

BACHELOR

Analysis of an algorithm for finding perfect matching in k -regular bipartite graphs

Blom, Danny A.M.P.

Award date:
2017

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Analysis of an algorithm for finding perfect matchings in k -regular bipartite graphs

Author:

D.A.M.P. (Danny) BLOM

ID: 0902970

d.a.m.p.blom@student.tue.nl

Supervisor:

dr. R.A. (Rudi) PENDAVINGH

July 31, 2017

Abstract

This report gives a thorough analysis on an $O(km)$ -algorithm that is used to find perfect matchings and minimum edge colorings in bipartite multigraphs with degree k and number of edges m . First, the reader is given some background knowledge needed for a good understanding of the proposed methods. Several constructive graph algorithms will be mentioned and analysed. Available data structures are named and graph algorithms will be implemented as efficient as possible in terms of running time complexity. Using one of the available methods from the literature, we propose two new algorithms that have complexity $O(m \log n)$. Empirical analysis on several graph algorithms and subroutines is conducted, using a certain subset of regular bipartite graphs.

Contents

| | |
|--|-----------|
| Nomenclature | 3 |
| 1 Introduction | 4 |
| 2 Theoretical background | 5 |
| 2.1 Preliminaries | 5 |
| 2.2 Time complexity | 7 |
| 2.3 Theorems for understanding | 8 |
| 3 Schrijver's algorithm for matching and coloring regular bipartite graphs | 11 |
| 3.1 Reduction to k -regular bipartite graphs | 11 |
| 3.2 Schrijver's linear time perfect matching algorithm for fixed k | 12 |
| 3.2.1 Explanation and time complexity of the algorithm | 12 |
| 3.3 Equivalence of complexity bounds for perfect matchings and k -edge colorings | 16 |
| 3.3.1 Eulerian orientations in bipartite graphs | 16 |
| 4 Literature review | 19 |
| 4.1 Using prime factorisation to edge color bipartite graphs | 19 |
| 4.2 Adding perfect matchings | 22 |
| 4.3 Strategy for finding minimal edge colorings in k -regular bipartite graphs | 23 |
| 5 Explanation of JAVA model | 27 |
| 5.1 JAVA implementation of Schrijver's $O(km)$ algorithm | 27 |
| 5.2 Use of data structures | 27 |
| 5.3 Complexity of elementary operations | 28 |
| 5.4 Graph concepts for Schrijver's algorithm | 29 |
| 5.4.1 Attributes of the graph's subobjects EDGE and VERTEX | 30 |
| 5.5 Methods for performing simulation of Schrijver's algorithm | 30 |
| 5.5.1 Choosing edges in the circuit-finding path | 30 |

| | | |
|----------|--|-----------|
| 5.5.2 | Performing Euler split regular bipartite graphs of even degree | 31 |
| 5.5.3 | Generating random regular bipartite graphs | 31 |
| 5.6 | Methods for the algorithm based on Alon's | 32 |
| 5.6.1 | Generating and adding random 'dummy' matchings | 32 |
| 6 | Research | 34 |
| 6.1 | Improving Alon's algorithm | 34 |
| 6.2 | Improving the $O(m \log n)$ method | 35 |
| 7 | Results | 38 |
| 7.1 | Empirical results of simulations on the algorithm of Schrijver | 38 |
| 7.1.1 | Results of data of the subsets with incrementing degree and fixed number of vertices | 40 |
| 7.1.2 | Results of data of the subsets with incrementing number of vertices and fixed degree | 43 |
| 7.1.3 | Results of data of the subsets with random degree and fixed number of vertices | 46 |
| 7.1.4 | Results of data of the subsets with random number of vertices and fixed degree | 49 |
| 7.1.5 | Summary | 51 |
| 7.2 | Empirical results of Alon's and new algorithms | 51 |
| 8 | Conclusion | 55 |

Nomenclature

$d(v)$ the degree of vertex v

E the set of edges

V the set of vertices

$\chi'(G)$ edge chromatic number

$\Delta(G)$ The maximum of the degrees of the vertices in a graph G

$\delta(v)$ the set of edges incident to vertex v

$\Gamma(X)$ the set of vertices that are adjacent to a vertex $v \in V$

$d_G(v)$ the degree of vertex u with respect to graph G

Chapter 1

Introduction

In a wide range of applications, members of two distinct sets A and B of objects need to be matched to each other in such a way that each object from set A is connected to only one specific object from set B .

A famous example of this was first mentioned by Philip Hall [6]. Imagine that there is a set A of women and a set B of men, with $|A| = |B| = n$ for some natural number n . Prior to a party, each woman and each man had to create a list of names of members of the other gender that they prefer to dance with. The organiser of the party wanted to match as many men and women to each other while complying to each person's preferences. This problem could be modelled by constructing a graph of men and women, and including edges between a man and a woman when they prefer to dance with each other.

Other problems revolve around repetitively matching members of two distinct sets such that two objects are never matched twice or more. For example, this problem arises when we want to find a fixture schedule in round-robin tournaments with a home-away structure such as national football leagues. In this case, each team appears once in both sets. Namely, the first set, called A , consisting of the home teams and the second set, called B , consists of the away playing teams. Matching the teams once means that we have obtained two rounds of the league and in each matching, teams that are matched play a football match that should not occur more than once. When one wants to obtain a fixture schedule for a round-robin tournament, all he has to do is produce the graph of home teams and away teams and connect all the teams that have to play against each other (all teams are connected to all other teams in the beginning). When a matching is found in this graph, the matches implied by the matching are played in the same weekend and the matching is removed from the graph. Then iteratively, find a round for the next weekend until all matches are played.

Mathematical methods for solving problems as described above are described and defined in this report. However, what we want to answer this paper is not how we obtain the solutions to those problems, but how fast we can obtain it. The definition of fast in this context is not the same as in plain English, as we will learn later.

Chapter 2

Theoretical background

2.1 Preliminaries

In this report, we will make extensive use of the mathematical field of graph theory and the properties of its most frequently used concepts. For this reason, this section introduces these concepts in a mathematical manner, in order to eliminate any confusion that may arise to the reader. We assume that the reader has basic knowledge of set theory.

The most elementary objects that we are going to work with are graphs. More formally:

Definition 2.1. (Graph)

*An undirected **graph** is an ordered tuple $G = (V, E)$ of a set of vertices V and a set of edges E , such that each element of E is a unordered pair of distinct elements of V . An edge $e \in E$ is called **incident** to a vertex $v \in V$ if $v \in e$ and vertices $u, v \in V$ are called **adjacent** or **neighbours** if there exists an edge $e \in E$ with $e = \{u, v\}$. A **multigraph** is a graph such that each edge is allowed two or more times in the set of edges.*

From now on, when we speak of graph algorithms in this report, the input of these algorithms are multigraphs. Note that "normal" graphs are a special subset of multigraphs, therefore increasing the generality of algorithms. This report is not focused on graphs in general, but our special interest goes to inventing and applying algorithms on bipartite multigraphs.

Definition 2.2. (Bipartite graph)

Let $G = (V, E)$ be a graph. A bipartition of the vertices V is a partition into subsets V_1 and V_2 such that:

$$\forall e \in E : e = \{u, v\} \text{ for some } u \in V_1, v \in V_2$$

*A graph $G = (V, E)$ is **bipartite** if there exists a bipartition of V .*

This means that for each edge $e \in E$, one vertex of e is an element of V_1 , while the other is in V_2 . From now on, a bipartite graph $G = (V, E)$ will also be denoted by $G = (V_1 \cup V_2, E)$. As we will prove later on in section 3.1 in the literature review section, theorems on the worst case performance of the graph algorithms that we are interested in only need to be proven for regular bipartite graphs. A measure of performance of algorithms will be formalised in the next chapter. To be able to introduce the definition of regular bipartite graphs, we first need to formally define the number of edges that are adjacent to a vertex:

Definition 2.3. (Degree)

Let $G = (V, E)$ be a graph and let $v \in V$. The **star** of v is defined as $\delta(v) := \{e \in E : v \in e\}$. The **degree** of vertex v is $d(v) := |\delta(v)|$. Furthermore, the **maximum degree** of G is denoted by $\Delta(G) := \max_{v \in V} d(v)$.

In more intuitive terms, this means that the degree of a vertex is the number of edges that are incident to v . Regularity of graphs can now be defined as follows:

Definition 2.4. (k -regular graph)

A graph $G = (V, E)$ is **k -regular** if:

$$\forall v \in V : d(v) = k$$

This definition implies that in a k -regular bipartite graph $G = (V_1 \cup V_2, E)$, the subsets of the bipartition have equal cardinality:

Lemma 2.5. For a k -regular bipartite graph $G = (V_1 \cup V_2, E)$, we have that $|V_1| = |V_2|$

Proof. Let $G = (V_1 \cup V_2, E)$ be a k -regular bipartite graph. Then:

$$k|V_1| = \sum_{v \in V_1} d(v) = \sum_{v \in V_2} d(v) = k|V_2|$$

Dividing both sides by $k \neq 0$ gives the statement. □

$$\sum_{v \in V_1} d(v) = \sum_{v \in V_2} d(v) \Leftrightarrow |V_1|k = |V_2|k \Leftrightarrow |V_1| = |V_2|$$

Furthermore, the concept of a circuit and the concept of the absence of circuits will be used in the description of the algorithm on which this paper is based, the definitions of these concepts will be given as well:

Definition 2.6. (Walk, path, circuit)

In a graph $G = (V, E)$, a **walk** from vertex v_0 to v_k is defined as a sequence of vertices (v_0, v_1, \dots, v_k) , with $v_i \in V$ for $i = 0, 1, \dots, k$, such that $\{v_{i-1}, v_i\} \in E$ for $i = 1, 2, \dots, k$.

A **path** from v_0 to v_k is a walk such that $v_i \neq v_j$ for $i, j = 0, 1, \dots, k, i \neq j$.

A **circuit** is a walk (v_0, v_1, \dots, v_k) such that $v_i \neq v_j$ for $i, j = 1, 2, \dots, k, i \neq j$ and such that $v_0 = v_k$. The **length** of a circuit is equal to k .

Definition 2.7. (Forest)

Let $G = (V, E)$ be a graph. Then G is a **forest** if G contains no circuits.

If in a graph $G = (V, E)$ it holds that for any $v_1, v_2 \in V$, there exists a path from v_1 to v_2 , then G is called a **connected** graph. Moreover, note that the definitions of bipartite graphs and circuits lead to the somewhat intuitive observation that a graph G is bipartite if and only if G contains no circuits of odd length, where circuit length is measured as the number of edges in the circuit. This result was first proven by Hungarian graph theoretician Dénes König in 1936.

Another goal of the algorithm that will be analysed in the following chapters is solving a subproblem of edge coloring bipartite graphs. What this means in concrete terms, is given in the next definition:

Definition 2.8. (Edge coloring, edge chromatic number)

A ***k*-edge coloring** of a graph $G = (V, E)$ is a function $f : E \rightarrow \{1, 2, \dots, k\}$ such that:

$$f(e_1) \neq f(e_2) \text{ for any two adjacent edges } e_1, e_2$$

A graph $G = (V, E)$ is called ***k*-edge-colorable** if there exists a *k*-edge coloring of G .

The **edge chromatic number** of a graph $G = (V, E)$, denoted by $\chi'(G)$, is defined as:

$$\min\{k \in \mathbb{N} \mid \text{there exists a } k\text{-edge coloring of } G\}$$

In more intuitive terms, this means that the edge chromatic number $\chi'(G)$ of a graph G is equal to the number of colors minimally needed to obtain a valid coloring of G , i.e. satisfying the condition given in definition 2.8. In order to formulate edge coloring algorithms that are understandable and practical in use, we need to consider the subproblem of finding a perfect matching. Mathematically speaking, this means:

Definition 2.9. (Matching)

A **matching** in a graph $G = (V, E)$ is a subset $M \subseteq E$ such that:

$$\forall e, e' \in M, e \neq e' : e \cap e' = \emptyset$$

A matching M is **perfect** if it covers all vertices $v \in V$, that is:

$$\forall v \in V \exists e \in M : v \in e$$

Therefore, a perfect matching F in a *k*-regular bipartite graph $G = (V, E)$, always satisfies $|F| = \frac{1}{2}|V|$.

Apart from a technique for finding perfect matchings in regular bipartite graphs, a technique is introduced for partitioning graphs in smaller subgraphs. This technique uses the concept of Eulerian orientations:

Definition 2.10. (Eulerian orientation)

Let $G = (V, E)$ be an undirected graph. An **Eulerian orientation** of G is an assignment of a direction to each edge of G such that the number of edges directed into v (**indegree** of v) is equal to the number of edges directed away from v (**outdegree** of v) for all $v \in V$.

With the definitions given above, we have enough basic knowledge on the field of graph theory to understand the main algorithm to be analysed in this paper.

2.2 Time complexity

Now that we have introduced the necessary concepts for our research, we can verbalize our problem definition in more explicit formal terms. In the literature study, our focus will go out to the technical aspects of graph algorithms that are used for finding minimal edge colorings for bipartite graphs. When we change subroutines of an algorithm or even find completely different algorithms that also give the appropriate output, we want to have a measure of comparing these different algorithms. In complexity theory, this is done by asymptotic analysis or complexity analysis. Of course, research on certain kinds of graph algorithms is conducted in order to find faster algorithms or in

order to prove that certain existing algorithms cannot be improved.

The asymptotic running time complexity of an algorithm is a measure of the order of magnitude of the number of steps that need to be performed to successfully execute the algorithm. The time complexity is often given using the big-O notation, which yields an upper bound on the order of magnitude on the number of operations needed. This upper bound should be as sharp as possible, to not overestimate the order of magnitude of the number of operations, but big enough to ensure that it does not contain fewer operations than the worst-case scenario for the input. More formally, this notation can be defined as follows:

Definition 2.11. (Big-O notation)

Let $f(n)$ be the number of elementary operations needed to successfully execute an algorithm with input length n and let g be a function defined on a (not necessarily strict) subset of the natural numbers \mathbb{N} . For $n \rightarrow \infty$, we say:

$$f(n) = O(g(n))$$

if and only if there exists a positive constant $M > 0$ such that for sufficiently large n , say $n \geq n_0$ for $n_0 \in \mathbb{N}$, it holds that:

$$\exists M > 0 \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0: |f(n)| \leq M|g(n)|$$

The definition can be extended to functions with multiple variables, such that the statement holds if all the variables specified in the function are greater than a certain threshold, as we will see later in this report. We will see that this last observation is needed for examining the complexity of the algorithms we are interested in.

Throughout this report, as we are interested only in graph algorithms, we will make use of a few notations:

- Δ is the maximum degree in a bipartite graph (which equals k for k -regular bipartite graphs).
- m is the number of edges in a bipartite graph.
- n is the number of vertices in a bipartite graph (in both vertex classes combined, so $\frac{n}{2}$ vertices per vertex class).

In the context of graph algorithms, we refer to table 5.1 for the fundamental operations used in graph algorithms. When we say that the running time complexity of a graph algorithm equals $O(m)$, this means that the total number of fundamental operations needed to successfully execute the algorithm will (at most) grow linearly when m grows. This typically occurs in an algorithm where a certain fundamental operation has to be performed *for each edge* of the input graph, such as updating the weight function for each edge. Obviously, the number of fundamental operations increases when m increases.

2.3 Theorems for understanding

The previous section was useful for understanding the concept of complexity and comparison of different (graph) algorithms. Apart from this, we need to make sure that the output we desire can reasonably be expected from an algorithm, i.e. does a desired output actually exist for each valid

input. Therefore, we need a more formal definition of algorithms in order to fully understand what mathematical properties of k -regular bipartite graphs we need to answer this problem:

Definition 2.12. (Algorithm)

An algorithm (for a problem) is a method that, for each valid input instance of the problem, solves the problem. To qualify as a deterministic algorithm, the following four requirements need to be satisfied:

1. *Finiteness (the algorithm can be described in a limited number of instructions)*
2. *Runnability (the algorithm is mechanically executable)*
3. *Termination (the algorithm stops after a finite number of iterations)*
4. *Determinism (the algorithm will, given a particular input, always return the same output after passing through the same sequence of states.)*

Of those four requirements, showing the third requirement "termination" for all valid inputs is often the hardest part of proving that a method classifies as an algorithm. The other properties are more trivial when we read and try to interpret an algorithm. In our situation, showing that an algorithm terminates means that if we have a valid input, i.e. a k -regular bipartite graph for some $k \in \mathbb{N}$, in our algorithm, can we assure that the desired output exists in the first place? This is in fact equivalent to proving that a k -regular bipartite graph always has a k -edge coloring and a perfect matching. To prove this statement, we need to introduce a theorem by König [10]:

Theorem 2.13. (König)

Every bipartite graph $G = (V, E)$ has the following property:

$$\chi'(G) = \Delta(G)$$

Proof. Let $G = (V, E)$ be a bipartite graph. We have that $\chi'(G) \geq \Delta(G)$. Indeed, find a vertex $v \in V$ such that $d(v) = \Delta(G)$, then the edges that are incident to v all need to have a different color. Therefore, the smallest number of colors needed for a correct edge coloring is greater than or equal to $d(v) = \Delta(G)$.

What we have left to prove is: $\chi'(G) \leq \Delta(G)$

Let $\Delta(G) = k$. Then we prove the statement by induction on m :

Let $|E| \leq k$, then there trivially exists an k -edge coloring on G .

For the induction hypothesis, suppose now that the statement is true for all bipartite graphs with $|E| < m$, for some $m > k$.

Let now G be a bipartite graph on m edges. Consider the graph $G' = G \setminus \{e\}$ for some $e \in E$, then G' is still a bipartite graph with fewer edges. By the induction hypothesis, we have that G' has a k -edge coloring. Color its edges with k colors this gives a colouring of G except for edge e . Let now $e = \{u, v\}$ for some vertices $u, v \in V$ of the bipartite graph. Clearly, u and v are in different color classes. We have two cases to take into account, and prove the statement per case:

1. There is no edge adjacent to u with color i , where $1 \leq i \leq k$, and the same holds for v . Then color edge e with color i and the statement follows.

2. There is a color, say color 1, such that there is an edge with this color incident to u , but there is no edge with color 1 incident to v . Let $e' = \{u, w_1\}$ the edge with color 1 with endpoint u . As $d_{G'}(u) = d_G(u) - 1 \leq k - 1$, there is a color, without loss of generality color 2, such that there is no edge with color 2 incident to u . Possibly, there is an edge $\{w_1, w_2\}$ with, without loss of generality, color 2, $\{w_2, w_3\}$ with color 1, and so forth. Build a path of maximum length with edges alternately having color 1 and 2 starting from u . This path does not visit v , as this could only happen with an edge of color 2. But if that would be the case, then there exists a path of even length from u to v , a contradiction by definition of a bipartite graph. Now interchange the colors of the edges on the path found, so that each edge with color 1 now has color 2 and vice versa. It now happens to be that there is no edge incident to u with color 1, just like vertex v . Now color edge e with color 1 and the statement is proven.

□

The generality of this theorem makes its result very useful to use in algorithms for finding minimal edge colorings, as it holds for all bipartite graphs, so for k -regular bipartite graphs in particular. As our focus will go to k -regular bipartite graphs, we can conclude from this theorem that a minimal edge coloring of a k -regular bipartite graph can be realised by taking a set of k different colors.

We also have a theorem, whose proof contains an technical element due to Hall [6], that directly implies the existence of a perfect matching in a k -regular bipartite graph:

Theorem 2.14. *Every k -regular bipartite graph has a perfect matching*

Proof. Let $G = (V_1 \cup V_2, E)$ be a k -regular bipartite graph, where V_1 and V_2 are the two color classes of G . Let $X \subseteq V_1$ arbitrary. Put $\Gamma(X) = \{y \in V_2 \mid \exists x \in X : \{x, y\} \in E\}$. Then it immediately follows that $|\Gamma(X)| \geq |X|$, as otherwise, there exists a vertex $y' \in \Gamma(X)$ such that $d(y') > k$, contradicting the definition of G . Indeed, if $|\Gamma(X)| < |X|$, $\Gamma(X)$ is adjacent to at least $k|X|$ edges, making the average number of edges incident to a vertex in $\Gamma(X)$ greater than k . This means that by Hall's Marriage Theorem, as X was picked arbitrarily, there exists a matching that saturates V_1 (this is also called a **complete** matching), so there exists a perfect matching, as $|V_1| = |V_2|$ as shown before. □

We can use this theorem to conclude that for each input graph, we can retrieve a perfect matching when we have an appropriate algorithm to find it. Furthermore, it also leads us to observe that a k -regular bipartite graph could be split into k disjoint perfect matchings. If a perfect matching $F \subseteq E$ is found in a k -regular bipartite graph $G = (V, E)$, F could be removed from the graph to obtain the $(k - 1)$ -regular bipartite graph $G' = (V, E \setminus F)$. This argument could be used iteratively until the final graph is just a perfect matching.

The theory above was to show the existence of a valid output (a k -edge coloring) for each input (a k -regular bipartite graph) of the algorithm. The last observation leads to a subproblem related to edge coloring bipartite graphs, namely constructing an algorithm for finding perfect matchings in bipartite graphs that is as fast as possible, i.e. the asymptotic complexity bound of the algorithm is as sharp as possible. The following chapter will describe algorithms that find perfect matchings in bipartite graphs and provide an explanation on their running time complexities. Furthermore, it will be shown that it suffices to have an algorithm that finds a perfect matching for k -regular bipartite graphs only.

Chapter 3

Schrijver's algorithm for matching and coloring regular bipartite graphs

In this chapter, we will describe one of the most efficient algorithms known today in graph theory on the subject of perfect matchings in bipartite graphs and its practical use as a subroutine of edge-coloring algorithms for bipartite graphs. Summarised, the main problems that we want to solve are:

- Given: a bipartite graph $G = (V, E)$.
Find: a $\Delta(G)$ -edge coloring of G .
- Given: a k -regular bipartite graph $G = (V, E)$.
Find: a perfect matching F in G .

3.1 Reduction to k -regular bipartite graphs

Finding a $\Delta(G) = k$ -edge coloring for general bipartite graphs is equivalent to finding a k -edge coloring in a k -regular bipartite graph. The reason for this is that each general bipartite graph G with maximum degree $\Delta(G)$ can be easily transformed to a $\Delta(G)$ -regular bipartite graph G' by manipulating the original graph by adding edges and vertices such that it becomes regular. The procedure to do this is as follows:

Let $G = (V, E)$ be a bipartite graph. For each of the two vertex color classes, merge together two vertices if the sum of their degrees is smaller or equal to k until there is at most one vertex v left with $d(v) \leq \frac{1}{2}k$. This means that at most two vertices are left with degree smaller than or equal to $\frac{1}{2}k$. For other vertices $v \in V$, $k - d_G(v) \leq \frac{1}{2}k \leq d_G(v)$. Make a copy H' of the resulting graph $H = (V_H, E_H)$ and connect each v with its copy $v' \in H'$ with $k - d_H(v) = k - d'_H(v)$ edges. This yields a k -regular bipartite graph $\tilde{G} = (\tilde{V}, \tilde{E})$ with:

$$|\tilde{E}| = |E| + \sum_{v \in V_H} (k - d_H(v)) \leq |E| + \sum_{v \in V} (k - d_G(v)) \leq |E| + \sum_{v \in V} d_G(v) = |E| + 2|E| = 3|E|$$

so $|\tilde{E}| = O(|E|)$. As the number of edges of the original graph G and the resulting graph \tilde{G} are of the same order of magnitude, and the same holds for vertices as $|\tilde{V}| \leq 2|V| = O(|V|)$, edge color-

ing algorithms will have a similar worst-case complexity bound on both graphs. Because of this result, the rest of the paper will focus merely on k -regular bipartite graphs, as this kind of bipartite graphs has properties that allow clever techniques that decrease the running time complexity. These techniques will be treated more thoroughly further in this paper.

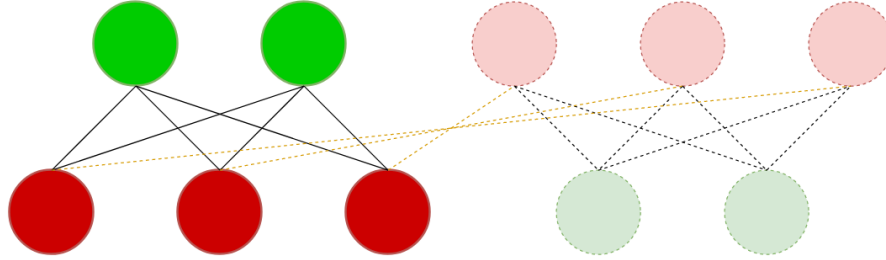


Figure 3.1: The bipartite graph $G = (V, E)$, given on the left by dark red and green vertices and solid edges, is copied to copy $G' = (V', E')$, and vertices $v \in V$ with non-maximal degree are connected with their copies $v' \in V'$ to make a 3-regular bipartite graph.

3.2 Schrijver's linear time perfect matching algorithm for fixed k

In this section, we will consider the article on which this report is based, *Bipartite Edge coloring in $O(\Delta m)$ time* written by Alexander Schrijver [11]. In contrast to earlier bipartite matching algorithms, such as the $O(n^{\frac{5}{2}})$ algorithm by Hopcroft-Karp [7] and the $O(m \log m)$ algorithm by Cole-Hopcroft (Alon [1] has elaborated a simpler algorithm with the same time bound, which will be mentioned in section 4.2) the algorithm by Schrijver was the first perfect matching algorithm that was linear with respect to the number of edges m for fixed values of Δ , namely $O(\Delta m)$. Using a divide-and-conquer technique due to Gabow [4], we will also derive that the time complexity of a perfect matching algorithm is equivalent to the time complexity of a minimal edge coloring algorithm.

3.2.1 Explanation and time complexity of the algorithm

The algorithm is given in the theorem below, immediately proving the time complexity bound of $O(km)$.

Theorem 3.1. *A perfect matching in a k -regular bipartite graph G can be found in $O(km)$ time, where m is the number of edges of G .*

Proof. (by construction)

Let $G = (V, E)$ be a k -regular bipartite graph on m edges. Let $w : E \rightarrow \mathbb{Z}^+$ be a weight function defined on the edges of G .

Put $E_w = \{e \in E \mid w(e) > 0\}$. Furthermore, for any matching $F \subseteq E$, let $w(F) = \sum_{e \in F} w(e)$. Lastly, we

denote the incidence vector of F by χ^F , where the entries of $\chi^F \in \{0, 1\}^E$ are such that

$$\chi^F(e) = \begin{cases} 1, & \text{for } e \in F \\ 0, & \text{for } e \notin F \end{cases}$$

Below is given the pseudo code of the algorithm by Alexander Schrijver, which is loosely based on an older matching algorithm with alternately adding and deleting edges due to Csima and Lovász [3] to find perfect matchings in k -regular bipartite graphs:

Algorithm 1 Finding a perfect matching in a k -regular bipartite graph

```

1: Input:  $k$ -regular bipartite graph  $G = (V, E)$ , weight function  $w : E \rightarrow \mathbb{Z}^+$ .
2: Output: perfect matching  $F \subseteq E$  in  $G$ .
3: for  $e \in E$  do
4:    $w(e) := 1$ ;
5: end for
6: while  $E_w$  is not a forest do
7:   Find a circuit  $C$  in  $G$ ;
8:   Define  $M, N$  matchings with  $C := M \cup N$ ,  $w(M) \geq w(N)$ ;
9:    $\alpha := \min_{e \in N} w(e)$ 
10:   $w := w + \alpha(\chi^M - \chi^N)$ ;
11: end while
12: return  $E_w$ ;
```

At first, the weight function is initialized and each edge is given weight value 1. When the graph limited to E_w is a forest, no circuits can be found anymore and this means the weights on the edges cannot be updated. This can only be the case when each edge in E_w has weight k , therefore implying a perfect matching is found. Furthermore, circuits in a bipartite graph are always of even length in terms of number of edges, so it is always possible to split a circuit in two matchings by traversing the circuit and alternately adding an edge to respectively the first and second matching.

Using the expression in line 10 instead of (3.1), one can assure that at least one of the edges that occurs in N will have zero weight after the update statement. As each vertex is visited at most once during the circuit and the two matchings in the circuit are chosen by alternately picking edges from the circuit, this means that all vertices visited in the circuit are incident to an edge in M and to an edge in N . This also means that $w(\delta(v)) = k$ is met for all vertices v and for each iteration of the algorithm, as the weight on the edge incident to v belonging to M increases by the same amount as the weight on the edge incident to v belonging to N decreases. Moreover, as in each iteration $|E_w|$ will decrease by at least 1, this shows us that algorithm 1 is indeed terminating. We next argue that the time complexity is $O(km)$, namely by noting that the sum of the squares of the weights on the edges has an upper bound, as for all edges in the graph it holds that $w(e) \leq k$. Before starting the first iteration, the sum of squares of weights is $m = \frac{1}{2}nk$, as each edge's weight is initialized at 1. The change in the sum of squares of weights in one iteration is:

$$\begin{aligned}
 \Delta(\sum_{e \in E} w(e)^2) &= \sum_{e \in E} w(e)_{new}^2 - \sum_{e \in E} w(e)_{old}^2 \\
 &= \sum_{e \in M} ((w(e) + \alpha)^2 - w(e)^2) + \sum_{e \in N} ((w(e) - \alpha)^2 - w(e)^2) \\
 &= \sum_{e \in M} (2\alpha w(e) + \alpha^2) + \sum_{e \in N} (-2\alpha w(e) + \alpha^2) \\
 &= 2\alpha (w(M) - w(N)) + \alpha^2 (|M| + |N|) \\
 &\geq \alpha^2 |C| \\
 &\geq |C|
 \end{aligned}$$

Note that in this calculation, Δ does not denote the maximum degree of a bipartite graph, but the change in the sum of squares of weights in a single iteration. Besides, the weight value of an edge e after the update statement given in (10) is denoted by $w(e)_{new}$, and $w(e)_{old}$ the weight value of edge e before. Note that the weights on the edges of E_w are always integer valued and positive, in other words $\alpha > 0, \alpha \in \mathbb{N}$, so that $\alpha \geq 1$, so this means that $\sum_{e \in E} w(e)^2$ will always increase by at least the number of edges in the circuit found.

Until now, no description on how to find a circuit in a bipartite graph is given. At the start of the execution, we begin with an empty path P , which will be filled with edges e such that $0 < w(e) < k$. Indeed, if we search for a circuit in E_w , this means that edges of weight zero are already eliminated and edges of weight k are element of the perfect matching to be found and not incident to other edges in E_w , so there is no possibility we will find a circuit that contains these maximum weight edges. We start at a random vertex v such that there is no edge of weight k incident to it. Then choose an edge $e = \{v, w\}$ and after that, keep visiting edges in E_w that are not visited earlier until a visited vertex is reached. Note that these edges are always of the desired weight between zero and k and the time complexity of finding a circuit is therefore $O(|C|)$. Now, apply the update statements of algorithm 1 on the edges of circuit $C \subset P$ and delete C from the path P . A next circuit, if it exists, will be found by using the part of P that was not deleted and doing the same iteration method as above.

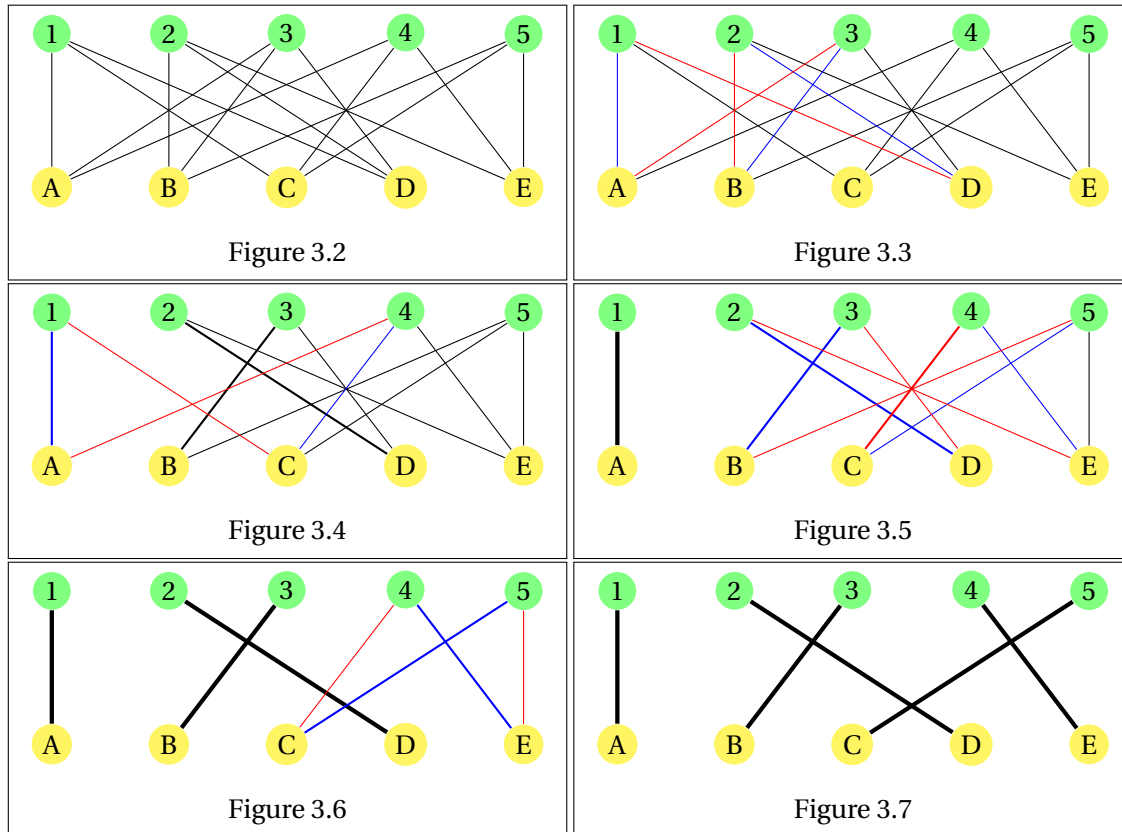
Proving the complexity of the algorithm can be done by looking at the sum of squares of weights on the edges. Let us denote this sum by $w(E)^2 = \sum_{e \in E} w(e)^2$. At the start of the algorithm, all edges' weights are set to one, so $w(E)^2 = \frac{1}{2}nk = m$. When we have found a perfect matching in the graph, we will have $w(E)^2 = \frac{1}{2}nk^2 = km$. Earlier, we observed that in each iteration, the increase in this sum of squares is at least the length of the circuit $|C|$. A circuit could be found in $O(|C|)$ time, as we have shown before. The difference between the start and the end of the algorithm in terms of the sum of squares is:

$$w(E)_{end}^2 - w(E)_{start}^2 = km - m = (k - 1)m$$

To find circuits with a total length of $(k - 1)m$ takes time $O((k - 1)m) = O(km)$. As the algorithm indeed terminating, this means that the algorithm can find a perfect matching in a k -regular bipartite graph in $O((k - 1)m) = O(km)$ time. \square

The execution of the algorithm is shown below in an example. The example 3-regular bipartite graph shown below in Figure 3.2- 3.7 is input of the above algorithm. The algorithm iteratively finds a circuit and divides it in two matchings, which are distinguished by color. Namely, M is given

by the blue-colored edges and N by the red-colored edges. The weight on an edge is proportional to the thickness of the edge, i.e. the thicker the edge, the higher the weight on it. When after an update statement, an edge has weight zero, it is deleted from the graph and not visible anymore.



Of course, choosing an unvisited edge to add to the circuit finding path occurs randomly in this interpretation. In the research part of this paper, we will analyse a few heuristics for choosing these edges and compare their performances on a randomly generated subset of regular bipartite graphs. These are introduced in a later part of the report, as the heuristics heavily rely on information on data structures for graphs that we have not mentioned yet.

Compared to the algorithm described by Schrijver in his paper, this implementation is slightly more efficient than Schrijver's because of the replacement of the update statement in his algorithm by the code in line 10. The original algorithm by Schrijver had an alternative update statement that required generally more iterations of the while-loop as the one given in line 10, namely:

$$w := w + \chi^M - \chi^N \quad (3.1)$$

In this implementation, the circuit's edges' weights are only either increased or decreased by one. Therefore, the algorithm will need more iterations to decrease the cardinality of E_w , indirectly implying that the implementation also needs more iterations in general.

This description and proof of the algorithm above raises the following questions, which we will try to answer in this report:

Question on worst-case running time (chapter 4):

- Can we develop an algorithm, whether or not related to the algorithm of Schrijver, that asymptotically performs better than Schrijver's algorithm, at least for a specific subclass of k -regular bipartite graphs?

Question on average-case running time (chapter 7):

- What can we say about the running time complexity of Schrijver's algorithm for finding perfect matchings in k -regular bipartite graphs in practice? (in theory, it is $O(km)$) (own research)

Questions on choices on the algorithm's implementation (chapter 5):

- How can we implement the algorithm in a programming language such that the theoretically promised complexity bound is attained?
- What is the most efficient strategy for finding circuits in bipartite graphs for the purpose of Schrijver's algorithm?

Questions on the interpretation of observed behaviour (chapter 7):

- How does the length of a circuit relate to the benefit (the increase in sum of squares of the edge's weights) gained from adjusting the weights of the circuit's edges?
- What is the behaviour (measured in number of traversed edges until the end of execution) of the algorithm when we vary the number of vertices n and the degree k ?

3.3 Equivalence of complexity bounds for perfect matchings and k -edge colorings

From the above time complexity bound of $O(km)$ of finding a perfect matching, one could derive a statement about the same bound for finding a k -edge coloring in a k -regular bipartite graph. A weaker statement for finding k -edge colorings in k -regular bipartite graph can be easily deduced from theorem 3.1, namely that it could be found in $O(k^2m)$ time:

Let $G = (U \cup V, E)$ be a k -regular bipartite graph with U and V the two color classes of vertices of G . Then, by theorem 2.14, there exists a perfect matching M , which could be found in $O(km)$ time. Deleting this perfect matching M from G leaves us the $(k-1)$ -regular bipartite graph $G' = G \setminus M$. Recursively finding perfect matchings and giving each matching another color yields a k -edge coloring of G with the complexity bound:

$$O(km + (k-1)m + (k-2)m + \dots + 2m + m) = O\left(m \sum_{i=1}^k i\right) = O\left(\frac{k(k+1)}{2}m\right) = O(k^2m)$$

3.3.1 Eulerian orientations in bipartite graphs

The result above however is not yet useful enough for our purposes and too slow for practice. Nevertheless, the result can be sped up by using a technique that was introduced by Harold Gabow [4].

His approach made use of a technical algorithm that finds an Eulerian orientation of the graph in the case that k is even. Trivially, this only works when each vertex has even degree. Therefore, we can only apply this method for algorithms when the input is a k -regular bipartite graph where k is an even number. Gabow's algorithm was based on general bipartite graphs, but as our specific interest goes to regular bipartite graphs, it is more relevant to adapt the algorithm for examining k -regular bipartite graphs. The pseudo code of his algorithm adapted to k -regular bipartite graphs is as follows:

Algorithm 2 Finding an Eulerian orientation in a k -regular bipartite graph, $k \neq 0$ even

```

1: Input:  $k$ -regular bipartite graph  $G = (V, E)$ .
2: Output: List of paths  $L$  of the edges  $E$ .
3: initialise empty list of paths  $L$ 
4: initialise empty queue of vertices  $Q$ 
5: put all vertices in  $Q$ 
6: while  $Q$  is not empty do
7:    $s$  first element in  $Q$ 
8:   delete  $s$  from  $Q$ 
9:   if  $d(s) \neq 0$  then
10:    instantiate empty path of edges  $p$ 
11:     $v := s$ 
12:    while  $d(v) \neq 0$  do
13:      pick random edge  $\{v, w\} \in E$ 
14:      delete  $\{v, w\}$  from  $E$ 
15:       $p := p \cup \{v, w\}$ 
16:       $v := w$ 
17:    end while
18:    put  $p$  in  $L$ 
19:  end if
20: end while
21: return  $L$ ;

```

The above algorithm runs in $O(m)$ time for k -regular bipartite graphs $G = (V_1 \cup V_2, E)$ where $|E| = m$. The algorithm starts by initialising a queue of all the vertices and an empty list of paths. The first vertex of the queue is pulled from the queue and an empty path is initialized. The path is iteratively elongated (by picking edge from the edge list of the current vertex, adding it to the path and deleting it from the graph, and updating current vertex to the value of the vertex on the other side of the picked edge) until we arrive at a vertex that has no edges left in its edge list. A new vertex is polled from the queue and a new list is initialised, repeating the argument above until there are no edges left in the graph. As an output of the algorithm, we have a list of paths. The assignment of the direction of an edge is made according to how the edge is traversed in the algorithm. As we see, only a constant number of operations is done per edge in this algorithm, justifying the claim made on the algorithm's complexity.

Let us prove that the time bound of finding k -edge colorings can be lowered to $O(km)$ time as well. This is done as follows:

Let $G = (V_1 \cup V_2, E)$ be a k -regular bipartite graph. If k is an odd integer, we can find a perfect matching M with algorithm 1, then assign a color to the edges in this matching, and recursively

find a $(k - 1)$ -edge coloring on the resulting graph $G' = (V_1 \cup V_2, E \setminus M)$ using a set of colors not containing the color used for the matching. If k is an even number, the graph could be split into two $\frac{k}{2}$ -regular bipartite graphs by determining an Eulerian orientation of G . The split is made by first making a $\frac{k}{2}$ -regular bipartite subgraph $G_1 = (V_1 \cup V_2, E_1)$ by taking the edges that are directed from V_1 to V_2 in the Eulerian orientation and the second subgraph $G_2 = (V_1 \cup V_2, E_2)$ by taking the edges directed from V_2 to V_1 . Then $|E_1| = |E_2| = \frac{m}{2}$ and $\Delta(G_1) = \Delta(G_2) = \frac{k}{2}$. This process of splitting even degree regular bipartite graphs $G = (V_1 \cup V_2, E)$ by means of an Eulerian orientation of G will from now on be called Euler split.

The time bound for finding a k -edge coloring for regular bipartite graphs can be calculated as follows: Let $T(k, m)$ be the running time of the algorithm on a k -regular bipartite graph on m edges. Then finding a perfect matching, if necessary, and Euler splitting the graph takes time ckm for some positive constant $c > 0$. Then $T(k, m) \leq 2ckm$ as we see that:

$$T(k, m) = ckm + 2T(\lfloor \frac{k}{2} \rfloor, \lfloor \frac{m}{2} \rfloor) = ckm + 2(c\lfloor \frac{k}{2} \rfloor \lfloor \frac{m}{2} \rfloor + 2T(\lfloor \frac{\lfloor \frac{k}{2} \rfloor}{2} \rfloor, \lfloor \frac{\lfloor \frac{m}{2} \rfloor}{2} \rfloor)) \leq ckm + ck\frac{m}{2} + ck\frac{m}{4} + \dots \leq 2ckm$$

using that $\lfloor k \rfloor \leq k$ for all $k \in \mathbb{R}$. The time needed for finding an Eulerian orientation is not taken into account in the separate terms that include ckm , as it is negligible compared to the time needed for finding perfect matchings.

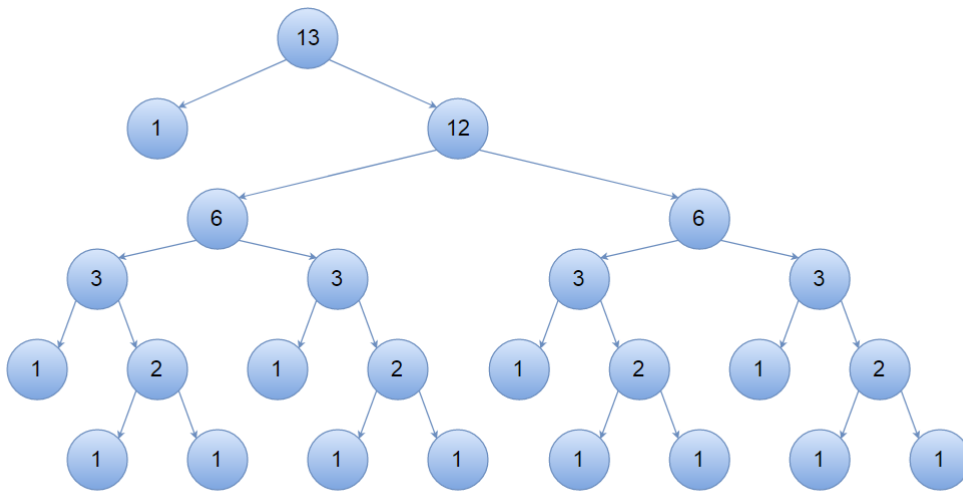


Figure 3.8: Visualisation of the edge coloring algorithm described above for $k = 13$. Except for the bottommost layer, arrows that point towards a 1 imply executing algorithm 1, while arrows that divide a number into two halves imply performing Euler splits.

Also, research will be conducted on the minimal edge coloring algorithm. This paper tries to solve the following questions:

- For small k , is there an optimal strategy for edge coloring a k -regular bipartite graph? (chapter 4)
- How many matchings are needed to edge color a graph? (chapter 4)

Chapter 4

Literature review

In this chapter, we are going to review literature that has been recently published on matching and coloring algorithms for bipartite graphs. As the paper by Schrijver has arrived almost two decades ago, a lot of new research on the subject has been conducted. Several scientific papers have been published in which mathematicians and computer scientists tried (and succeeded) to improve the time complexity of the perfect matching algorithm and the edge coloring algorithm.

Below we will give an overview of relevant, available techniques for decreasing the running time complexity of the algorithms. The reason for this is that our research will try to blend the methods described below to improve the algorithms in terms of asymptotic running time complexity. If improving the algorithms is not a possibility, we will try to develop an algorithm with the same or a slightly worse asymptotic running time complexity as the best algorithm known that is much simpler and easier to understand. We will go over the techniques described by the following researchers:

- Schrijver [11]
- Alon [1]
- Kapoor and Rizzi [8]

4.1 Using prime factorisation to edge color bipartite graphs

Schrijver himself provided a technique for speeding up his own algorithm, which makes use of the prime factorisation of the degree k . It could be derived as follows:

Suppose $G = (V, E)$ is a k -regular bipartite graph with $k = p_1 p_2 \dots p_t$ the **prime factorisation** of k such that $p_1 \leq p_2 \leq \dots \leq p_t$ primes. We will now give a definition that will be of great use in explaining the algorithm involving the prime factorisation of k .

Definition 4.1. Let $k \in \mathbb{N}$ and suppose $k = p_1 p_2 \dots p_t$ where each p_i is prime and $p_1 \leq p_2 \leq \dots \leq p_t$.

Then we define $\phi(k)$:

$$\phi(k) = \sum_{i=1}^t \frac{p_i}{\prod_{j=1}^{i-1} p_j}$$

This definition is needed for both the formulation and the proof of the next theorem:

Theorem 4.2. *Let $G = (V, E)$ be a k -regular bipartite graph. Then G could be k -edge colored in $O((\phi(k) + \log k)m)$ time.*

Proof. Suppose that $k = p_1 p_2 \dots p_t$ where each p_i is prime and $p_1 \leq p_2 \leq \dots \leq p_t$. Put $k' = p_2 p_3 \dots p_t$. Then $k = p_1 k'$. Then G can be partitioned in p_1 bipartite subgraphs which are all k' -regular, as follows:

Put $k' = p_2 p_3 \dots p_t$. Then, for each vertex $v \in V$, replace v by new vertices $v_1, v_2, \dots, v_{k'}$ and distribute the $p_1 k'$ edges incident to v equally over the newly created vertices. The resulting graph H is therefore a p_1 -regular bipartite graph with $k'|V|$ vertices. Then H can be p_1 -edge colored in $O(p_1 m)$ time where m is the number of edges of H , which is in fact the same number as for G . When each group of k' vertices belonging to an original vertex are merged together, we see that the edges of the original graph G can be partitioned in p_1 groups, which we will call E_1, E_2, \dots, E_{p_1} , as we can put all edges having the same color into one group, and we obtained an edge coloring of p_1 colors, yielding p_1 bipartite graphs that are k' -regular, namely $G_i = (V, E_i)$ for $i = 1, 2, \dots, p_1$.

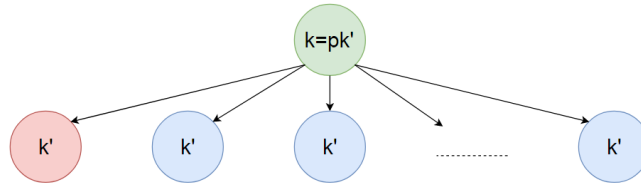


Figure 4.1: Result of the partitioning process described above. Now the first subgraph needs to be edge colored using iteration on the primes in the factorisation of k .

After the decomposition step as proposed above, we can proceed to edge coloring the original graph G by first finding a k' -edge coloring, without loss of generality, of G_1 . When the algorithm for finding this edge coloring is terminated, we have partitioned E into the sets of edges of k' -regular bipartite graphs E_2, E_3, \dots, E_{p_1} and perfect matchings $M_1, M_2, \dots, M_{k'}$. So the number of regular bipartite graphs is $p_1 - 1$ and the number of matchings is k' . Now we need to define the following variable, where we start at $\alpha = 1$:

$$\xi := \min\{\alpha, p_1 - \alpha\}$$

This variable keeps track of the minimum of the following two quantities: the number of k' -regular bipartite graphs that still need to be colored and the number of k' -regular bipartite graphs that are already broken up into perfect matchings. Now merge ξ (which is 1 in the first iteration) bipartite graphs that still need to be colored and add r readily obtained perfect matchings to the result such that $\xi k' + r = 2^t$ for some $t \in \mathbb{N}$ and $r \leq \xi k'$. We can efficiently edge color this final graph by recursively performing Euler splits on it (see algorithm 2). After this iteration, we therefore have

$(\xi k' + r) + (k' - r) = (\xi + 1)k'$ perfect matchings and $p_1 - 1 - \xi$ regular bipartite graphs still to be colored. In fact, repeating this argument means that ξ will always double, except maybe the last iteration. In each iteration, edge coloring the merged graph $\tilde{G} = (V, \tilde{E})$ with degree a power of two is done in $O(|\tilde{E}| \log \xi k' + r) = O(|\tilde{E}| \log k)$, as we work with subgraphs of the original k -regular bipartite graph G . We have that $\tilde{E} = \frac{1}{2}(\xi k' + r)n \leq \xi k'n$, and ξ doubles in each iteration, so all iterations together take a certain running time complexity of:

$$O\left((1 + 2 + 2^2 + \dots + 2^{\log p_1})k'n \log k\right) = O(2p_1 k'n \log k) = O(m \log k) \quad (4.1)$$

The above gives us the following general information above the time complexity of the proposed algorithm. At first, we need to find an edge coloring for a p_1 -regular bipartite graphs on m edges. This takes $O(p_1 m)$. We obtain the k' -regular subgraphs, of which one is edge colored. This graph we have to color has $\frac{m}{p_1}$ edges and $k' = p_2 p_3 \dots p_t$. So this graph can be recursively split in p_2 different subgraphs. Repeating this argument will finally yield the edge coloring of a p_t -regular bipartite graphs into p_t perfect matchings. Up until then, this step has taken $O\left(p_2 \frac{m}{p_1}\right) + O\left(p_3 \frac{m}{p_1 p_2}\right) + \dots + O\left(p_t \frac{m}{p_1 p_2 \dots p_{t-1}}\right) = O(\phi(k')m)$ time. Then, the reasoning as to make the degree a power of two can be applied, which takes $O(m \log k')$ time in total, as we saw in equation 4.1. So the edge coloring of the k' -regular bipartite graph takes $O((\phi(k') + \log k')m)$ time. Combining the first split into k' -regular bipartite graphs, finding one k' -edge coloring and using the argument of making the degree a power of two, we have obtained an algorithm with complexity $O(p_1 m) + O((\phi(k') + \log k')m) + O(m \log k) = O((\phi(k) + \log k)m)$, as we have:

$$\phi(k) = \sum_{i=1}^t \frac{p_i}{\prod_{j=1}^{i-1} p_j} = p_1 + \sum_{i=2}^t \frac{p_i}{\prod_{j=1}^{i-1} p_j} = p_1 + \frac{\phi(k')}{p_1} \quad (4.2)$$

□

It might not seem very trivial that the time bound proposed by this algorithm is indeed (much) better than the $O(km)$ bound we saw earlier. One might observe that, if $k = p_1 p_2 \dots p_t$, it holds that:

$$\phi(k) = \sum_{i=1}^t \frac{p_i}{\prod_{j=1}^{i-1} p_j} \leq \sum_{i=1}^t \frac{p_{\max}}{2^{i-1}} \leq 2p_{\max}$$

Therefore the algorithm above could be estimated by a time complexity of $O((p_{\max} + \log k)m)$. This means that the algorithm proposed above is not efficient for k that have a prime factorisation consisting of big maximum primes, and especially k that are prime themselves, as $\phi(p) = p$ if p is prime. Later on, we will research the time complexity of a new, completely different algorithm that might resolve this problem in some special cases of odd or prime degrees k . Given below is a table that gives the value of $\phi(k)$ for $k = 2, 3, \dots, 30$:

| k | $\phi(k)$ | k | $\phi(k)$ | k | $\phi(k)$ |
|-----|---------------|-----|----------------|-----|----------------|
| | | 11 | 11 | 21 | $\frac{16}{3}$ |
| 2 | 2 | 12 | $\frac{15}{4}$ | 22 | $\frac{15}{2}$ |
| 3 | 2 | 13 | 13 | 23 | 23 |
| 4 | 3 | 14 | $\frac{11}{2}$ | 24 | $\frac{31}{8}$ |
| 5 | 5 | 15 | $\frac{14}{3}$ | 25 | 6 |
| 6 | $\frac{7}{2}$ | 16 | $\frac{15}{4}$ | 26 | $\frac{17}{2}$ |
| 7 | 7 | 17 | 17 | 27 | $\frac{17}{3}$ |
| 8 | $\frac{7}{2}$ | 18 | 4 | 28 | $\frac{19}{4}$ |
| 9 | 4 | 19 | 19 | 29 | 29 |
| 10 | $\frac{9}{2}$ | 20 | $\frac{17}{4}$ | 30 | $\frac{13}{3}$ |

Table 4.1: Values of $\phi(k)$ for small $k = 2, 3, \dots, 30$

4.2 Adding perfect matchings

Another algorithm to take into account before we can look at the characteristics of interest of the algorithm by Schrijver and before we can try to improve its time complexity in terms of the degree k and the number of vertices n , is the algorithm proposed by Noga Alon [1]. As suggested earlier, the algorithm by Alon is simpler than the algorithm by Cole and Hopcroft [7], yet attains the same time bound of $O(m \log m)$. We will now explain its characteristic steps:

Theorem 4.3. *Let $G = (V, E)$ be a k -regular bipartite graph on m edges. Then a perfect matching on G can be found in $O(m \log m)$ time.*

Proof. Let $m = \frac{1}{2}nk$ be the number of edges of G and let $2^t \geq m$ such that t is as small as possible. Define $\gamma = \lfloor \frac{2^t}{k} \rfloor$ and $\beta = 2^t - \gamma k < k$. Find a random matching M in the complete bipartite graph $K_{\frac{n}{2}, \frac{n}{2}}$, this can be done in $O(n)$ time, but it can also be precomputed for each possibility of $n \in \mathbb{N}$ even, making its running time negligible in this algorithm. Now, do the following:

Create an empty graph $\tilde{G} = (V, \emptyset)$. For each edge $e \in E$, add γ copies of e to \tilde{G} . Then also add β copies of each edge $e \in M$ of the random matching. For each copy of an edge from this random matching, mark it as a dummy edge, so one can see that it does not belong to the original graph. In this case, we have that \tilde{G} is 2^t -regular with a total number of $\frac{1}{2}\beta n < \frac{1}{2}kn = m < 2^t$ dummy edges. We can perform an Euler split and from the two subgraphs obtained, consider the 2^{t-1} -regular bipartite graph with the fewest dummy edges, i.e. at most $\frac{1}{4}\beta n < \frac{1}{4}kn = \frac{m}{2} < 2^{t-1}$ dummy edges. Repeating this argument $t = \lceil \log m \rceil$ times, we have obtained a 1-regular bipartite graph with less than 1 dummy edge, so this means it is in fact a perfect matching from the original graph G . As we saw earlier, in a regular bipartite graph with degree $k = 2^t$, a perfect matching could be found in $O(tm) = O(m \log k)$ time, as it involves performing t Euler splits where each split takes $O(m)$ time. In this case, the degree of the regular bipartite graph is $O(m)$, so this implies a $O(m \log m)$ algorithm. \square

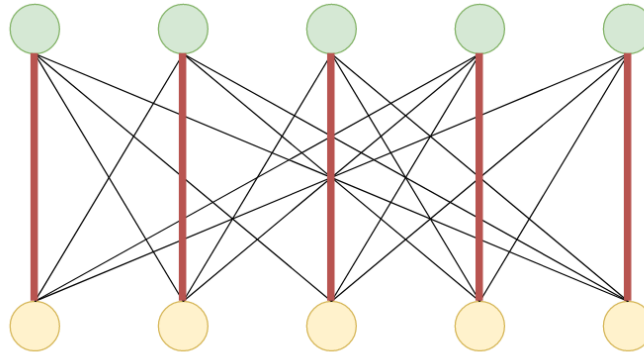


Figure 4.2: The 3-regular bipartite graph given by the black edges is expanded by adding one dummy perfect matching (given in red) to make k a power of 2, i.e. 4.

4.3 Strategy for finding minimal edge colorings in k -regular bipartite graphs

In their 2000 paper, Kapoor and Rizzi [8] give a strategy for finding minimal edge colorings in k -regular bipartite graphs. In their paper, they introduced new graph theory concepts that need to be defined here as well to be able to understand their method. Firstly, they introduce bins:

Definition 4.4. (Bin, almost solved bin)

A **bin** of a k -regular bipartite graph $G = (V, E)$ is a list of integers (b_1, b_2, \dots, b_p) such that E is partitioned into subsets E_1, E_2, \dots, E_p and $G_i = (V, E_i)$ is a b_i -regular bipartite graph. A bin is **almost solved** if:

- $b_1 = 1$
- $b_i \leq \sum_{j=1}^{i-1} b_j$

The notion of bins in the proposed method drastically increases the simplicity of the notations, as operations as executing a perfect matching algorithm and performing Euler splits are summarised by the degrees of the subgraphs obtained from the original graph. Apart from perfect matching algorithms and Euler splits, a new variation on Euler splits is introduced, namely Euler splits on even degree regular bipartite graphs after two odd degree regular bipartite graphs have merged. This method is denoted by Odd-Euler(k_1, k_2) when $k_1 \equiv k_2 \equiv 1 \pmod{2}$ degrees of regular bipartite graphs.

Furthermore, the method that they propose proves a useful theorem. But first, we need to define subroutine for finding almost solved bins to make the proof of the theorem easier to understand. For odd k , the inevitable start of the method is to find a perfect matching first, and performing an Euler split on the residual $(k-1)$ -regular bipartite graph, yielding the starting bin $(1, \frac{k-1}{2}, \frac{k-1}{2})$. This starting bin can be transformed in an almost solved bin by performing the following algorithm:

Algorithm 3 Finding an almost-solved bin of a k -regular bipartite graph, k odd

```

1: Input: odd  $k$ .
2: Output: almost solved bin of  $k$ .
3:  $H := (1, \frac{k-1}{2}, \frac{k-1}{2}) = (a, b, b)$ ;
4:  $T := \emptyset$ ;
5:  $B := (H, T)$ ;
6: while  $a \neq b$  do
7:   while  $b \equiv 0 \pmod{2}$  do
8:      $T = (b, T)$ ;
9:      $H = (a, \frac{b}{2}, \frac{b}{2})$ ;
10:     $b = \frac{b}{2}$ ;
11:   end while
12:   if  $a \neq b$  then
13:     while  $\frac{a+b}{2} \equiv 1 \pmod{2}$  do
14:        $H = (b, \text{Odd-Euler}(a, b))$ ;
15:        $temp = a$ ;
16:        $a = b$ ;
17:        $b = \frac{temp+b}{2}$ ;
18:     end while
19:   end if
20: end while
21: return  $B$ ;

```

At the start of the algorithm's execution, we see that the bin is given as the concatenation of the header $H = (1, \frac{k-1}{2}, \frac{k-1}{2}) = (a, b, b)$ and the empty tail T . This means that in the beginning, $a \neq b$, so the code inside the loop defined by line 6-20 is always executed.

First, lines 7-11 make sure that the second and third element of the header are odd natural numbers. This is done by possibly performing Euler splits on one of the even degree regular bipartite graph represented by b and pushing the graph that is left over in the process to the front of the tail part of the bin. An important aspect that we can conclude from this is that the natural numbers that occur in the tail part of the bin are smaller or equal than the sum of the header elements and the tail elements appearing earlier in the tail. More formally, this means that for all elements $t_i \in T$:

$$t_i \leq \text{Val}(H) + \sum_{j < i} t_j$$

where $\text{Val}(H)$ is the sum of the elements in the header H . This complies with the desired property of almost solved bins, so with this explanation, the statement is proven for the tail of the bin.

Now, look at the header of the bin. It consists of three odd natural numbers, of which the second and third are equal. Then, repeatedly change the header from (a, b, b) to $(b, \frac{a+b}{2}, \frac{a+b}{2})$ and updating a and b to the new values, by taking the first and second subgraph induced by the elements in the bin, merging the subgraphs and performing an Euler split upon them, which is possible as the degrees of the subgraphs were both odd, so an even degree regular bipartite graph was created. In this repeated process, the new second and third subgraph represented by the header elements will eventually become even degree subgraphs. This way the sum of the header's elements

decreases when the whole algorithm performs an iteration defined in lines 6-20. This algorithm clearly terminates, as $Val(H)$ decreases each iteration in 6-20 and $Val(H) \geq 3$, as the minimum objects considered are perfect matchings or 1-regular bipartite graphs. Furthermore, at the end of the execution $a = b$, so clearly, as already proven for the tail of the bin, also the header complies to the definition of almost solved bins, so the algorithm outputs an almost solved bin.

Now, all the tools needed to understand the method by Kapoor and Rizzi have been introduced. We prove the following statement by giving their strategy:

Theorem 4.5. *Let $G = (V, E)$ be a k -regular bipartite graph with $k \in \mathbb{N}$ arbitrary. Then we only need to perform the perfect matching algorithm at most once to be able to find a minimal edge coloring, i.e. using k colors, for G .*

Proof. First, consider the case k is even. Then one can rewrite $k := s \cdot 2^r$ where s is odd. Then you should recursively perform Euler splits on one of the two subgraphs obtained by an Euler split (and subsequent subgraphs) until you have two regular bipartite subgraphs with degree s :

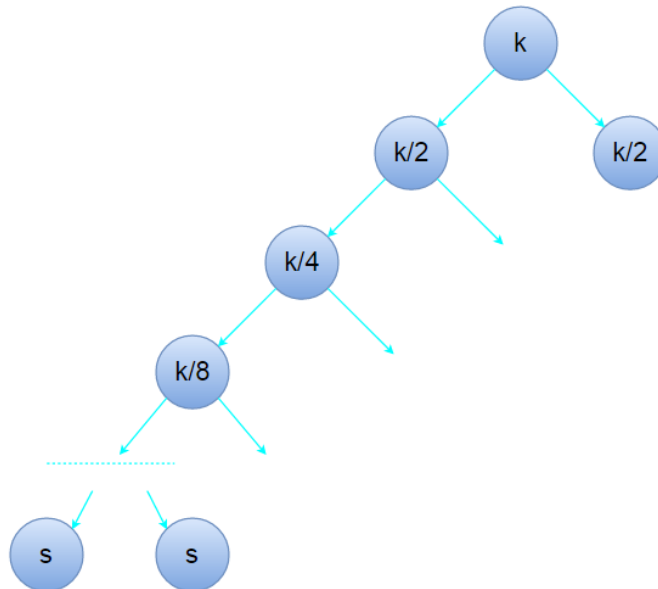


Figure 4.3: Visualisation of the strategy for even k .

Edge coloring the original k -regular bipartite graph is done as follows: Find recursively a s -edge coloring of the first s -regular subgraph. This yields us s perfect matchings. Then, we can edge color the other subgraphs without having to find a perfect matching. Firstly, the other s -regular subgraph is colored by adding a subset of the s perfect matchings we already have to it such that we obtain a 2^t -regular bipartite graph for some $t \in \mathbb{N}$. This is possible, as there always exists $t \in \mathbb{N}$ such that $s \leq 2^t < 2s$. After this we have $2s$ perfect matchings. Repeat this argument while going back up the subtree as shown in Figure 4.3 until a k -edge coloring is found for G . This argument implies that we only need to prove the statement for odd k .

Therefore, let us focus on odd k . As we saw earlier, the inevitable start of a k -edge coloring algorithm for odd k is to find a perfect matching and Euler split the remaining $(k - 1)$ -regular bipartite

graph. At the start of the execution of algorithm 3, we see that the bin is given as the concatenation of the header $H = (1, \frac{k-1}{2}, \frac{k-1}{2}) = (a, b, b)$ and the empty tail T . This means that in the beginning, $a \neq b$ if $k \neq 3$, so the code inside the loop defined by line 6-20 is always executed.

We see that algorithm 3 does not make use of finding perfect matchings at all, making the call to the perfect matching algorithm before finding the starting bin the only time that we need to execute the perfect matching algorithm for odd k until we have an almost solved bin.

When we have an almost solved bin, all we have to do to find a k -edge coloring from this bin is iteratively solve subbins with the property $(1, \dots, 1, b_i)$ with at least b_i ones. This is analogous to the reasoning used above in the iterative process of edge coloring subgraphs by using already found matchings to make regular bipartite graphs with degree a power of two, as those kind of regular bipartite graphs can be colored efficiently. Therefore, the process after obtaining an almost solved bin does not require finding another perfect matching anymore, thus proving the statement. \square

Chapter 5

Explanation of JAVA model

5.1 JAVA implementation of Schrijver's $O(km)$ algorithm

The main goal of the research is to efficiently implement the algorithm described by Alexander Schrijver in his paper (algorithm 1). For the implementation of the code, JAVA was used as the programming language. To be able to implement the code such that the theoretical bound of $O(\Delta m)$ could be obtained, this required an analysis on the algorithm itself and on useful data structures that are available in JAVA. Therefore, the next section will thoroughly analyse which data structures to use for this implementation.

5.2 Use of data structures

The concept of graphs is widely used in computer programming languages and there are a lot of ways to program graphs, where each implementation has its own applications in which it is efficient.

The two most commonly used data structures for implementing graphs are adjacency matrices and adjacency lists. The adjacency matrix A of a graph $G = (V, E)$ is a $|V| \times |V|$ -matrix in which the matrix element a_{ij} has value 0 if there is no $e \in E$ such that $e = \{i, j\}$, and 1 if there actually exists an edge connecting vertices i and j . Although accessing edges and vertices in the graph is very time-efficient in this way, namely $O(1)$, as it only takes reading a value from a matrix, this implementation is mostly very inefficient in terms of memory space, as zero-valued array elements do not actually contain any useful information about the graph, except that there is no edge between two vertices of the graph. This bottleneck really plays a role when you are dealing with sparse graphs, that is a graph with a relatively low number of edges compared to the maximum possible number of edges or in other words, a graph with an adjacency matrix that relatively contains a lot of zeroes. Furthermore, adjacency matrices cannot be used for multigraphs, as each matrix coefficient only determines if there exists an edge between two vertices, not how many edges. This problem can be overcome by changing the matrix elements to the number of edges between i and j .

On the other hand, in a graph $G = (V, E)$, an adjacency list of a vertex $v \in V$ is a list of edges that are incident to v . These edges can be characterized by giving the other endvertex of the edge and

the weight of the edge. A graph therefore consists of the set of adjacency lists of all vertices in the graph. However, this means that each edge is represented twice in each graph, as it occurs in its both endpoints' adjacency lists. Because of this, each representation of an edge needs to be linked to the other representation, in order to avoid the case in which one representation is deleted while the other still exists. This tiny waste of loss of memory space could be justified as it only takes memory space if there exists an edge. Especially in sparse graphs this implementation should, depending on which algorithm or subroutines you want to apply, often be chosen over the use of an adjacency matrix. The main disadvantage of this result is that edges could not be accessed in $O(1)$ time, except for the first and last c edges of the doubly linked list where c is a constant natural number. However, heuristics on finding circuits for Schrijver's algorithm will be described that only require accessing the first c edges in a doubly linked list where c is a constant, so this disadvantageous property of doubly linked lists does not negatively influence the complexity bound of our implementation of Schrijver's algorithm anymore. This implementation makes use of a so-called adjacency map, which is a mapping $f : V \rightarrow E$ so that $f(v)$ is the adjacency list of vertex v . This means that each vertex contains a doubly linked list of the edges that are incident to it.

For our implementation of algorithm 1, the use of adjacency lists is preferred, as the algorithm will mostly be applied to k -regular bipartite graphs such that k is either bounded or fixed, the assumption is made that this particular set of graphs is rather sparse (i.e. $k \ll n$), so the adjacency matrix implementation is especially abandoned for its high space complexity, to avoid the fact that the amount of space needed for storing and referencing variables of an algorithm exceeds the size of a heap. Furthermore, the asymptotic running time complexity of algorithm 1 is higher than the complexity of the algorithm described later on in theorem 6.1, so our focus with Schrijver's algorithm can be narrowed to finding perfect matchings in relatively sparse graphs. The most efficient data structure for adjacency lists are doubly linked lists [5]. In a doubly linked list, each element of the list has a pointer to the element that is directly previous and directly next to it in the list. The main advantage of doubly linked lists is that adding and removing edges from a graph are $O(1)$, as this only takes a constant number of operations assuring the pointers point to other elements in the list. In the explanation of the JAVA algorithm, it will become clear why this is so essential.

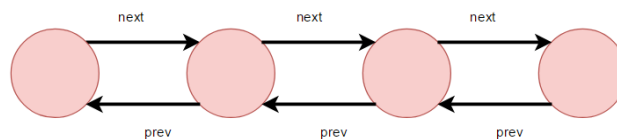


Figure 5.1: In a doubly linked list, each list element contains a pointer to the previous and next element in the list. Removing an element now only involves updating pointer values.

5.3 Complexity of elementary operations

This subsection will deal with the elementary operations that are needed to perform the algorithms of interest. The complexity bounds are derived from the structure of doubly linked lists. A list of the fundamental operations needed is given below:

| Operation | Complexity |
|---|-----------------------|
| Get the number of vertices | $O(1)$ |
| Get the set of vertices | $O(n)$ |
| Get the number of edges | $O(1)$ |
| Get the set of edges adjacent to u | $O(d_u)$ |
| Check if the edge $e = \{u, v\}$ exists | $O(\min\{d_u, d_v\})$ |
| Get the degree of u | $O(1)$ |
| Add to / remove from the graph the vertex u | $O(d_u)$ |
| Add to / remove from the graph the edge e | $O(1)$ |
| Move edge e to the end of the edge list | $O(1)$ |
| Get the first / last of an edge list | $O(1)$ |
| Check or mark the visited variable of an edge or a vertex | $O(1)$ |
| Get or set the weight of an edge | $O(1)$ |
| Get or set corresponding edge of an edge | $O(1)$ |
| Get or set previous or next pointer in an edge list | $O(1)$ |

Table 5.1: Table with the fundamental operations needed for implementing Schrijver's algorithm with doubly linked lists. The letters u and v are used for referencing vertices, and e for edges.

The complexity bounds of the operations described are trivial, but mentioned for completeness sake. Therefore, from now on, these operations will be used in explaining the computer program without proof of their complexities

5.4 Graph concepts for Schrijver's algorithm

When analyzing the algorithm of Schrijver, one should take into account which operations have to be performed to successfully execute the algorithm. These operations are defined in table 5.1 and also determine what information should be contained in the objects' vertices and edges to perform the algorithm. Firstly, an empty list P is created. This list P will be the list of edges that form the circuit-finding walk. A random vertex u is picked from the k -regular bipartite graph in the input and marked visited and an edge in the adjacency list of u is picked by means of a specific heuristic and added to L . This edge is marked visited, as well as its corresponding edge in the adjacency list of the other endvertex of the edge. Then the added edge is traversed and u becomes the value of the other endvertex of the added edge. This process is repeated until the updated u is already marked visited, as this means that the circuit-finding path has found a circuit in the graph. The circuit will be traversed once again to determine the minimum weight of the edges in the circuit and to determine matchings M and N such that $w(M) \geq w(N)$. Depending on which matching an edge is contained in, the edge will be updated above or below depending on the choice of the maximal matchings M and N in the circuit.

In case an edge is updated to value zero, the edge and its corresponding edge are moved to the end of the respective adjacency lists they are elements of. This is done by removing the edge from the adjacency lists and remembering its reference, and adding it back to the end of the list, doing the same for the corresponding edge. In this way, no information about the edge is lost in the process, while pushing a zero weight edge to the end of the list is analogous to removing an edge from

$E_w := \{e \in E \mid w(e) > 0\}$. After updating, a new circuit-searching path is constructed by starting with the tail of P not contained to the found circuit, i.e. $P := P \setminus C$ and the process of adding edges and updating u is repeated iteratively. (When P is empty, this means that we have to search for a vertex such that the first edge in its adjacency list does not have weight equal to k). The algorithm ends when the weight of the first edge in each vertex' adjacency list is equal to k , this means a perfect matching is found and given by the first edges in each vertex' doubly linked list of edges.

5.4.1 Attributes of the graph's subobjects EDGE and VERTEX

The explanation above makes clear what information the objects VERTEX and EDGE need to contain in order to execute the steps. Essential information for vertices is a boolean variable `isVisited`, and the name of the vertex. The variable `isVisited` is set to false for all vertices and edges at the start of execution, and when a vertex is being visited it is set to true. The name of the vertex is used to identify a vertex. For edges, much more information is needed. An object EDGE should contain information about the two vertices it is incident to (the first vertex is the vertex whose doubly linked list contains this EDGE object and the second is the adjacent vertex), its weight, a pointer to the corresponding EDGE object (contained in the second vertex' doubly linked list), also a boolean variable `isVisited` and of course pointers to the previous and next edge in the adjacency list it is contained in, as these adjacency lists are doubly linked. In the algorithm for finding minimal edge colorings, edges trivially also need to reserve memory space for the color of the edge.

5.5 Methods for performing simulation of Schrijver's algorithm

5.5.1 Choosing edges in the circuit-finding path

Up until now, we have the basic knowledge on the objects VERTEX and EDGE that are needed to perform the algorithm speaking in general pseudo code language. However, the algorithm of Schrijver leaves space to choosing techniques for picking edges in the circuit-searching path due to the general description of the algorithm. Using the defining characteristics of doubly linked lists, there are several heuristics that we want to pinpoint and investigate in the research part, the names of the methods are given in between brackets:

- pick the first edge of the list, or the second if the first is already visited (FIRST)
- Suppose $c \in \mathbb{N}$ a constant. Pick from the first c edges in the list the edge with maximum weight (FIRST c MAXIMUM)
- Analogous to (FIRST c MAXIMUM), but now take the minimum (FIRST c MINIMUM)
- Analogous to (FIRST c MAXIMUM), but now pick alternately the edge with maximum weight and the edge with minimum weight from the first C edges of the list (FIRST c ALTERNATE)

For each heuristic, except for FIRST, edges that appear earlier in the list are prioritised when there are multiple maximum weight edges are found. For example, if from the first five edges, both the second and the fifth edge have maximum weight, we will pick the second as it appears earlier in the list. The reason for considering the methods listed above is that the complexity of picking an

edge and adding it to the circuit finding path is $O(1)$ for all of them, while other methods contain iteration over a nonconstant number of edges, inherently implying higher complexity bounds than constant time.

The main question that we want to answer with the analysis of the methods listed above is:

- Which of the above methods should be chosen in order to specify Schrijver's algorithm such that it is as efficient as possible in terms of number of fundamental operations, as specified in table 5.1?

5.5.2 Performing Euler split regular bipartite graphs of even degree

Another aspect of the algorithms of Schrijver, specifically for the edge coloring algorithm, is using a way to divide a regular bipartite graph into smaller regular bipartite graphs that are equally large in terms of number of edges. Earlier, we already saw that this can be done by a technique due to Gabow called 'Euler splits' [4], as it is derived from the fact that even degree regular bipartite graphs do actually have the property of being Eulerian graphs. In the JAVA model, the technique is much more equivalent to the definition we described earlier in algorithm 2 than generally known algorithms for finding an Euler tour in regular bipartite graphs.

At the start of the algorithm of Gabow, we have an original input graph G and an empty list of paths L is used for maintaining edge paths that are to be found during the execution of the algorithm. Furthermore, an empty queue Q is initialized for vertices of G . All vertices are consecutively offered to the queue. While the queue is not empty, a new path P is initialized. Then, pick the first element in the queue and iteratively pick an edge from the vertex' adjacency list and traverse the edge to the other endvertex of the edge. The traversed edge is removed from G and added to path P . This process stops if a vertex is attained that has no edges left in its adjacency list and P is added to L . Again, the first element of the queue is picked. If the element has no edges in its adjacency list, the next vertex in the queue is polled, otherwise the statement above of making paths as described above is repeated until the queue is empty.

5.5.3 Generating random regular bipartite graphs

In order to perform an analysis of the algorithm by means of the JAVA model, we need to provide a function that can generate random regular bipartite graphs that corresponds with the input of the degree k and the number of vertices n such that the algorithm of Schrijver can be tested on the generated graph. This is done in the following way:

Let us consider the model graph $G = (V_1 \cup V_2, E)$ in which we denote the vertices in vertex color class V_1 by $v_1, v_2, \dots, v_{\frac{n}{2}}$ and the vertices in the second color class V_2 by $v_{\frac{n}{2}+1}, v_{\frac{n}{2}+2}, \dots, v_n$. We want to create a random k -regular bipartite graph $G = (V_1 \cup V_2, E)$ and to be able to do this, we create an array (or a similar data structure that is more efficient) of the $\frac{1}{2}nk$ edges which we will denote by $e_1, e_2, \dots, e_{\frac{1}{2}nk}$.

In the implementation used, to create k -regularity in the first vertex class V_1 , let e_1, \dots, e_k be the edges incident to v_1 , e_{k+1}, \dots, e_{2k} the edges incident to v_2 and so forth until $v_{\frac{n}{2}}$. Then, perform a shuffle technique on the array A of edges (now ordered such that $A[i] = e_{i+1}$ for $i = 0, 1, \dots, \frac{1}{2}nk - 1$ (as array coefficients start at 0). In this implementation, the technique introduced by Donald Knuth is used [9], known as Knuth shuffle or Fisher-Yates shuffle in a slightly different form. Note that the first element of the array gets index 0. It performs the following algorithm on an array of length n :

Algorithm 4 Shuffling the elements in an array (Knuth, 1997)

```

1: Input: An array  $A$  of length  $n$ .
2: Output: An array  $A'$  of length  $n$  with  $A$ 's elements' names, but shuffled.
3: for  $i$  from  $n - 1$  downto 1 do
4:    $j$  = random integer such that  $0 \leq j \leq i$ ;
5:    $\text{temp} = A[i]$ ;
6:    $A[i] = A[j]$ ;
7:    $A[j] = \text{temp}$ ;
8: end for
9: return  $A$ ;
```

The output of algorithm 4 is an array A' ordered as $e'_1, e'_2, \dots, e'_{\frac{1}{2}nk}$. Now, consider the second vertex class V_2 and let e'_1, \dots, e'_k be the edges incident to $v_{\frac{n}{2}+1}$, e'_{k+1}, \dots, e'_{2k} be the edges incident to $v_{\frac{n}{2}+2}$ and so forth until v_n . This results in a k -regular bipartite graph, generated randomly due to the stochastic character of algorithm 4. So our initial array determines the first vertex (in V_1) of the edge, and the output of the algorithm determines the second vertex (in V_2) of the edge, yielding a k -regular bipartite graph.

5.6 Methods for the algorithm based on Alon's

The implementation of the algorithm by Alon is not very different. It uses all the tools described in section 5.5 as well, except for the method concerning finding circuits. The only difference is an extra method that is used for pre-processing purposes.

5.6.1 Generating and adding random 'dummy' matchings

This method contains the same technique for vertex labelling as described in section 5.5.3. In the input of the algorithm based on Alon, we can derive n easily from the vertex set of the input graph, or it can be asked as input as well. Given the number of vertices n , we can generate a random matching in $O(n)$ time, but as it can be used for any input graph with number of vertices n , these random matchings can be pre-processed and kept in a database. The generation of random matchings is done as follows:

For an input graph $G = (V, E)$, which evidently has to be a regular bipartite graph, let the number of vertices be $|V| = n$. Let V be partitioned in sets V_1 and V_2 . Then, label the vertices that are element of V_1 with $v_1, v_2, \dots, v_{\frac{n}{2}}$ and the vertices in V_2 with $v_{\frac{n}{2}+1}, \dots, v_n$. Trivially, as it does not matter which

matching is used, as long as it is perfect, one can construct a perfect matching by taking the set of edges with end-vertices v_i and $v_{i+\frac{n}{2}}$ for $i = 1, 2, \dots, \frac{n}{2}$.

Another problem that needs to be addressed is the fact that the generated dummy edges need to be added a specific number of times to the original graph, dependent of k . However, adding matchings are desired to be done in constant time. This means that pre-processing needs to be done on adding perfect matchings of dummy edges as well. The input of the algorithm should be a regular bipartite graph $G = (V, E)$, $k = \frac{2|E|}{|V|}$ and $n = |V|$. If that is the case, a regular bipartite graph can be generated with degree a power of two in constant time. This argument is repeated in the case that an iteration does not find a perfect matching that consists of merely real edges of the original graph G .

Chapter 6

Research

This chapter contains the new discoveries we made using the theory available. At first, one of the existing algorithms examined will be improved in terms of complexity in two different ways. Description of these discoveries are needed for understanding the empirical research where the methods are compared to Alon's.

6.1 Improving Alon's algorithm

In this section, a new algorithm for finding perfect matchings is proposed, which is heavily based on the algorithm by Alon [1]. We will compare the complexity bounds of both algorithms after proving the following theorem:

Theorem 6.1. *Let $G = (V, E)$ be a k -regular bipartite graph on n vertices and $m = \frac{1}{2}nk$ edges. Then a perfect matching in G can be found in $O(m \log n)$ time.*

Proof. Let r be such that $2^{r-1} < k \leq 2^r$. Then define $\alpha := 2^r - k < 2^{r-1}$. Using the same technique as Alon, pick a random perfect matching from the complete bipartite graph $K_{\frac{n}{2}, \frac{n}{2}}$, which can be pre-computed for general n taking $O(n)$ time. Create an empty graph $G' = (V, \emptyset)$ and add the set of edges E to G' , as well as α copies of each edge in the random perfect matching, which we mark as being dummy edges. This means that we have obtained a 2^r -regular bipartite graph G' with $\frac{1}{2}n\alpha < \frac{1}{2}n2^{r-1} = n2^{r-2}$ dummy edges. Perform an Euler split on $H = G'$ and from the two subgraphs, consider the one with the fewest dummy edges. When we repeat this argument until we have a 1-regular bipartite graph \tilde{M} , we can assure that \tilde{M} has at most $\frac{1}{2} \frac{1}{2^r} n\alpha < \frac{1}{2} \frac{1}{2^r} n2^{r-1} = \frac{1}{4}n$ dummy edges. In the case of $n < 4$, this implies that you have indeed found a perfect matching, as the number of dummy edges is less than one in this case. Now, instead of using a random perfect matching on $K_{\frac{n}{2}, \frac{n}{2}}$, use the obtained matching \tilde{M} and overwrite the G' by copying the original graph G and adding α copies of \tilde{M} . Now we have accomplished to get a 2^r -regular bipartite graph G' with less than $\frac{1}{4}n\alpha < \frac{1}{4}n2^{r-1} = n2^{r-3}$ dummy edges. One can find a perfect matching with fewer than $\frac{1}{8}n$ dummy edges now.

When we generalise this statement, after i iterations, including the first in which we use a random matching, we have a perfect matching from the 2^r -regular auxiliary graph with fewer than $\frac{1}{2^{i+1}}n$ dummy edges. This perfect matching of the auxiliary graph is a perfect matching of the original

graph for the first time after i iterations when:

$$\frac{1}{2^{i+1}}n < 1 \leq \frac{1}{2^i}n \quad (6.1)$$

$$\frac{2^i}{n} \leq 1 < \frac{2^{i+1}}{n} \quad (6.2)$$

$$2^i \leq n < 2^{i+1} \quad (6.3)$$

$$i \leq \log n < i + 1 \quad (6.4)$$

This means that the number of iterations needed to perform the algorithm is $O(\log n)$. In each iteration, we perform an Euler split on a 2^r -regular, on a 2^{r-1} -regular, ..., and a 2-regular bipartite graph, taking $O\left(\frac{1}{2}n(2^r + 2^{r-1} + 2^{r-2} + \dots + 2)\right) = O\left(\frac{1}{2}n2^{r-1}(2 + 1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^{r-2}})\right) = O(2nk) = O(m)$ time. Because of this, we have the required time bound. \square

For the technique described above, significantly fewer dummy edges need to be added to the original graph, at the cost of needing multiple iterations to finish the algorithm. An iteration of this algorithm is defined as the process of splitting down a 2^f -regular bipartite graph until an 1-regular bipartite graph is obtained (not necessarily only consisting of edges of the original graph G).

Theorem 6.1 gives us an algorithm with an better worst-case complexity than Schrijver's $O(km)$ algorithm for a specific subclass of regular bipartite graphs, namely regular bipartite graphs for which $\log n < k$, or in other words $n < 2^k$. This means that the algorithm is faster for regular bipartite graphs that satisfy a certain graph density, specifically when the number of vertices is at most exponential in the degree of the graph. Asymptotically speaking, this is an significant improvement for a subclass of regular bipartite graphs. However, as the complexity of algorithms in this report are given by their worst-case scenario running time, we cannot draw strong conclusions on the average-case behaviour of both Schrijver's algorithm and the algorithm described in theorem 6.1.

6.2 Improving the $O(m \log n)$ method

In section 6.1, a new method for finding perfect matchings in regular bipartite graphs was defined based on a known scheme by Noga Alon [1]. This section will give an improvement on the aforementioned method. The method is described below, and we will show that it actually is an improvement of the method described in the proof of theorem 6.1:

Let r and t be such that $2^{r-1} < k \leq 2^r$. Using the same technique as Alon and algorithm described in theorem 6.1, pick a random perfect matching from the complete bipartite graph $K_{\frac{n}{2}, \frac{n}{2}}$, which can be precomputed for general n taking $O(n)$ time. Then apply the following algorithm, which is fully based on the degree k of the regular bipartite graph G :

Algorithm 5 Finding a perfect matching in a k -regular bipartite graph

```

1: Input:  $k$ -regular bipartite graph  $G = (V, E)$ .
2: Output: perfect matching  $F \subseteq E$  in  $G$ .
3: Create copy  $G'$  of original graph  $G$ ;
4: Create random dummy matching  $M$  with  $|M| = \frac{1}{2}|V|$ ;
5: repeat
6:   while  $k \neq 1$  do
7:     if  $k \equiv 1 \pmod{2}$  then
8:        $G' := (V, E \cup M)$ ;
9:     end if
10:    EulerSplit( $G'$ );
11:     $G'$  becomes the subgraph obtained by the previous command with the fewest dummies.
12:  end while  $M := G'$ 
13: until  $M$  contains no dummy edges
14: return  $M$ ;

```

What the algorithm does, contrary to the algorithm in theorem 6.1, is performing an Euler split on a regular bipartite graph when possible, and if not possible, this is always caused by the fact that the current graph has odd degree. When this is the case, a dummy matching (in later iterations this matching is a combination of 'real' and dummy edges) is added to the graph to make the degree even. During an iteration, we continue performing Euler splits and possibly adding matchings until we obtain a perfect matching. Then, there are two cases we should consider. In the first case, this perfect matching consists only of edges occurring in the original graph G and the algorithm terminates. In the other case, the perfect matching consists also of dummy edges, so another iteration is needed before the algorithm can terminate. Then, at the start of the new iteration, not a random pre-processed matching is used, but the matching obtained in the previous iteration. The advantage of this is that this perfect matching contains fewer dummy edges, so in an iteration, fewer dummy edges are added to the graph.

The calculation for the complexity bound is as follows. We will focus on proving the statement for odd k . Namely, when the degree k is even, we can write $k = 2^s t$ with $t \in \mathbb{N}$ odd. Then, performing s Euler splits on the original k -regular bipartite graph results in a t -regular bipartite graph. This can be done in $O\left(\frac{n}{2} t (2^s + 2^{s-1} + \dots + 2)\right) = O\left(\frac{n}{2} t 2^s \left(1 + \frac{1}{2} + \dots + \frac{1}{2^{s-1}}\right)\right) = O\left(\frac{n}{2} t 2^s\right) = O\left(\frac{n}{2} k\right) = O(m)$ time. This complexity bound is smaller than the complexity bound we try to prove for the whole algorithm, so we can justify our focus. Obviously, the worst case scenario is the case where after adding a matching and performing an Euler split, the resulting subgraph always has odd degree. This is actually the case for $k = 2^t + 1$ with $t \in \mathbb{N}$. Namely:

$$k = 2^t + 1 \Rightarrow \frac{k+1}{2} = \frac{2^t + 1 + 1}{2} = 2^{t-1} + 1$$

During the first iteration of the algorithm, this means that for each added matching (this happens t times), a total number of $\frac{n}{2}$ dummy edges are added to the graph. In total, $t + 1$ Euler splits are performed. After the addition of the first matching, an Euler split is performed and maximally $\frac{n}{4}$ dummy edges are left in the sub-graph. So adding another matching brings us maximally $\frac{3n}{4}$ dummy edges which is decreased to maximally $\frac{3n}{8}$ after the next Euler split and this process repeats until a perfect matching (whether or not containing dummy edges) is found. At last, two

consecutive Euler splits are performed, after a matching is added to make the degree equal to 4. This makes the maximum number of dummy edges after the first iteration equal to $\frac{2^t-1}{2^{t+2}}n$. Notice that for large t , this value converges to $\frac{n}{4}$ from below. Therefore, an upper bound on the number of dummy edges after the first iteration of the algorithm could be given by $\frac{n}{4}$. Working with his newly obtained perfect matching in the second matching yields an upper bound of $\frac{n}{8}$ dummy edges in the perfect matching acquired after the second iteration. In general, this means that the upper bound of dummy edges in the perfect matching after i iterations equals $\frac{1}{2^{i+1}}n$. The algorithm terminates when the number of dummy edges is equal to zero, in other words smaller than one, so the number of iterations is $O(\log n)$.

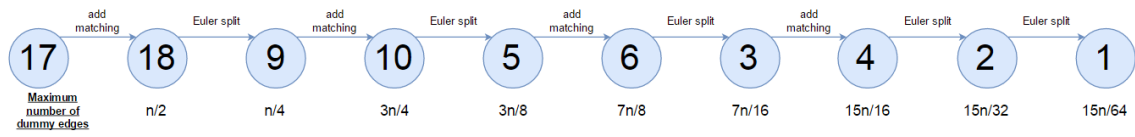


Figure 6.1: Strategy for the first iteration of the algorithm for $k = 17$. The nodes give the degrees of the graphs on which operations (add matching and Euler splits) are applied, until $k = 1$. Furthermore, the maximum number of dummy edges, i.e. in a worst case scenario, is given underneath the nodes.

During each iteration, a total number of t matchings needed to be added to the graph, taking $O(tn) = O(n \log k)$ time. Furthermore, Euler splits needed to be performed on regular bipartite graphs with the following degrees:

$$2^t + 2, 2^{t-1} + 2, \dots, 4 \text{ and } 2$$

This takes asymptotically:

$$\begin{aligned}
 O\left(n\left((2^t + 2) + (2^{t-1} + 2) + \dots + 4 + 2\right)\right) &= O\left(n\left(2^{t+1} + 2^t + \dots + 4 + 2\right)\right) \\
 &= O\left(n2^{t+1}\left(1 + \frac{1}{2} + \dots + \frac{1}{2^{t-1}} + \frac{1}{2^t}\right)\right) \\
 &= O(4nk) \\
 &= O(m)
 \end{aligned}$$

time. Therefore, the time needed for successfully executing the algorithm is $O(\log n(n \log k + m)) = O(m \log n)$.

The asymptotical running time complexity of this algorithm might not be faster than the algorithm described in theorem 6.1, but the number of dummy matchings that are added to the graph is smaller. Moreover, an equal number of Euler splits need to be performed, but these splits are performed on smaller graphs, which means that the number of steps needed to execute the algorithm is decreased by a positive constant. Because of this, the algorithm described above is considered an improvement of the algorithm proven in theorem 6.1.

Chapter 7

Results

In the previous chapters, tools are given that are needed to obtain data of interest of the algorithm in a programming context. Later in this chapter, we pose questions that we want to answer with our simulation study. The results of this simulation study, conducted both on algorithm 1 and the new algorithm that is derived from Alon's, are given as well. We will examine strategies of both algorithms that can be interpreted in multiple ways. In the case of algorithm 1, an example of this is the choice of edges from a vertex' adjacency list that defines a path traversed to find a circuit in a bipartite graph. Earlier, several heuristics were described for this way of choosing edges and in this chapter, we will especially examine differences in performance between these heuristics.

7.1 Empirical results of simulations on the algorithm of Schrijver

Simulations are conducted on the algorithm of Schrijver in order to examine the strategy for finding circuits during the algorithm's execution. Furthermore, the theoretical complexity bound for worst case scenarios will be compared to the actual average running time for a specified subset of regular bipartite graphs. As proven earlier, the worst case bound is equal to $O(km)$ for k -regular bipartite graphs consisting of $m = \frac{1}{2}nk$ edges. For these purposes, random regular bipartite graphs are generated for simulation. More specifically, four different subsets of regular bipartite graphs, each with their own specific properties, are generated:

1. Set of 100, with $n = 500$ and $k = 10, 20, \dots, 100$, such that there are 10 graphs per k (FIXED VERTICES, DETERMINISTIC DEGREE)
2. Set of 100, with $k = 50$, and $n = 100, 200, \dots, 1000$, such that there are 10 graphs per n (FIXED DEGREE, DETERMINISTIC VERTICES)
3. Set of 100, with $n = 500$ and $k \in \mathbb{N} \cap [10, 100]$ arbitrary (FIXED VERTICES, RANDOM DEGREE)
4. Set of 100, with $k = 50$, and $n \in \mathbb{N} \cap [100, 1000]$ arbitrary (FIXED DEGREE, RANDOM VERTICES)

The strategies of interest of algorithm 1 are described as the heuristics used for finding circuits in the graph induced with the edges with nonzero weight, as using different heuristics might cause

different numbers of iterations of the algorithm for the same input graph. Specifically, the following heuristics are considered:

- FIRST: Pick the first edge from an adjacency list, or the second if the first was visited earlier.
- FIRST THREE MAXIMUM: Pick the edge with maximum weight from the first three edges of an adjacency list, or the second maximum weight if the maximum weight edge was visited earlier.
- FIRST THREE MINIMUM: Pick the edge with minimum weight from the first three edges of an adjacency list, or the second minimum weight if the minimum weight edge was visited earlier.
- FIRST THREE ALTERNATE: Alternately, pick from the first three edges in an adjacency list the edge with maximum weight (step 1) and for the next edge, pick from the first three edges from the adjacency list of the other vertex of the edge picked in step 1 the edge with minimum weight
- FIRST FIVE MAXIMUM: analogous to FIRST THREE MAXIMUM
- FIRST FIVE MINIMUM: analogous to FIRST THREE MINIMUM
- FIRST FIVE ALTERNATE: analogous to FIRST THREE ALTERNATE

Now that we have described the data input set and the heuristics to be tested, we can formalize a research question that we want to answer with the data:

- Which of the aforementioned heuristics for finding circuits in k -regular bipartite graphs should we use, such that the algorithm's performance is as efficient as possible in terms of number of steps needed until termination?

The four data sets are used to compare the performance of the different heuristics and the conclusions for each data set will be compared to come to a final conclusion on which heuristic to choose if we prioritise efficiency.

Empirical data on the heuristics are gathered through simulation and the graphics are generated by means of the statistical software R. The first piece of information that the algorithm gives that is analysed with the different heuristics for searching circuits is the increase in the sum of squares of weights on the edges scaled by the length of the found circuits after each iteration. The graphs of the subsets are used as input in algorithm 1 for each heuristic implementation of the method for finding circuits, and information of all iterations (The sum of squares of weights (SSW), increase in SSW, circuit length and number of traversed edges) of the algorithm are kept track of in a data frame per graph, and averages on the increased SSW per edge in the circuit are calculated by means of an R script. The averages are given below per heuristic in scatter plots. Furthermore, the number of traversed edges are compared per method, as this quantity also gives us a reliable measure of the order of magnitude of the number of steps needed to execute the algorithm on specific input depending on degree and number of vertices.

7.1.1 Results of data of the subsets with incrementing degree and fixed number of vertices

Let us first apply the algorithm with the different heuristic methods for finding circuits in a regular bipartite graph on the first data set: **(FIXED VERTICES DETERMINISTIC DEGREE)**. Testing the increase of the sum of squares of weights of the edges (SSW) of each graph on each method gives the following results, where each dot represents one graph:

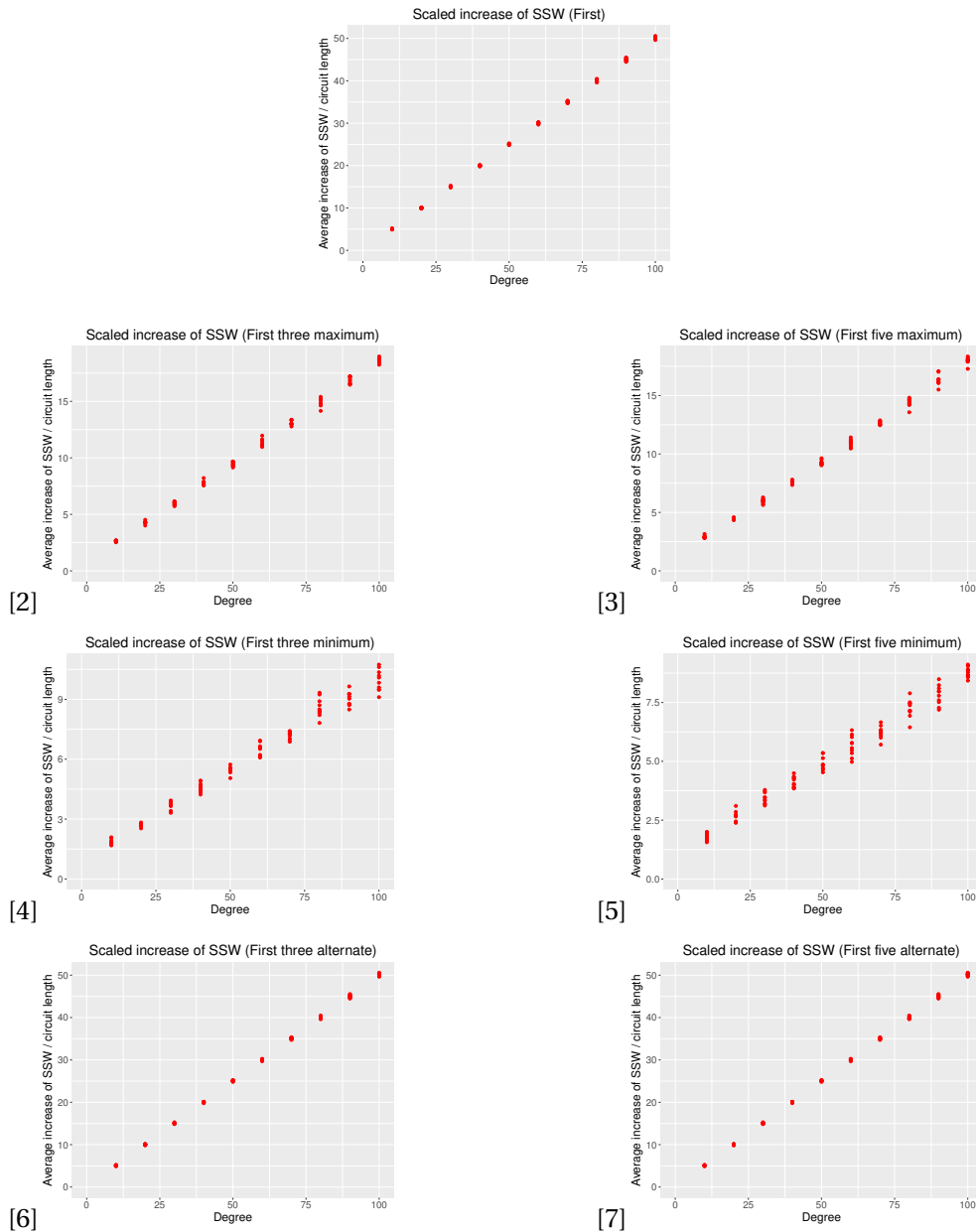


Figure 7.1: Scatter plots of the average increase in SSW per heuristic for FIXED VERTICES DETERMINISTIC DEGREE

Furthermore, a big deal of the number of operations to successfully execute the algorithm depend

on the number of traversed edges during the execution of the algorithm. The number of traversed edges is defined as the sum of the number of edges added to the circuit-finding path until a circuit was found of all iterations of the algorithm. The edges of a tail left over from a previous iteration were not taken into account in this sum. This results in the following scatter plots of the number of traversed edges for the different heuristics:

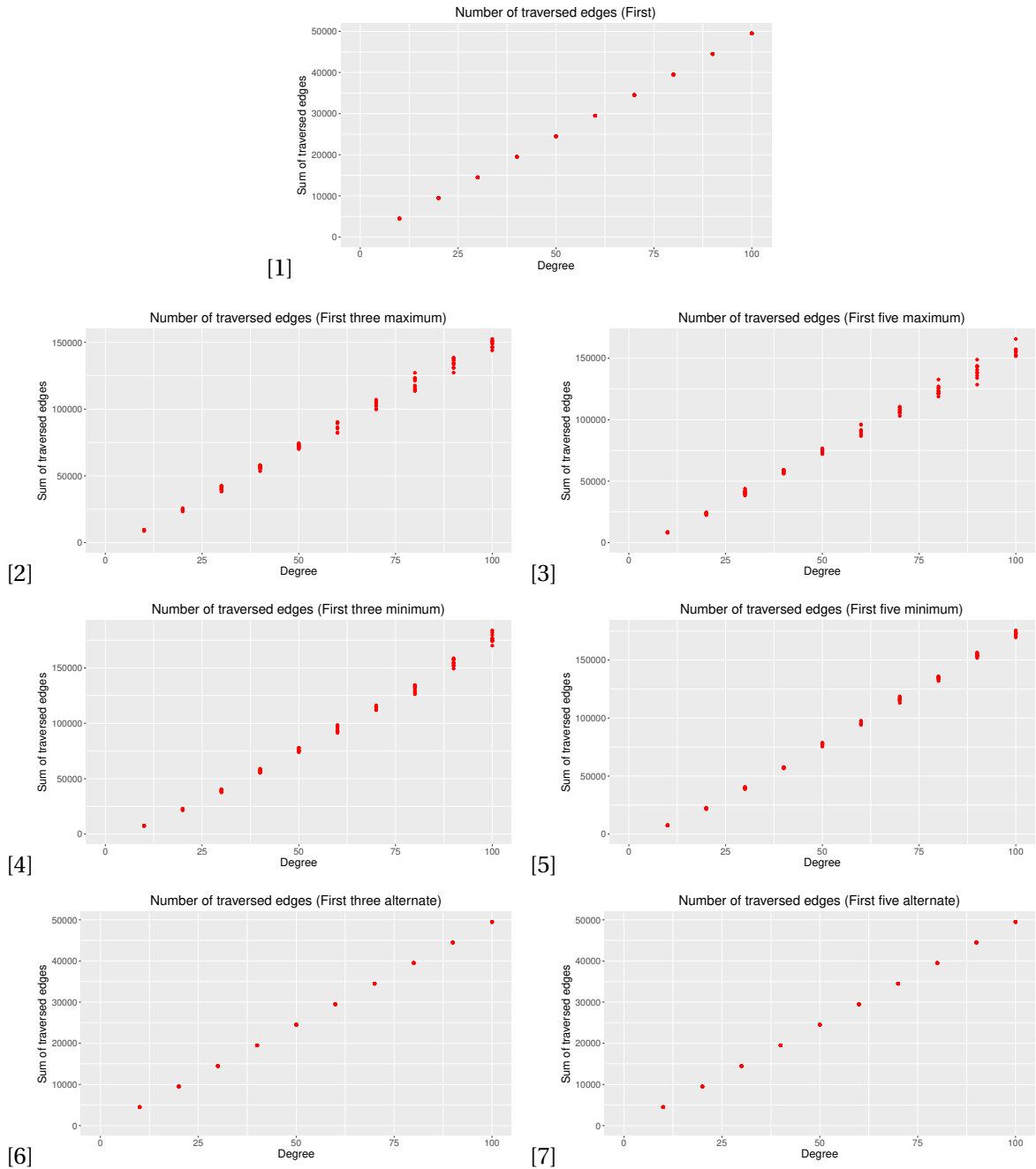


Figure 7.2: Scatter plots of the number of traversed edges in the algorithm until the desired output is obtained per heuristic (FIXED VERTICES DETERMINISTIC DEGREE)

Looking at the scatter plots above, we see that notable phenomena occur. For all seven heuristic methods, both the average increase in SSW and the number of traversed edges grow linearly in the degree of the regular bipartite graph. In theory, for a fixed number of vertices n , we would expect quadratic behavior in the number of traversed edges for increasing degree k , as the algorithm has complexity bound $O(km) = O(k^2n)$. However, the average increase in SSW grows linearly, thus it is not behaving constantly. As the average increase in SSW grows linearly, the number of traversed edges is not expected to grow quadratically in the degree anymore, but linear as is the behaviour we see. One could argue that the average running time (in practice) of the algorithm of Schrijver might not be $O(km)$, but even $O(m)$, which is as fast as the fastest worst-case scenario bound known on the problem, due to Cole, Ost and Schirra [2]. Moreover, the number of traversed edges per heuristic have differentiating orders of magnitude. For the heuristics FIRST, FIRST THREE ALTERNATE and FIRST FIVE ALTERNATE, the number of traversed edges for degree equal to 100 is more than three times as small compared to this number for the four other methods. This subset therefore gives us stronger arguments for choosing one of the three methods with the significant smaller number of traversed edges regarding efficiency of the algorithm.

7.1.2 Results of data of the subsets with incrementing number of vertices and fixed degree

The second subset (**FIXED DEGREE DETERMINISTIC VERTICES**) is analysed similarly:

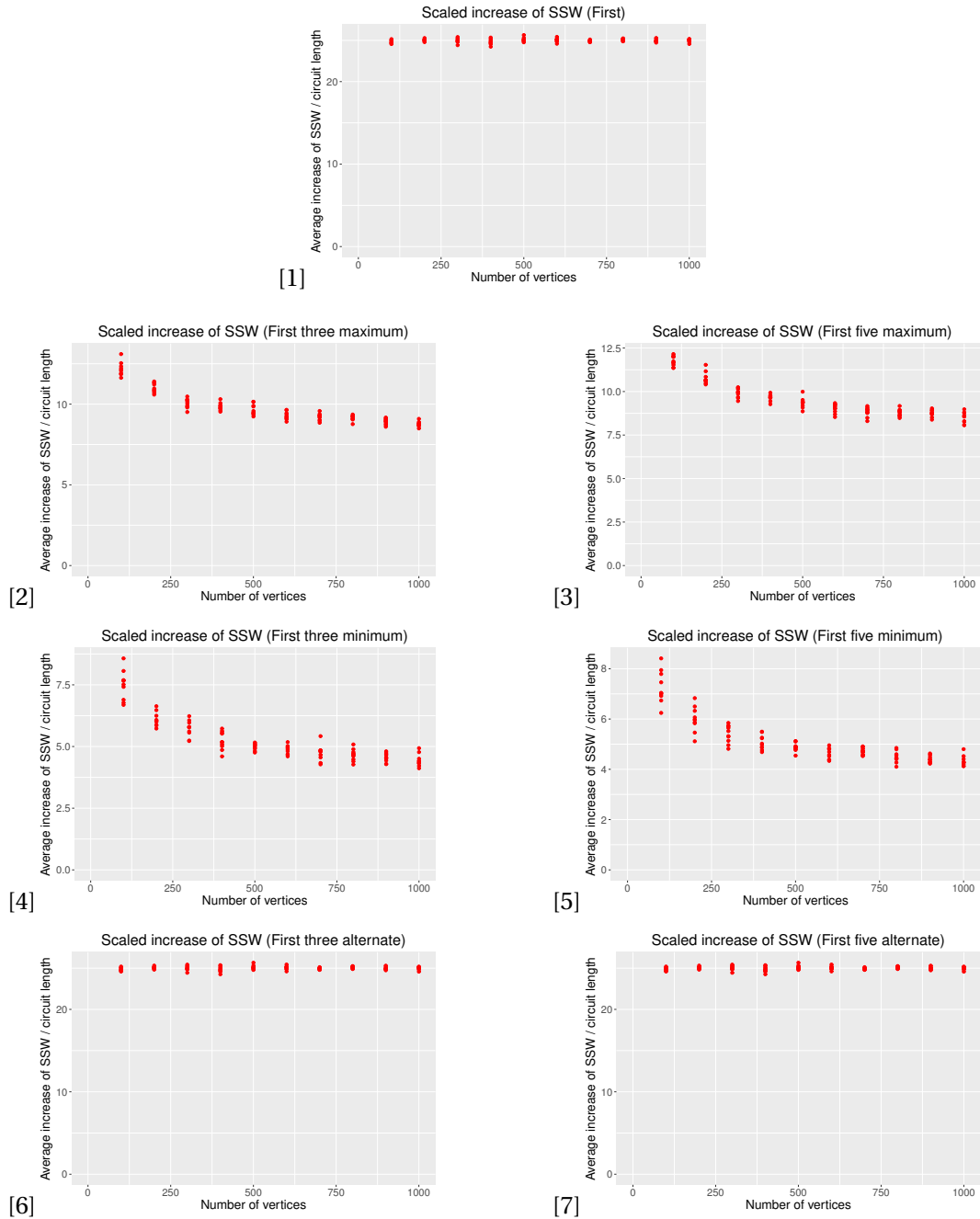


Figure 7.3: Scatter plots of the average increase in SSW per heuristic for FIXED DEGREE DETERMINISTIC VERTICES

Most surprisingly, for the second data set consisting of graphs with fixed degree and vertices in-

creasing in steps of 10, the results of the heuristics FIRST, FIRST THREE ALTERNATE and FIRST FIVE ALTERNATE coincide exactly. An explanation of this phenomenon is to be found yet and this will be a topic of interest for future research.

For the three heuristics with coinciding results, we clearly have that the increase in sum of squares of weights, scaled by length of circuits, behaves constantly as n grows and the average increase is also much larger than for the other four methods. For the other four heuristics, i.e. FIRST THREE MAXIMUM, FIRST THREE MINIMUM, FIRST FIVE MAXIMUM and FIRST FIVE MINIMUM, we see that the average increase in sum of squares of weight decreases as n grows. For relatively small n , the decrease is much faster than when n is large, because for large n , it seems like the decrease behaves linearly, while for smaller n , the decrease behaves quadratically. If we combine this with the results retrieved from the data on the number of edges used in all circuit finding paths, we will get an indication of the performance of the algorithm in terms of the number of vertices n and the degree k .

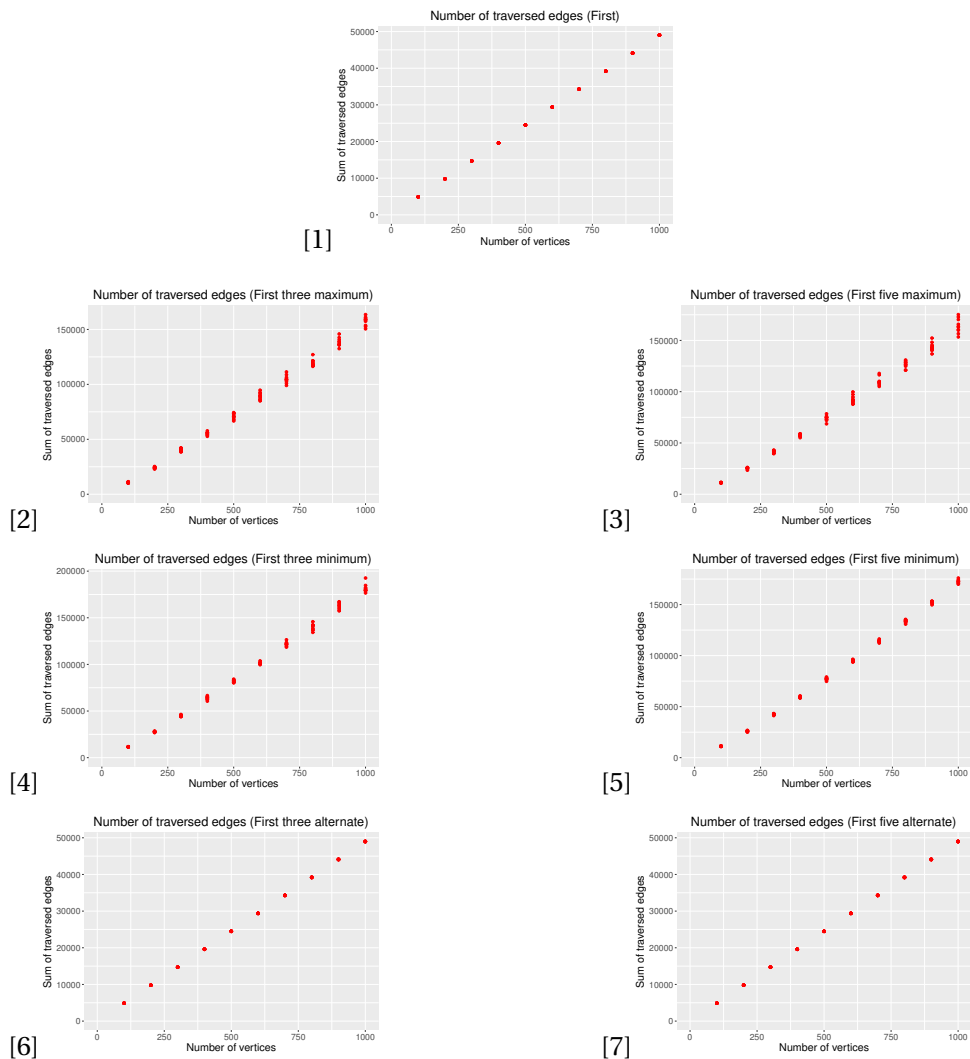


Figure 7.4: Scatter plots of the number of traversed edges in the algorithm until the desired output is obtained per heuristic (FIXED DEGREE DETERMINISTIC VERTICES)

For all different heuristics, the behaviour of the number of traversed edges is clearly linear in the number of vertices. For the heuristics that have a constant average increase of sum of squares of weights, the behaviour of the number of traversed edges is almost perfectly linear in the number of vertices, while the behaviour of the number of traversed edges for the other four heuristics shows a bigger variance when n gets large enough. Furthermore, as the average increase of sum of squares of weights decreases for the other four heuristics, this directly means that the number of traversed edges is much larger, looking at the windows of the y -axes of the scatter plots. The results above again imply that the most efficient heuristic strategies for finding circuits in regular bipartite graphs are given by the heuristics FIRST, FIRST THREE ALTERNATE and FIRST FIVE ALTERNATE, with no pairwise difference between these three whatsoever.

7.1.3 Results of data of the subsets with random degree and fixed number of vertices

We will again perform the same empirical analysis as described in the previous subsection. We begin with scatter plots of the subset **FIXED VERTICES RANDOM DEGREES**:

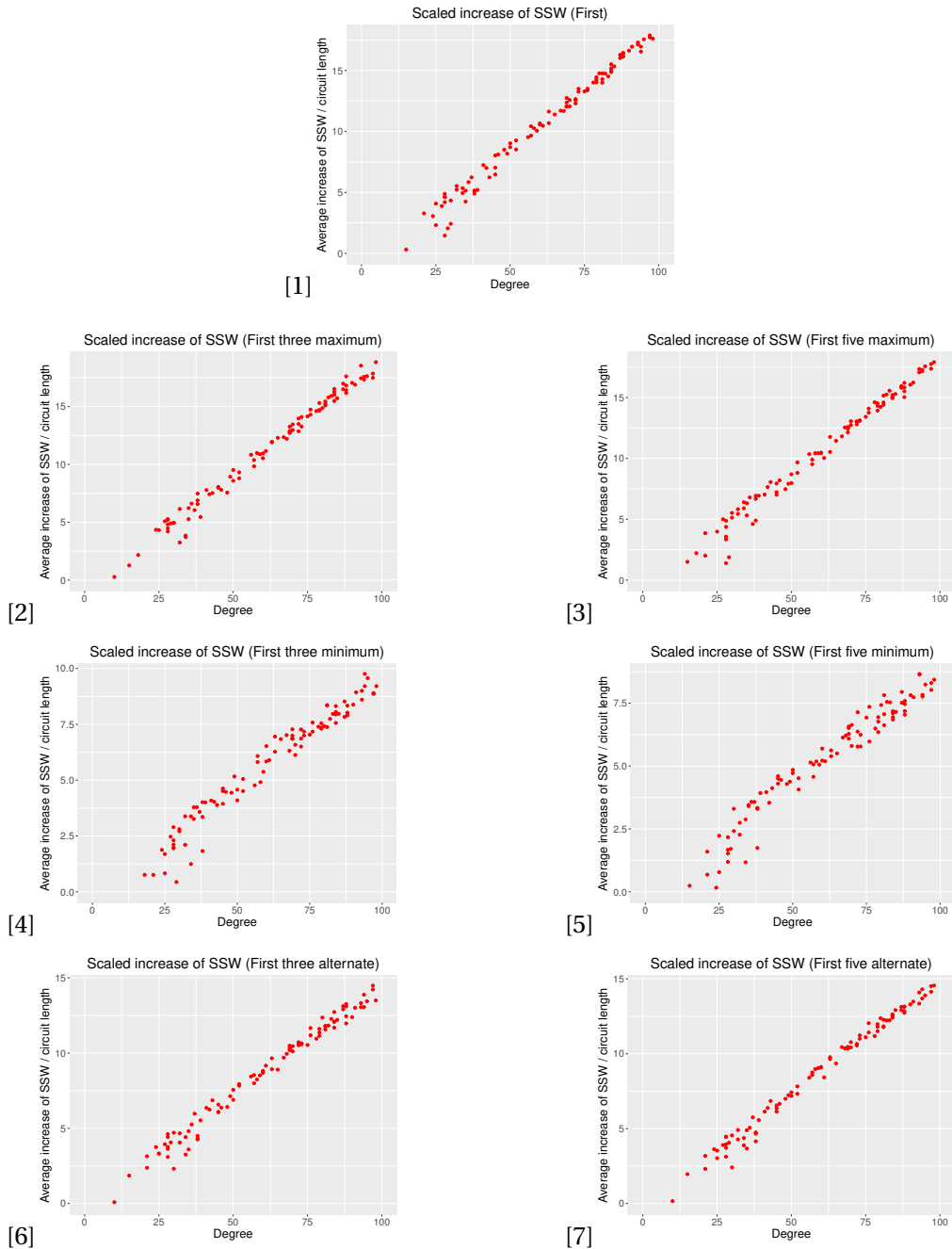


Figure 7.5: Scatter plots of the average increase in SSW per heuristic for FIXED VERTICES RANDOM DEGREES

In all seven methods, we clearly see a linear trend in the average increase in SSW per heuristic in the

degree of the regular bipartite graphs. The average increase of SSW is smaller for the four heuristics involving looking at minimum weight edges and alternating maximum and minimum weight edges, i.e. FIRST THREE MINIMUM, FIRST FIVE MINIMUM, FIRST THREE ALTERNATE and FIRST FIVE ALTERNATE. Results from the number of traversed edges will be needed to draw strong conclusion from these results:

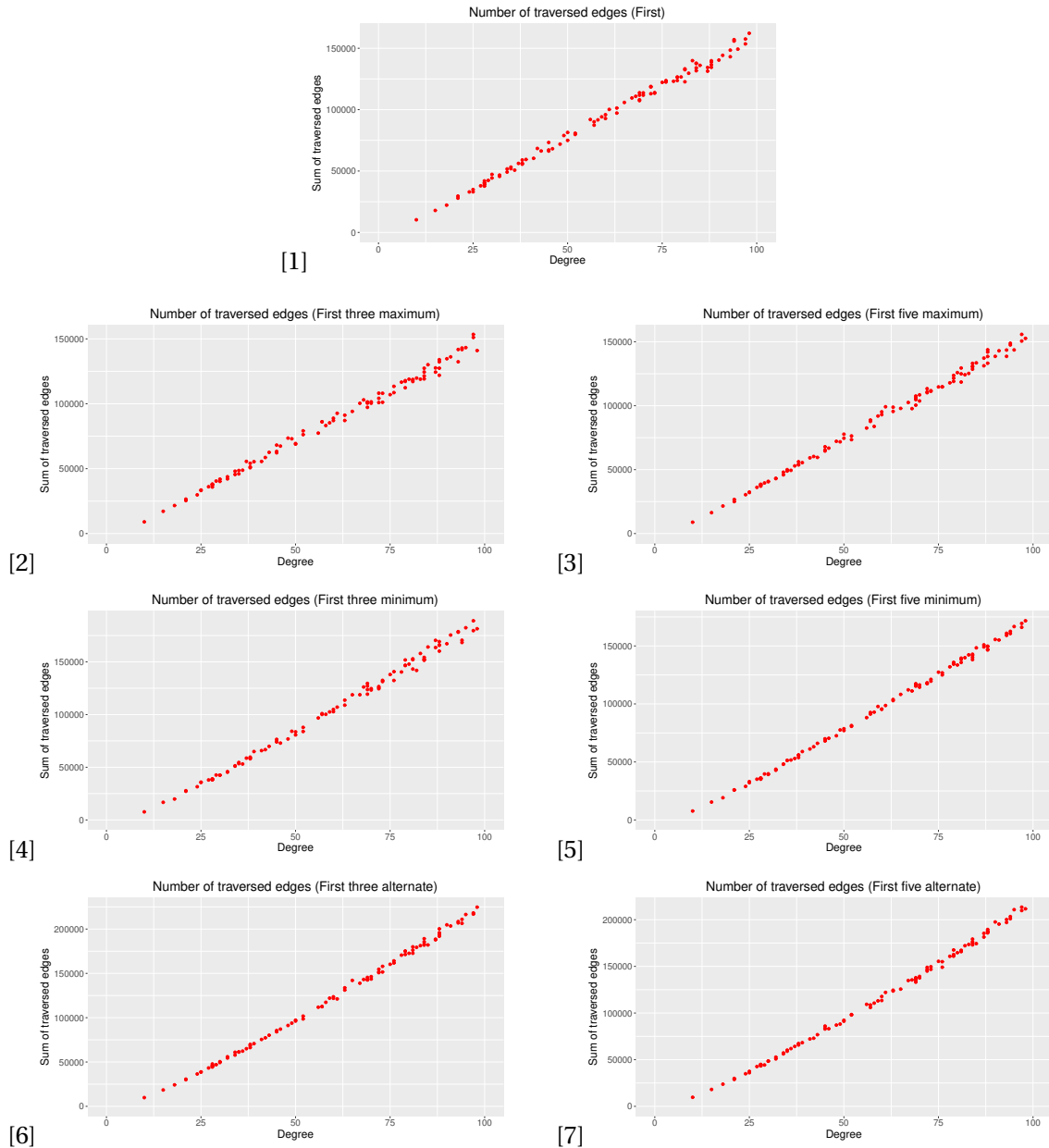


Figure 7.6: Scatter plots of the number of traversed edges in the algorithm until the desired output is obtained per heuristic (FIXED VERTICES RANDOM DEGREES)

Again, we see a linear trend in both the average benefit in SSW and the number of traversed edges for a subset with a fixed number of vertices. This makes the need for investigating whether or not the average running time is $O(m)$ even stronger, as we can draw the same conclusions from these

data as from the first data set `FIXED VERTICES DETERMINISTIC DEGREES`. However, the number of traversed edges is now much larger for the heuristic methods `FIRST THREE ALTERNATE` and `FIRST FIVE ALTERNATE`, so this gives evidence to thinking that `FIRST` is the method that should be chosen when the number of vertices n is fixed, although the performance in terms of number of traversed edges of `FIRST THREE MAXIMUM` and `FIRST FIVE MAXIMUM` is similar for random k .

7.1.4 Results of data of the subsets with random number of vertices and fixed degree

The data set FIXED DEGREE RANDOM VERTICES yields us the following results:

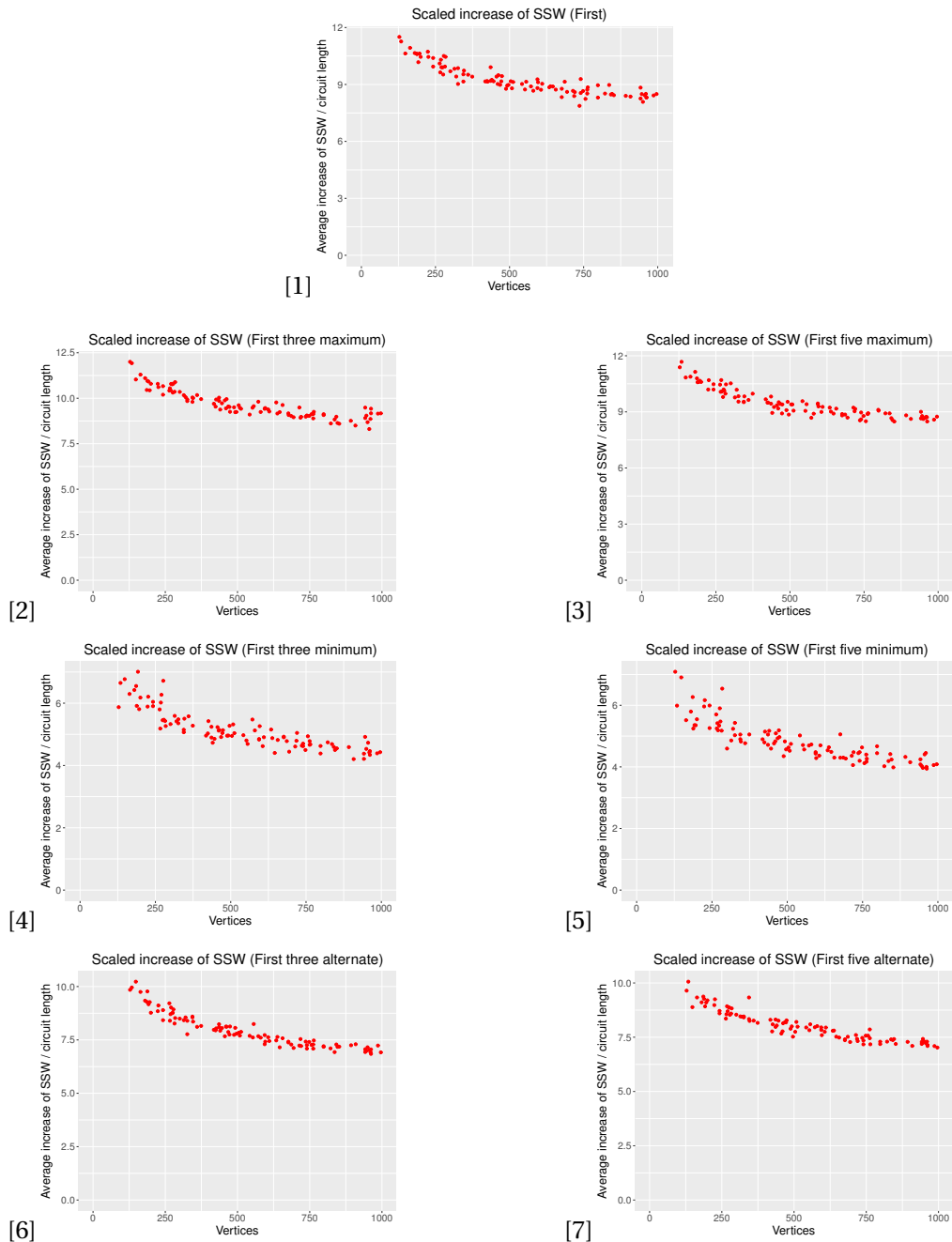
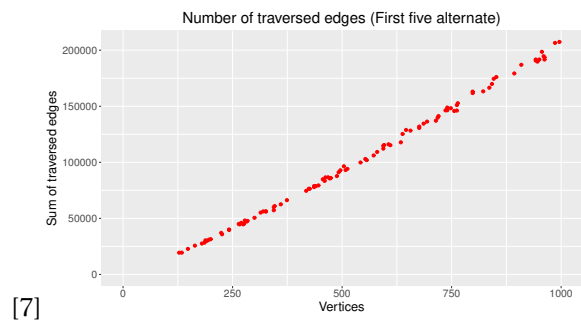
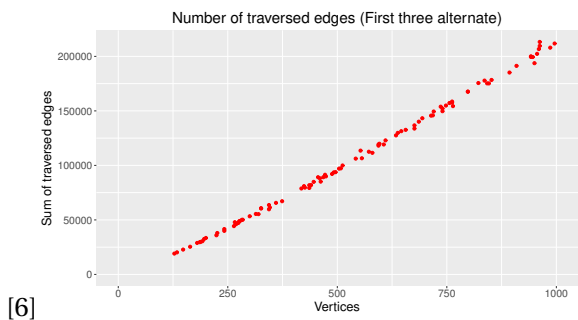
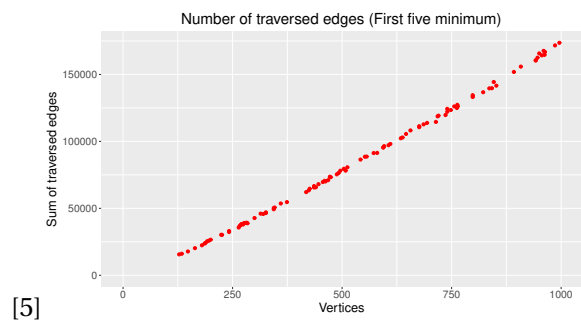
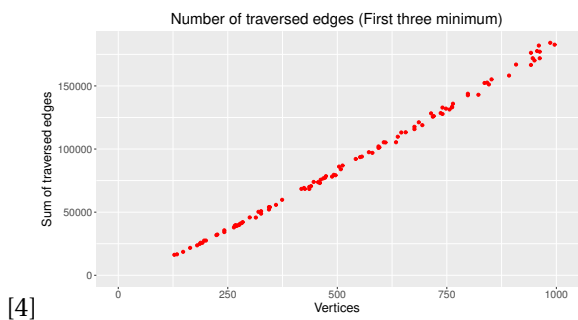
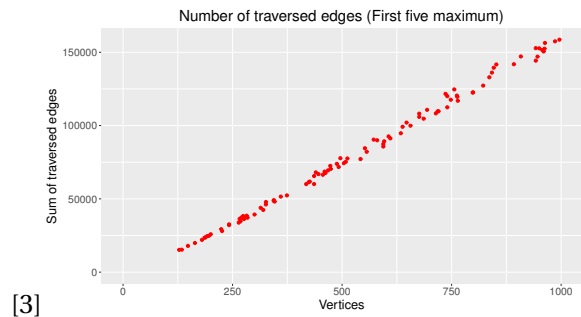
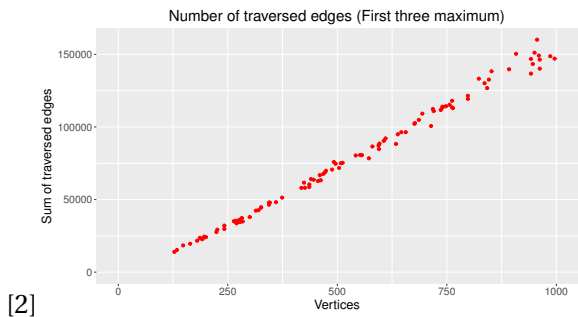
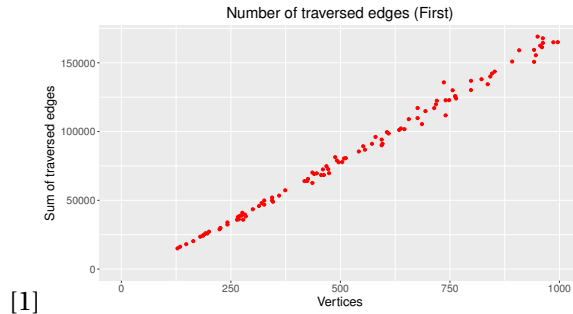


Figure 7.7: Scatter plots of the average increase in SSW per heuristic for FIXED DEGREE RANDOM VERTICES

In all seven methods, we clearly see the same trend in the average increase in SSW per heuristic for increasing number of vertices of the regular bipartite graphs as in the second data set FIXED

DEGREE DETERMINISTIC VERTICES. However, the average increase is slightly bigger for the two heuristics FIRST THREE MAXIMUM and FIRST FIVE MAXIMUM than for FIRST. Again, we will need results from the number of traversed edges to draw strong conclusions from these results:



Again, we can observe a linear trend in the number of traversed edges for all seven methods. We see that the heuristics FIRST THREE MAXIMUM and FIRST FIVE MAXIMUM slightly outperform FIRST in terms of number of traversed edges. In practice however, FIRST will be faster, as in the other two heuristics, every time that an edge needs to be picked, three or five edges need to be considered. So, also for fixed k , the heuristic FIRST is the most efficient in use.

7.1.5 Summary

The empirical data gave a lot of information on the behaviour of the algorithm on the number of operations that are needed for successfully executing the algorithm. For fixed number of vertices and increasing degree, we saw that the algorithm is almost perfectly linear for the heuristics FIRST, FIRST FIVE ALTERNATE and FIRST THREE ALTERNATE when we look at the number of edges that are added to the circuit-searching paths of the algorithm. For fixed degree and increasing number of vertices, the average increase of SSW is monotone decreasing, while the number of traversed edges behaves linearly for increasing n . As the number of traversed edges behaves linear for both degree k and number of vertices n , and regular bipartite graphs only have these two properties defining the size of the graph, the results give space to thinking that the average running time is in fact $O(m)$, as fast as the fastest methods available. Both for fixed k , increasing n and fixed n , increasing k , the most efficient strategy for the seven proposed heuristics is FIRST.

7.2 Empirical results of Alon's and new algorithms

The previous descriptions of Alon's [1] and the two new algorithms, respectively given by theorem 6.1 and algorithm 5, are also inviting to perform empirical analyses on the order of magnitude of the number of steps needed to terminate the algorithms. As the three algorithms are closely related, comparison between the three algorithms is a logical next step for the research. All three algorithms make use of only adding (partially) dummy matching and Euler splits, which makes the data structures involved much simpler than the data structures needed for the algorithm of Schrijver. As a measure of the number of steps that it takes before termination of the algorithm, in the case of Euler splits we say that the algorithm 'traverses' all edges in the regular bipartite graph that is split. This number of traversed edges is again our quantity of interest. The same data sets are used as the ones used for conducting the empirical research on the heuristic methods that could be used in Schrijver's algorithm. A question that we want to see answered by this research is as follows:

- Although the complexity bounds of Alon's and both own algorithms are the same, what is the difference in the practical running time of the three algorithms?
- Do the own algorithms really deem more efficient compared to Alon's?

Once again, depending on the data set that the three algorithms are tested for, the number of traversed edges is plotted against either the degree of the graph or the number of vertices. The algorithm induced by theorem 6.1 is referred to as version 1, and algorithm 5 is referred to as version 2. This yields the following results for the four data sets:

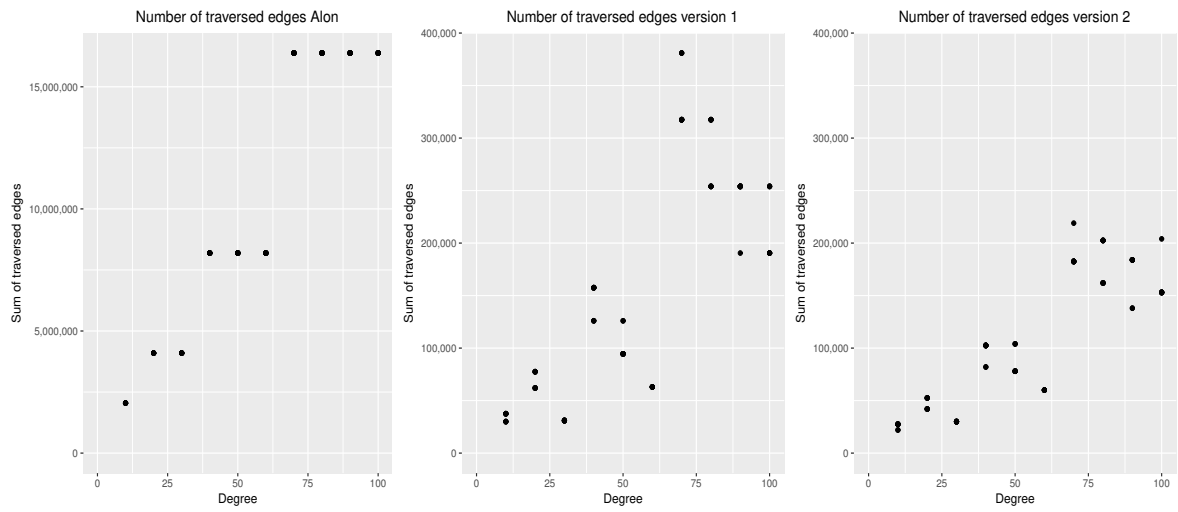


Figure 7.8: Number of traversed edges for Alon, version 1 and version 2 on the data set FIXED VERTICES DETERMINISTIC DEGREES.

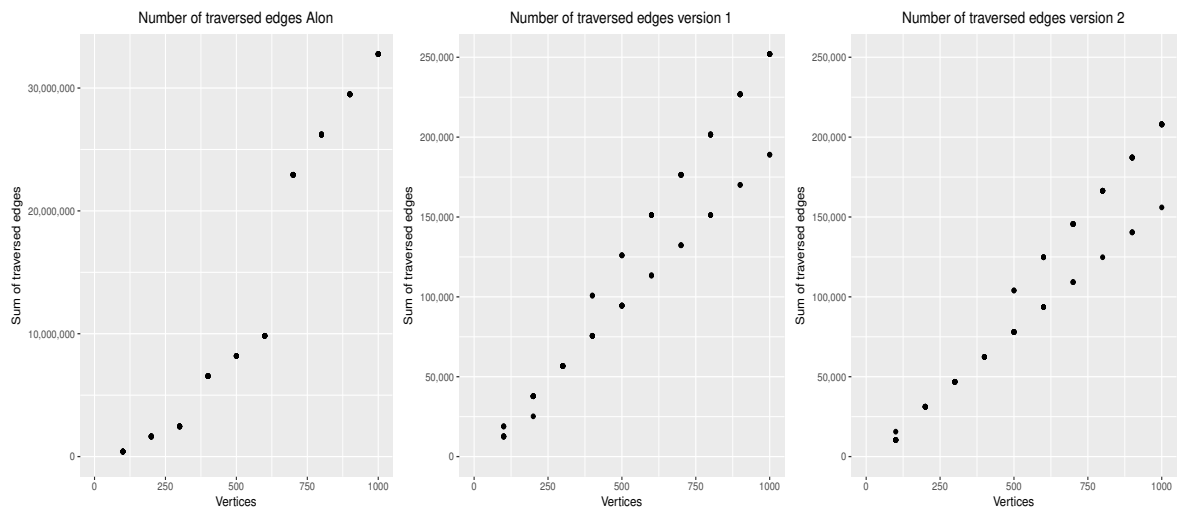


Figure 7.9: Number of traversed edges for Alon, version 1 and version 2 on the data set FIXED VERTICES DETERMINISTIC VERTICES.

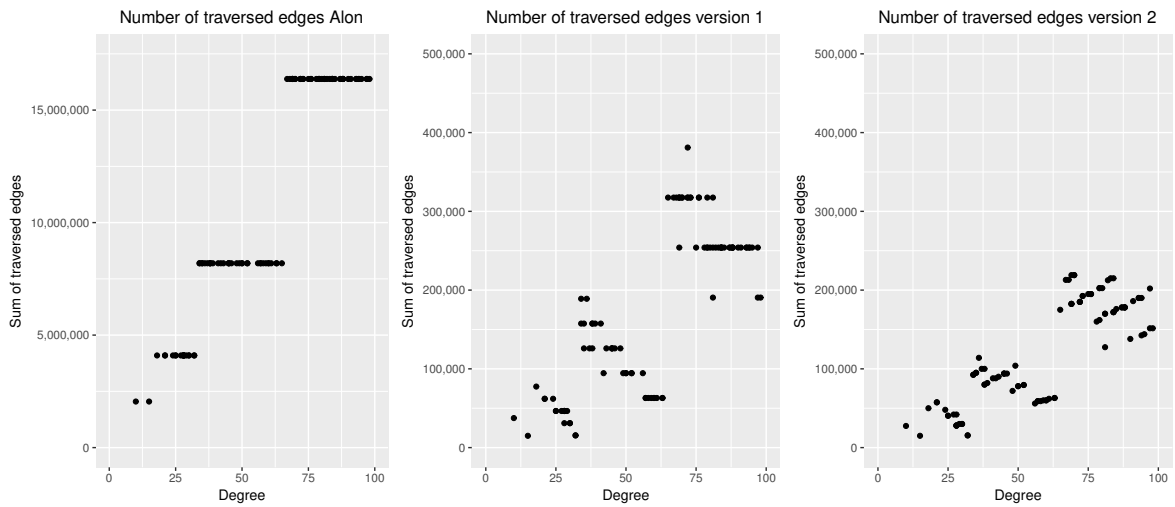


Figure 7.10: Number of traversed edges for Alon, version 1 and version 2 on the data set FIXED VERTICES RANDOM DEGREES.

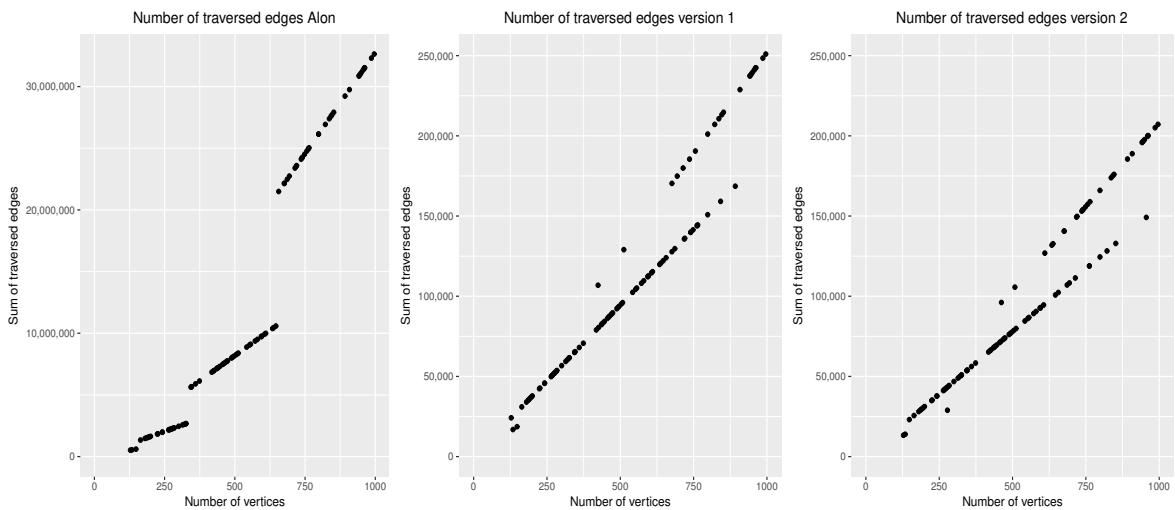


Figure 7.11: Number of traversed edges for Alon, version 1 and version 2 on the data set FIXED DEGREE RANDOM VERTICES.

Let us first discuss the results of the two data sets with a fixed number of vertices. In these data sets, the number of traversed edges in Alon's algorithm does not directly relate to the degree of a graph, but to 2^t , where $2^t \geq k$ and t as small as possible. For version 1 and version 2, the number of traversed edges directly relates to the degree k , and the number of traversed edges is much larger if a lot of (partially) dummy matchings need to be added to the graph. In version 1 this happens for degrees k that are a tiny bit larger than a power of two, compliant to the theory. This explains why the number of traversed edges is sometimes larger even for smaller k . This effect is not as significant for version 2, but still visible in the results. We observe that the order of magnitude of version 1 and version 2 do not scale with Alon's algorithm, as Alon's algorithm traverses significantly larger numbers of edges, often with a factor even larger than 50.

The two data sets with a fixed degree show a completely different behaviour. For Alon's algorithm, between each two powers of two of $n \log n$ we see a strong linear relation when the number of vertices increases. The slopes of these linear trends tend to increase significantly when m becomes a power of two. For version 1 and version 2, two different trends can be observed. Finding a reasonable explanation for the existence of multiple trends will be of interest when conducting future research. Notice again that the windows on the y -axis are not in the same order of magnitude when we compare the graphs of Alon's algorithm and the graphs of the other two algorithms. This means that on all data sets tested, version 1 and version 2 seem to be very significant improvements of the already existing algorithm by Alon.

Chapter 8

Conclusion

The goal of this report was to do more thorough analysis on the perfect matching algorithm of Schrijver (algorithm 1) and try to find variations or completely new algorithms that outperform it in a certain way. The two methods we proposed are asymptotically faster for regular bipartite graphs for which $\log n < k$. Nevertheless, this conclusion is drawn based only on worst-case bounds. From the empirical analysis, algorithm 1 seems to have complexity $O(m)$ in practice, making the algorithms proposed in this report unnecessary. However, we observe that the new algorithms are very simple and an improvement to the method of Alon's, specifically when we look at the performance in practice.

As research is conducted especially on relatively sparse graphs, the data structures used are adjacency lists, specifically speaking doubly linked lists, due to their low complexity for adding and removing edges from graphs. According to empirical analysis, we can furthermore conclude that finding circuits in regular bipartite graphs as a subroutine of algorithm 1 is most efficiently done by always taking the first edge of a vertex' adjacency list.

Also, we dedicated a part of our report to edge coloring bipartite graphs. The conclusion we can draw about this topic is that the method by Kapoor and Rizzi needs to execute a perfect matching algorithm at most once, rendering it a very efficient method, not only for small degrees k .

Nevertheless, most of the conclusions are based on empirical analysis. Points of interest for future research is trying to find a mathematical explanation or proof for the complexity of algorithm 1 being $O(m)$. Also, we still want to find out the reason why the results of the heuristics FIRST, FIRST, FIRST THREE ALTERNATE and FIRST FIVE ALTERNATE coincided for the subset with fixed vertices and incrementing degree. Finally, also an explanation for the double linear trend in the algorithms we proposed is yet to be found.

Bibliography

- [1] N. ALON, *A simple algorithm for edge-coloring bipartite multigraphs*, Inf. Process. Lett., 85 (2003), pp. 301–302.
- [2] R. COLE, K. OST, AND S. SCHIRRA, *Edge-coloring bipartite multigraphs in $O(E \log D)$ time*, Combinatorica, 21 (2001), pp. 5–12.
- [3] J. CSIMA AND L. LOVÁSZ, *A matching algorithm for regular bipartite graphs*, Discrete Applied Mathematics, 35 (1992), pp. 197–203.
- [4] H. N. GABOW, *Using euler partitions to edge color bipartite multigraphs*, International Journal of Computer & Information Sciences, 5 (1976), pp. 345–355.
- [5] M. GOLDWASSER, M. GOODRICH, AND R. TAMASSIA, *Fundamental Data Structures*, in Data Structures and Algorithms in Java, John Wiley & Sons, Inc., 6 ed., 2014, ch. 3, pp. 104–148.
- [6] P. HALL[†], *On Representatives of Subsets*, Journal London Mathematical Society, 10 (1935), pp. 26–30.
- [7] J. E. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM Journal on Computing, 2 (1973), pp. 225–231.
- [8] A. KAPOOR AND R. RIZZI, *Edge-coloring bipartite graphs*, Journal of Algorithms, 34 (2000), p. 390–396.
- [9] D. E. KNUTH, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [10] D. KÖNIG, *Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre*, Mathematische Annalen, 77 (1916), pp. 453–465.
- [11] A. SCHRIJVER, *Bipartite edge-colouring in $O(m)$ time*, Siam J. Comput., 28 (1998), pp. 1–7.