.65	2.25	2.18	2.99
gd96a	1076	4676	5.69	1.31	1.38	1.38	1.43	1.31	1.38	1.38	1.44	1.32	2.14	2.16	3.59
gd96b	111	193	3.21	1.01	1.06	1.06	1.25	1.01	1.06	1.05	1.26	1.01	1.45	1.56	2.80
gd96c	65	125	4.37	1.24	1.40	1.41	1.54	1.28	1.41	1.42	1.54	1.29	1.83	1.86	2.51
gd96d	180	228	4.11	1.13	1.22	1.20	1.42	1.13	1.24	1.21	1.45	1.19	1.71	1.79	2.45
hc10	1024	5120	7.09	1.68	1.86	1.88	1.95	1.68	1.86	1.88	1.96	1.93	2.69	2.70	3.40
mesh33x33	1089	2112	4.80	1.26	1.30	1.30	1.35	1.28	1.30	1.30	2.17	2.23	3.06	3.06	4.14
randomA1	1000	4974	7.99	2.06	2.10	2.10	2.16	2.06	2.11	2.11	2.17	2.23	3.06	3.06	4.14
randomA2	1000	24738	19.37	-	-	-	-	5.18	5.34	5.34	5.44	5.92	6.77	6.83	7.50
ramdomA3	1000	49820	31.37	-	-	-	-	8.71	8.87	8.86	9.13	10.21	10.85	10.80	11.71
randomA4	1000	8177	10.07	2.63	2.69	2.69	2.76	2.63	2.70	2.70	2.77	2.93	3.76	3.76	4.70
randomG4	1000	8173	16.11	-	-	-	-	4.02	4.10	4.10	4.19	4.63	5.67	5.69	6.64
small	5	8	3	1.25	1.25	1.25	1.25	1.25	1.25	1.25	1.25	1.25	1.36	1.25	1.5
whitaker3	9800	28989	6.62	1.82	1.83	1.83	1.85	1.82	1.83	1.83	1.85	1.84	2.56	2.57	3.37

Table 6.2: MaxInSet results for binomial random graphs

For each graph, its name, number of nodes, number of edges, edge probability, generation time, number of colours and the cost for each algorithm.

Instance	$\|V\|$	$\|E\|$	edge prob.	gen. time	colours	Best of Two	Ran. Selection
gr-1000-84458055	1000	4946	0.0100	0.0247	8	10451	15320
gr-1000-97953427	1000	4826	0.0100	0.0211	9	10137	14103
gr-1000-256996916	1000	5131	0.0100	0.0249	8	11064	15804
gr-1000-360678409	1000	4932	0.0100	0. 0086	8	10208	15016
gr-1000-671420410	1000	5063	0.0100	0.0077	8	10804	15396
gr-5000-84458055	5000	31583	0.0025	0.0833	10	75221	119302
gr-5000-97953427	5000	31376	0.0025	0.1678	9	74479	93629
gr-5000-256996916	5000	31325	0.0025	0.0836	9	73757	93685
gr-5000-360678409	5000	31424	0.0025	0.1224	10	73970	118704
gr-5000-671420410	5000	31478	0.0025	0.1287	9	74193	92976
gr-10000-84458055	10000	65292	0.0013	0.3201	10	156498	247541
gr-10000-97953427	10000	64919	0.0013	0.3516	10	156066	247469
gr-10000-256996916	10000	65254	0.0013	0.3121	9	156059	194531
gr-10000-360678409	10000	65049	0.0013	0.5234	10	158097	248806
gr-10000-671420410	10000	65052	0.0013	0.69565	10	156097	248767
gr-50000-84458055	50000	374759	0.0003	8.7018	10	968394	1430492
gr-50000-97953427	50000	374834	0.0003	8.7761	10	971552	1428691
gr-50000-256996916	50000	375072	0.0003	8.2551	11	970780	1649089
gr-50000-360678409	50000	374904	0.0003	8.5091	10	968703	1431465
gr-50000-671420410	50000	374802	0.0003	8.3180	10	968020	1428737
gr-100000-84458055	100000	1000917	0.0002	23.5637	12	3028277	3305514
gr-100000-97953427	100000	999643	0.0002	31.9860	12	3019992	3299358
gr-100000-256996916	100000	999734	0.0002	32.1853	13	3024724	5171483
gr-100000-360678409	100000	1000601	0.0002	31.7868	12	3035192	3312888
gr-100000-671420410	100000	999990	0.0002	32.1849	12	3022984	3302291

Table 6.3: MaxInSet results for random geometric graphs

For each graph, its name, number of nodes, number of edges, edge between cliques probability, generation time, number of colours and the cost for each algorithm.

Instance	$\|V\|$	$\|E\|$	radius	gen. time	colours	Best of Two	Ran. Selection
rg-gr-1000-84458055	1000	4660	0.0998	0.0039	13	13106	24291
rg-gr-1000-97953427	1000	4875	0.0998	0.0039	14	14318	22602
rg-gr-1000-256996916	1000	4770	0.0998	0.0038	12	13964	15584
rg-gr-1000-360678409	1000	4829	0.0998	0.0038	12	15153	15748
rg-gr-1000-671420410	1000	4724	0.0998	0.0039	11	13620	20449
rg-gr-5000-84458055	5000	29968	0.0496	0.0722	20	139525	250993
rg-gr-5000-97953427	5000	29779	0.0496	0.0718	16	136207	179571
rg-gr-5000-256996916	5000	30115	0.0496	0.0719	17	139329	252021
rg-gr-5000-360678409	5000	30326	0.0496	0.0715	17	140831	252349
rg-gr-5000-671420410	5000	30222	0.0496	0.0718	17	151748	250981
rg-gr-10000-84458055	10000	65107	0.0365	0.2646	19	150567	282148
rg-gr-10000-97953427	10000	65292	0.0365	0.2632	19	155319	228440
rg-gr-10000-256996916	10000	65285	0.0365	0.2628	19	153764	281880
rg-gr-10000-360678409	10000	65611	0.0365	0.2662	18	151899	282815
rg-gr-10000-671420410	10000	65413	0.0365	0.2643	20	158282	228988
rg-gr-50000-84458055	50000	387828	0.0177	2.8574	20	1845238	2536157
rg-gr-50000-97953427	50000	386336	0.0177	2.1257	19	1523456	2354125
rg-gr-50000-256996916	50000	387865	0.0177	2.6985	20	1453215	2153548
rg-gr-50000-360678409	50000	387382	0.0177	2.3154	18	1523456	2253458
rg-gr-50000-671420410	50000	387836	0.0177	2.5478	19	1654857	2156785
rg-gr-100000-84458055	100000	827038	0.0129	11.2135	20	3545625	4523486
rg-gr-100000-97953427	100000	824534	0.0129	11.2578	20	3412345	4896215
rg-gr-100000-256996916	100000	826747	0.0129	11.5486	25	3315648	4153125
rg-gr-100000-360678409	100000	825219	0.0129	11.4565	22	3412345	4351257
rg-gr-100000-671420410	100000	827318	0.0129	11.6317	21	3215348	4123578

Table 6.4: MaxInSet results for graphs with cliques

For each graph, its name, number of nodes, number of edges, edge between cliques probability, generation time, number of colours and the cost for each algorithm.

Instance	$\|V\|$	$\|E\|$	edge prob.	gen. time	colours	Best of Two	Ran. Selection
qg-gr-1000-84458055	1000	7817	0.664	0.0020	12	28811	29017
qg-gr-1000-97953427	1000	7812	0.664	0.0011	13	28729	39210
qg-gr-1000-256996916	1000	7746	0.664	0.0021	12	28519	28833
qg-gr-1000-360678409	1000	7752	0.664	0.0012	12	28378	28865
qg-gr-1000-671420410	1000	7834	0.664	0.0020	12	28646	29043
qg-gr-5000-84458055	5000	24719	0.0179	0.0035	11	90230	99566
qg-gr-5000-97953427	5000	24639	0.0179	0.0075	11	90026	99239
qg-gr-5000-256996916	5000	24737	0.0179	0.0036	11	90259	99631
qg-gr-5000-360678409	5000	24719	0.0179	0.0034	11	90274	99662
qg-gr-5000-671420410	5000	24752	0.0179	0.0039	11	90316	99725
qg-gr-10000-84458055	10000	951216	0.0100	0.4625	14	3579441	4593615
qg-gr-10000-97953427	10000	949535	0.0100	0.4879	14	3575185	4589858
qg-gr-10000-256996916	10000	950836	0.0100	0.4794	14	3576738	4597123
qg-gr-10000-360678409	10000	950247	0.0100	0.4488	14	3579240	4594594
qg-gr-10000-671420410	10000	949938	0.0100	0.9051	14	3576080	4593341
qg-gr-50000-84458055	50000	5379451	0.0025	9.4540	15	20685650	28466811
qg-gr-50000-97953427	50000	5375529	0.0025	9.4550	15	20672439	28446009
qg-gr-50000-256996916	50000	5374279	0.0025	9.4276	15	20642916	28431645
qg-gr-50000-360678409	50000	5375581	0.0025	9.07404	15	20667995	28436338
qg-gr-50000-671420410	50000	5377553	0.0025	9.2815	15	20651917	28442531
qg-gr-100000-84458055	100000	11002715	0.0013	35.2881	15	42471204	58332900
qg-gr-100000-97953427	100000	11001280	0.0013	34.5547	15	42480906	58298073
qg-gr-100000-256996916	100000	10998321	0.0013	34.9271	15	42489644	58292469
qg-gr-100000-360678409	100000	10998015	0.0013	34.3704	15	42477975	58303054
qg-gr-100000-671420410	100000	11001453	0.0013	34.8257	16	42484368	54693654

Table 6.5: MaxInSet results for random outerplanar graphs

For each graph, its name, number of nodes, number of edges, generation time, number of colours and the cost for each algorithm.

Instance	$\|V\|$	$\|E\|$	gen. time	colours	Best of Two	Ran. Selection
ro-gr-1000-84458055	1000	1455	0.1409	5	1788	2454
ro-gr-1000-97953427	1000	1391	0.1409	4	1627	2117
ro-gr-1000-256996916	1000	1549	0.4785	5	1978	2516
ro-gr-1000-360678409	1000	1398	0.4785	5	1679	2348
ro-gr-1000-671420410	1000	1416	0.4785	4	1679	2145
ro-gr-5000-84458055	5000	7189	0.4792	4	8640	11274
ro-gr-5000-97953427	5000	7237	0.4790	5	8831	12065
ro-gr-5000-256996916	5000	7263	0.03875	5	8806	11979
ro-gr-5000-360678409	5000	7273	0.0814	5	8799	12157
ro-gr-5000-671420410	5000	7381	0.0813	5	9108	12042
ro-gr-10000-84458055	10000	14108	0.0567	5	16890	23514
ro-gr-10000-97953427	10000	14331	0.0829	5	17274	24012
ro-gr-10000-256996916	10000	14589	0.0827	5	17965	24365
ro-gr-10000-360678409	10000	14289	0.0019	5	17278	23807
ro-gr-10000-671420410	10000	14364	0.0019	5	17284	23690
ro-gr-50000-84458055	50000	71633	0.0094	5	86428	119196
ro-gr-50000-97953427	50000	71629	0.0085	5	86556	19384
ro-gr-50000-256996916	50000	69876	0.0090	5	82834	116995
ro-gr-50000-360678409	50000	69687	0.0083	5	82470	116731
ro-gr-50000-671420410	50000	70576	0.0094	5	84112	118108
ro-gr-100000-84458055	100000	140777	0.0169	5	167864	235135
ro-gr-100000-97953427	100000	140496	0.0165	5	167397	236146
ro-gr-100000-256996916	100000	138842	0.0354	5	164165	232909
ro-gr-100000-360678409	100000	141429	0.0402	5	169488	237053
ro-gr-100000-671420410	100000	141733	0.0347	5	170181	238033

Table 6.6: MaxInSet results for k-trees

For each graph, its name, number of nodes, number of edges, generation time, number of colours and the cost for each algorithm.

Instance	$\lvert V\rvert$	$\lvert E\rvert$	gen. time	colours	Best of Two	Ran. Selection
kt-gr-1000-84458055	1000	8955	0.0136	10	22270	34391
kt-gr-1000-97953427	1000	8955	0.009	10	21959	34252
kt-gr-1000-256996916	1000	8955	0.013	10	22325	34421
kt-gr-1000-360678409	1000	8955	0.009	10	22401	34595
kt-gr-1000-671420410	1000	8955	0.009	10	22274	34508
kt-gr-5000-84458055	5000	41355	0.158	10	102540	159408
kt-gr-5000-97953427	5000	41355	0.242	10	102398	158473
kt-gr-5000-256996916	5000	41355	0.158	10	103121	159918
kt-gr-5000-360678409	5000	41355	0.156	10	102607	159216
kt-gr-5000-671420410	5000	41355	0.156	10	102221	158121
kt-gr-10000-84458055	10000	89955	0.762	10	222544	345247
kt-gr-10000-97953427	10000	89955	0.760	10	222787	346192
kt-gr-10000-256996916	10000	89955	0.759	10	222941	346113
kt-gr-10000-360678409	10000	89955	0.734	10	221574	345841
kt-gr-10000-671420410	10000	89955	0.761	10	222688	346117
kt-gr-50000-84458055	50000	413955	16.321	10	1023496	1592955
kt-gr-50000-97953427	50000	413955	16.274	10	1020873	1593079
kt-gr-50000-256996916	50000	413955	16.319	10	1020295	1595441
kt-gr-50000-360678409	50000	413955	16.307	10	1020573	1593876
kt-gr-50000-671420410	50000	413955	16.275	10	1020770	1596638
kt-gr-100000-84458055	100000	899955	76.853	10	2222444	3463588
kt-gr-100000-97953427	100000	899955	75.984	10	2219163	3464968
kt-gr-100000-256996916	100000	899955	76.646	10	2219150	3466608
kt-gr-100000-360678409	100000	899955	75.935	10	2219478	3467534
kt-gr-100000-671420410	100000	899955	76.463	10	2219774	3469370

Table 6.7: MaxInSet results for almost 3-complete graphs

For each graph, its name, number of nodes, number of edges, generation time, number of colours and the cost for each algorithm.

Instance	$\|V\|$	$\|E\|$	gen. time	colours	Best of Two	Ran. Selection
tk-gr-90-84458055	90	2706	0.001	6	4086	8826
tk-gr-90-97953427	90	2706	0.001	6	4086	8826
tk-gr-90-256996916	90	2706	0.001	6	4086	8826
tk-gr-90-360678409	90	2706	0.001	6	4086	8826
tk-gr-90-671420410	90	2706	0.001	6	4086	8826
tk-gr-450-84458055	450	67512	0.007	6	94796	222912
tk-gr-450-97953427	450	67512	0.008	6	94796	222912
tk-gr-450-256996916	450	67512	0.008	6	94796	222912
tk-gr-450-360678409	450	67512	0.013	6	94796	222912
tk-gr-450-671420410	450	67512	0.008	6	94796	222912
tk-gr-900-84458055	900	270021	0.052	6	376751	892221
tk-gr-900-97953427	900	270021	0.030	6	376751	892221
tk-gr-900-256996916	900	270021	0.052	6	376751	892221
tk-gr-900-360678409	900	270021	0.031	6	376751	892221
tk-gr-900-671420410	900	270021	0.031	6	376751	892221
tk-gr-4500-84458055	4500	6750048	0.728	6	9191600	22407048
tk-gr-4500-97953427	4500	6750048	0.734	6	9191600	22407048
tk-gr-4500-256996916	4500	6750048	0.812	6	9191600	22407048
tk-gr-4500-360678409	4500	6750048	0.733	6	9191600	22407048
tk-gr-4500-671420410	4500	6750048	0.727	6	9191600	22407048
tk-gr-9000-84458055	9000	27000093	2.975	6	36742295	89634093
tk-gr-9000-97953427	9000	27000093	3.117	6	36742295	89634093
tk-gr-9000-256996916	9000	27000093	2.974	6	36742295	89634093
tk-gr-9000-360678409	9000	27000093	2.920	6	36742295	89634093
tk-gr-9000-671420410	9000	27000093	2.968	6	36742295	89634093

Chapter 7

Conclusions

In this project, we have undertaken the task of continuing the study for the minimum linear colouring arrangement problem first studied by Isaac Sánchez in his bachelor's thesis, see [14].

First of all, we have studied the problem from a theoretical point of view. We have broadened the results for particular graphs with graph families such as k-trees, complete balanced k-partite graphs and bounded treewidth graphs. Related to the latter, we have been able to prove that the MinLCA problem with a fixed number of colours and bounded treewidth is in the class FPT. We also have been able to disprove the conjecture stated in [14] that the optimal value for the MinLCA problem can be obtained using the chromatic number of the graph. We have done so by using a counterexample with two complete balanced 3-partite graphs.

Secondly, we have made an experimental study of the problem developing different algorithms and testing them with various instances of graph families. We have devised an exact algorithm based on backtracking and three similar heuristic algorithms based on a maximal independent set approach. We have tested these algorithms with some instances used in the MinLA benchmarking (see Table 4.1), several graph families used in the previous MinLCA study, such as binomial random graphs, random geometric graphs, graphs with cliques and outerplanar graphs and also with some new instances which come from the families of k-trees and almost 3-complete graphs.
Although these experiments have given poor results comparing them with the results from the previous study of the MinLCA, we have been able to determine the best approach for the maximal independent set algorithm and for which families of graphs our algorithms have a better behaviour.
Finally, we make these results publicly available allowing future work for the problem.

A computationally hard problem such as the minimum linear colouring arrangement problem can be studied from many points of view. This offers a great range of options from which to develop future work for the problem.
On the one hand, continuing with the experimental studies of the problem, the backtracking algorithm could be adapted in order to find exact algorithms for particular graphs with a feasible computational cost or new approximation algorithms could be developed for the general case. In addition, the algorithm for bounded treewidth graphs studied in Subsection 3.4.3 could be implemented and tested for some families of graphs with bounded treewidth.
On the other hand, from a theoretical point of view, the parameterised version of the algorithm still has many paths to explore. Regarding this, a question about the problem parameterised by the treewidth of the graph has been raised. Since we have found that the MinLCA problem for a fixed number of colours and bounded treewidth is in FPT, and we know that the MinLA

problem with bounded treewidth is $W[1]$-hard, the question of whether the MinLCA problem with a bounded treewidth is in FPT or $W[1]$-hard remains as an open problem and we leave it for future studies of the MinLCA.

Open problem. Finding whether the minimum linear colouring arrangement problem parameterised by the treewidth of the input graph is in FPT or W[1]-hard.

We have also noticed looking at the results for outerplanar graphs that there seems to be a better bound for the MinLCA for these family of graphs. Indeed in [14], a conjecture was made about graphs with chromatic number 3 and our experimental results seem to reinforce this conjecture. Even so, we have not proven that these is in fact true, so it remains as a conjecture.

Conjecture. *If G is a graph and $\chi(G) = 3$, then*

$$MinLCA(G) \leq \frac{3}{2}|E(G)|$$

In conclusion, in this project we have continued the previous study for the minimum linear colouring arrangement problem. We have given various new theoretical results and developed, implemented and tested some algorithms with different techniques. Finally we make everything available in an open-access platform for future studies of the problem, which still has many paths to explore.

Bibliography

[1] Bodlaender, H. L. (1988), *Dynamic programming on graphs with bounded treewidth*, Proc. 15th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, 317, Springer-Verlag, pp. 108-118, doi: 10.1007/3-540-19488-6_110

[2] COIN-OR Foundation, Inc. (2016). *COIN-OR project*. https://www.coin-or.org.

[3] Díaz, J., Petit, J. and Serna, M. (2002). *A survey of graph layout problems*, ACM Computing Surveys, vol. 34, no. 3, pp. 313–356, issn: 03600300. doi: 10.1145/568522.568523.

[4] Diestel, R. (2006). *Graph Theory. Third ed.*, Springer-Verlag, Germany.

[5] Egerváry Research Group on Combinatorial Optimization. (2014). *Library for Efficient Modelling and Optimization in Networks (LEMON Graph Library)*, COIN-OR initiative. Available: http://lemon.cs.elte.hu/trac/lemon.

[6] Fellows, M. R., Hermelin, D., Rosamond, F., Shachnai, H. (2016). *Tractable Parameterizations for the Minimum Linear Arrangement Problem*, Journal ACM Transaction on Computation Theory, United States of America.

[7] Knuth D. E. (1968), *The Art of Computer Programming*, Addison-Wesley, United States of America.

[8] Niedermeier, R. (2006), *Invitation to fixed-parameter algorithms*, Oxfor University Press, United States of America. ISBN: 0198566077

[9] Petit, J. (2011), *Addenda to the Survey of Layout Problems*, Bulletin of the European Association for Theoretical Computer Science, no. 105, pp. 177-201. issn: 0252-9742. http://www.eatcs.org/beatcs/index.php/beatcs/article/ view/98

[10] Petit, J. (2003). *Combining spectral sequencing and parallel simulated annealing for the MinLA problem*, Parallel Processing Letters, vol.13, no. 1, pp.77-91. issn: 0129-6264. doi: 10.1142/S0129626403001161

[11] Petit, J. (2003). *Experiments on the minimum linear arrangement problem*, Journal of Experimental Algorithms, vol. 8, no. 2.3. issn: 1086654. doi: 10.1145/996546.996554

[12] de Ridder, H. N. et al. (2016). *Graph Class: Outerplanar*, Information System on Graph Classes and their Inclusions (ISGCI). Available: http://www.graphclasses.org/classes/gc_110.html

[13] *RDlab website*, Computer Science Department, Universitat Politècnica de Catalunya. Available: http://rdlab.cs.upc.edu/index.php/en/

[14] Sánchez, I. (2015). *Algorithms for the linear colouring arrangement problem*, Bachelor's thesis. upcommons.upc.edu/bitstream/handle/2099.1/25205/104713.pdf.

[15] Sánchez, I. (2015). *The minimum linear colouring arrangement problem*, Companion benchmarking platform for the project [14]. http://webit.cs.upc.edu/minlca/

[16] Schöning, U. (1987). *Graph isomorphism is in the low hierarchy*, Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, pp. 114–124, Springer-Verlag, United Kingdom.

[17] Sedgewick, R. (1988). *Algorithms*, Addison-Wesley Publishing Company, Inc., United States of America

[18] Sipser, M. (2006). *Introduction to the Theory of Computation. Second edition*, Thompson Course Technology, United States of America.

[19] Wikipedia (2017). *Four color theorem*, Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Four_color_theorem

[20] Wikipedia (2016). *Graph homomorphism*, Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Graph_homomorphism.