Experimental study of the roots of graph polynomials

Diana-Andreea Tanase

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2019

Abstract

This paper presents through experimental studies the distribution of roots of graph polynomials associated with b-matchings of regular graphs. It aims to acknowledge the existence of a zero-free region containing the $[0,1]$ interval implying that polynomial-time algorithms could be used to approximate Holant problems. Comprehensive focus is devoted to the algorithms to generate regular graphs, inspired by well-known techniques with this purpose, and the algorithms to compute global polynomials, that combine already-established and novel approaches.

One of the main contributions of this paper is describing the implementation of a recursive algorithm for generating global polynomials that has allowed experiments for graphs of up to 20 vertices of degree at most 11. Although the work has been mostly devoted to matchings of size less or equal to 5, a limited number of polynomials have been considered for 6, 7 and 8-matchings.

Analysis has confirmed the existence of a zero-free region by applying the bounds on the arguments of the roots suggested by Lebowitz, Pittel, Ruelle and Speer in [LPRS16] and has discussed the distribution of the roots close to the origin, providing as well examples for polynomials with complex roots in the right-half plane. Furthermore, similar patterns of the distribution of the roots have been observed and shown through various plots.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Heng Guo, for his valuable guidance and insightful feedback. I would also like to thank my family and friends for their constant moral support throughout my academic years.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Counting problems represent an expressive framework that addresses a large variety of combinatorial questions in many fields, such as the number of matchings, cycles or edge colourings in graph theory [Val79b] or the density of states in statistical mechanics [Ver96]. For expressing the computational complexity of computing the number of solutions of a counting problem associated with an *NP* search problem, Valiant has introduced in [Val79a] the complexity class denoted by *#P*. Counting problems are known to be computationally expensive, with a good summary of their complexity being given in [Val79b]. Finding the number of k-edge colourings for a d-regular graph, for example, is *#P*-hard when $k \geq d \geq 3$, as proved in [CGW16b], meaning that there is no known efficient (polynomial-time) algorithm that solves the problem.

Combinatorial counting problems are often associated with graph polynomials, defined in 2.4 and known as special cases of Holant Problems [CGW16a]. Holant is a partition function on graphs, represented as a sum of products, where edges are variables and vertices are constraint functions. In various papers, such as [CGW16b] and [GLLZ], the Holant of a graph $G = (V, E)$ with $\pi : V \to F$ an assignment from the set of vertices V to a set of functions F and $f_v = \pi(v)$ a constraint function $\{0,1\}^{deg(v)} \to \mathbb{C}$ associated with the vertex v is defined as:

$$Z(G, \pi) = \sum_{\sigma \in \{0,1\}^{|E|}} \prod_{v \in V} f_v(\sigma|_{E(v)}),$$

where $E(v)$ is the set of adjacent edges of v, $\sigma|_{E(v)}$is the restriction of σ on E(v) and $deg(v) = |E(v)|$ is the degree of vertex v.

To illustrate the Holant formula on a short example, consider a complete graph of 3 vertices (a triangle with exactly one edge between every two vertices), with $V = \{0,1,2\}$ the set of vertices, and a symmetric binary function $f : \{0,1\}^2 \to \{0,1\}$ such that:

$$\pi(v) = f, \text{ for every } v \in V.$$

As f is symmetric,

$$f(01) = f(10).$$

If $E(v) = (e_i, e_j)$ is the set of edges of vertex v, with $i \neq j$ and $i, j \in \{0,1,2\}$, and $\sigma = \sigma_0\sigma_1\sigma_2 \in \{0,1\}^3$, then:

$$\sigma|_{E(v)} = \sigma_i\sigma_j \in \{0,1\}^2.$$

The Holant value is:

$$Z(G,\pi) = \sum_{\sigma\in\{0,1\}^3} f(\sigma|_{E(0)})f(\sigma|_{E(1)})f(\sigma|_{E(2)}) \Rightarrow$$

$$Z(G,\pi) = f(000|_{E(0)})f(000|_{E(1)})f(000|_{E(1)}) + \sum_{\sigma\in\{001,010,100\}} f(\sigma|_{E(0)})f(\sigma|_{E(1)})f(\sigma|_{E(2)}) +$$

$$+ \sum_{\sigma\in\{011,101,110\}} f(\sigma|_{E(0)})f(\sigma|_{E(1)})f(\sigma|_{E(2)}) + f(111|_{E(0)})f(111|_{E(1)})f(111|_{E(1)}) \Rightarrow$$

$$Z(G,\pi) = f(00)^3 + 3f(00)^2f(01) + 3f(01)^2f(11) + f(11)^3.$$

Holant is a generalization of *counting* constraint satisfaction problems [CLX09], in which variables must satisfy a number of constraints. It can express problems such as counting perfect matchings, where every vertex of the graph is matched with exactly one edge. Counting perfect matchings is not expressible as a counting constraint satisfaction function, as proved in [FLS07] for graphs with real-valued weighted vertices and in [CG19] for graphs with complex weights. However, with the right choice of the constraint function, the Holant counts the number of perfect matchings in G [CLX09].

For problems that are #P-hard, such as counting the number of perfect matchings in an arbitrary graph [Val79a], the Holant is, in general, hard to compute (#P-hard) as well. Papers such as [GLLZ] and [PR17] present polynomial-time algorithms that approximate Boolean Holant problems for a various constraints, such as matchings, even subgraphs and edge covers. The existence of these approximation algorithms depends on the absence of zeros of graph polynomials in a certain disk centered at the origin [PR17]. Moreover, with a theoretical focus on d-regular graphs, it has been shown in [GLLZ] that, if the graph polynomial is zero-free in a strip containing $[0,1]$, then a series of transformations can be applied to obtain the necessary polynomial that is zero-free in an origin-centered disk, implying the existence of an approximation algorithm for the Holant problem it represents.

These results have motivated gaining a better understanding of the roots of graph polynomials. Experiments could be carried out for different graphs and signature functions that could establish a starting point for more theoretical research in this area. However, there is no knowledge of any previous experimental studies to generate the polynomials and analyse their roots. We have hence decided to direct our attention towards b-matchings on regular graphs, a function that constraints vertices to be matched with at most b edges (defined din detail in Section 2.3). A polynomial-time randomized approximation scheme for counting the number of b-matchings for $b \leq 7$, using the Markov Chain Monte Carlo method [Gey11], has been developed and presented in [HLZ16], but [GLLZ] makes no reference to approximation algorithms using Holant zeros for this counting problem.

1.2 Goals and achievements

The project aims to analyse the roots of graph polynomials resulted from regular graphs constrained by b-matchings, expressing a Boolean Holant problem. The research has been motivated by the interest in validating the existence of a zero-free region containing the [0,1] interval, that would imply that polynomial-time approximation algorithms could be applied to solve the associated question of counting the number of b-matchings.

This paper presents the results of experimental studies on the distribution of the roots of regular graph polynomials of b-matchings. The roots of polynomials of regular graphs of up to 20 vertices have been generated for up to 8-matchings and empirical observations have been made on the pattern of the distribution of the roots. We have, in particular, compared the argument of the roots against a bound presented in Section 2.6, from which the existence of a zero-free [0,1] region is inferred.

In order to achieve the goal of analysing the roots of polynomials, the following major steps have been completed first:

- **Generate random regular graphs**. Aiming for representative results, an efficient algorithm to generate random regular graphs (defined in Section 2.2) was necessary. Inspired by well-known methods and research in the area, different approaches have been implemented and compared to decide which is the most appropriate to use for the task. In the end, the polynomials resulting from a total of of 295 non-isomorphic graphs have been generated and analysed.
- **Generate global polynomials**. The implementation of an efficient algorithm to generate the required global polynomials (defined in Section 2.4) has played a fundamental role towards accomplishing the goal. The lack of similar work in the area has represented a challenge, as there were no previous implementations to be used as guidance. Four algorithms have been implemented and compared and the most efficient one in terms of time complexity has been chosen. This algorithm has been used to generate a total of 800 different polynomials, with some of the results being computed using servers for computationally demanding jobs, provided by the School of Informatics, University of Edinburgh.

1.3 Structure of the report

The present reports follows the development of the project. Introductory theory on polynomials and graph theory is presented in the next chapter, extended by explanatory information on signature functions and the local and global polynomials of a graph. Chapter 3 explains the decisions on programming language and libraries used and presents in detail the algorithms for generating random regular graphs and graph polynomials. Chapter 4 describes the experiments that were performed and analyses the resulting polynomials and their roots. In the end, Chapter 5 summarises the undertaken work and highlights the goals considered for the second part of this project.

Chapter 2

Background

This chapter introduces definitions and observations related to the theoretical aspects required for achieving the tasks described in Section 1.2, building from elementary information on polynomials and graphs to the more advanced topics of signature functions and graph polynomials.

2.1 Polynomials

Before describing the computational process of generating the graph polynomials that make the scope of this project, we firstly present some introductory theory on polynomials, while the notion of graph polynomials is described later in Section 2.4.

Definition 2.1.1. *Given a non-negative integer d, an* ***univariate polynomial in x of degree d*** *is a function p(x) of the form*

$$p(x) = \sum_{i=0}^{d} c_i x^i,$$

where the constants c_0, c_1, ... c_d *are the* ***coefficients*** *of the polynomial with* $c_d \neq 0$ [RS02]. An univariate polynomial is a polynomial in one variable.

In Section 3.6.2, it is explained how multivariate polynomials can be used to obtain the final univariate graph polynomial. For convenience, in the rest of the paper, a graph polynomial refers to a univariate polynomial.

Definition 2.1.2. *A* ***multivariate polynomial*** *is a polynomial in more than one variable.*

Example 2.1.1. *p(x,y)* below is a multivariate polynomial:

$$p(x,y) = c_{22}x^2y^2 + c_{21}x^2y + c_{12}xy^2 + c_{11}xy + c_{10}x + c_{01}y + c_{00}.$$

Definition 2.1.3. *A* ***root*** *(or* ***zero****) of the univariate polynomial p(x) is the complex number* α *such that p(*α*)=0.*

2.2 Graph theory

Graphs represent a widely researched and of high importance topic in many areas, due to their ability to model relationships between objects. In technology, in particular, graphs are applied to a variety of tasks, such as resource allocation in Operating Systems [MCOD06] or nodes communication in Computer Networks [Kur05]. This section presents some definitions related to the graph abstract data type known in Computer Science, with an emphasis on introducing regular and connected graphs.

Definition 2.2.1. *A **directed graph** (or **digraph**) G is a pair (V, E), where V is a finite set and E is a binary relation on V. The set V is called the vertex set of G, and its elements are called **vertices** (singular: vertex). The set E is called the edge set of G, and its elements are called **edges*** [RS02]. In a directed graph, edges have orientations. An arrow *(x,y)* is considered to be a directed edge from x to y.

Definition 2.2.2. *In an **undirected graph** G = (V, E), the edge set E consists of unordered pairs of vertices, rather than ordered pairs. That is, an edge is a set $\{u, v\}$, where u,v∈V and $u \neq v$* [RS02]. An undirected graph is a graph in which edges have no orientation.

Note that the terms *vertex* and *node* can be used interchangeably. By convention, the pair notation *(u,v)* is used for an edge, rather than the set notation $\{u, v\}$, and *(u,v)* and *(v,u)* are considered to be the same edge. Moreover, to describe the size of the two sets V and E, the notations $|V| = n$ and $|E| = m$ have been generally adopted.

For the experiments presented in this paper, only simple graphs have been considered, more specifically simple connected $d-regular$ graphs.

Definition 2.2.3. *If (u, v) is an edge in a graph G = (V, E), then the vertex v is **adjacent** to vertex u and u and v are **neighbours*** [RS02]. When the graph is undirected, the adjacency relation is symmetric.

Definition 2.2.4. *If (u, v) is an edge in a directed graph G=(V, E), then (u, v) is **incident** from or leaves vertex u and is incident to or enters vertex v* [RS02].

Definition 2.2.5. ***Multiple edges** are two or more edges that are incident to the same two vertices.*

Definition 2.2.6. *A **loop** is an edge that connects a vertex to itself.*

Definition 2.2.7. *A **simple graph** is an undirected graph with neither multiple edges nor loops.*

Definition 2.2.8. *The **degree** of a vertex in an undirected graph is the number of edges incident on it.*

Observation 2.2.1. In a simple graph with n vertices, the degree of every vertex is at most $n-1$.

Definition 2.2.9. *A **regular graph** is a graph in which each vertex has the same number of neighbours, so very vertex has the same degree.* A regular graph with vertices of degree d is called a $d-regular\ graph$ or regular graph of degree d.

Observation 2.2.2. The necessary and sufficient conditions for a $d-regular$ graph of order n to exist are that $n \geq d+1$ and that $n \cdot d$ is even [Gan18].

Observation 2.2.3. The number of edges of a d-regular undirected graph with n vertices is

$$|E| = \frac{n \cdot d}{2}.$$

Three examples of regular graphs are provided in Figure 2.1.

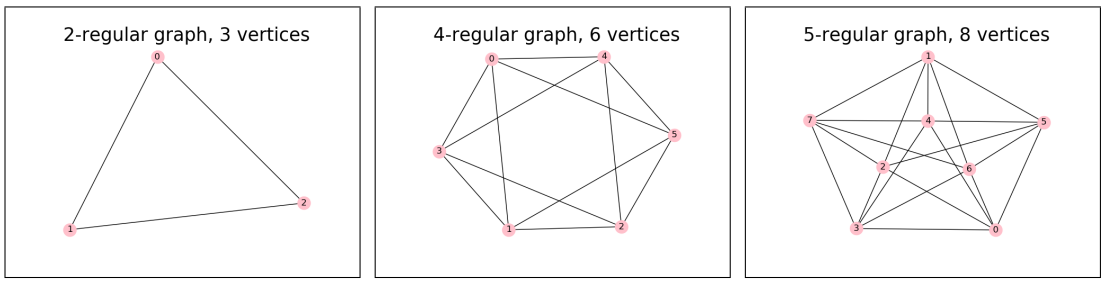

Figure 2.1: Examples of regular graphs

For simplicity, in the rest of the paper, the short notation $(n,d)-graph$ refers to a $d-regular$ graph of n vertices. Lastly, the algorithms to generate random regular graphs described in Section 3.4 only accept connected graphs, defined next.

Definition 2.2.10. *A **path** of length k from a vertex u to a vertex u' in a graph G = (V, E) is a sequence* $< v_0, v_1, v_2, ..., v_k >$ *of vertices such that u =* v_0*, u' =* v_k *and* $(v_{i-1}, v_i) \in E$ *for i = 1, 2,...,k* [RS02].

Definition 2.2.11. *An unordered pair of vertices* $\{x, y\}$ *of a graph is called **connected** if there exists a path that leads from x to y.*

Definition 2.2.12. *An undirected graph is **connected** if every pair of vertices is connected by a path.*

2.3 Signature functions

The following definitions build the background for understanding signature functions, which play an essential role in constructing graph polynomials. The sections concludes

with presenting b-matchings, the constraint applied to regular graphs to generate the polynomials analysed in this paper.

Definition 2.3.1. *The **Hamming weight** of a string is the number of symbols that are different from zero.* For the binary case, the Hamming weight represents the number of bits different from the 0 bit. The Hamming weight of the string x is written as $|x|$.

Definition 2.3.2. *A **Boolean function** of n variables is a function on $\{0,1\}^n$ into $\mathbb{R}_+$, where n is a positive integer* [CH11].

Definition 2.3.3. *A **constraint satisfaction problem (CSP)** is a class of Boolean equations which can be formulated by imposing a finite number of constraints on Boolean variables* [CH11].

Definition 2.3.4. *A Boolean constraint function $f: \{0,1\}^n \to \{0,1\}$ is **symmetric** if $f(x)$ depends only on the Hamming weight, not on the permutation of indices, for every $x \in \{0,1\}^n$.*

Definition 2.3.5. *A symmetric function $f: \{0,1\}^n \to \{0,1\}$ with d arguments (**arity d**) is associated with a **signature** $f = [f_0, f_1, f_2.., f_d]$, where $f_i = f(x)$ if $|x| = i$* [GLLZ]. The terms constraint function and signature can be used interchangeably.

> **Example 2.3.1.** For the "exact-one" function (for which exactly one value in each x is equal to 1) is $f = [0,1,0,...,0]$. The Holant problem with the "exact-one" function counts the number of perfect matchings in a graph [CLX09].

The signature functions that have been applied to the regular graphs in the experiments presented in this paper correspond to **b–matchings** of graphs, defined below.

Definition 2.3.6. *Given an undirected graph G = (V, E), a **matching** is a subset of edges M⊆E such that for all vertices v ∈ V, at most one edge of M is incident on v.*

Definition 2.3.7. *A **b-matching** M of G=(V,E) is a subset of edges such that every vertex has at most b edges of M incident with it* [Pal10].

The signature function f for a $b-matching$ is $f = [1,1,..,1,0,0,..,0]$, with $b+1$ 1s. If we have a matching M of $G = (V,E)$, consider that each vertex $v \in V$ is associated with an element $x = x_0x_1..x_{n-1} \in \{0,1\}^n$, such that x_i = 1 if $e_i \in E$ is an edge of v and $e_i \in M$. Hence, v is matched with e_i and $|x|$ represents the number of matches of v in M. A b-matching imposes $|x| \leq b$. Thus, a matching with $|x| = i \leq b$ is accepted, so $f_i = 1$. Then $f_i = 0$ for $i > b$ forbids a matching of size greater than b.

> **Example 2.3.2.** The signature function for a 3-matching is $f = [1,1,1,1,0,...,0]$.

2.4 Algebraic graph theory

Further steps are made towards defining the graph polynomials known in the field of algebraic graph theory, which uses methods of algebra to deduce theorems and properties about graphs. The definition of graph polynomial uses the notions of *graph invariant* and *polynomial ring*, which are defined first together with the auxiliary concepts of *graph isomorphism* and *ring*.

Definition 2.4.1. *Two graphs G=(V, E) and G'=(V',E') are* ***isomorphic*** *if there exists a bijection $f : V \to V'$ such that (u, v) $\in$ E if and only if (f(u), f(v)) $\in$ E'* [RS02]. The bijection is called **graph isomorphism**. If G and H and two isomorphic graphs, we write G $\simeq$ H.

Definition 2.4.2. *A function g : G$\to \mathbb{C}$, G family of graphs, is a* ***graph invariant*** *if g(G) = g(H) whenever G $\simeq$ H* [GLLZ]. It is a property of graphs that depends only on the abstract structure, not on graph representations such as particular labelings or drawings of the graph.

Definition 2.4.3. *A* ***ring*** *is a set together with two laws of composition (multiplication and addition), satisfying commutativity with respect to addition, associativity with respect to multiplication and distributivity* [Lan02].

Definition 2.4.4. *A* ***polynomial ring*** *a ring formed from the set of polynomials in one or more indeterminates (variables) with coefficients in another ring* [Lan02].

Having introduced the fundamental background, we can define graph polynomials.

Definition 2.4.5. *A* ***graph polynomial*** *is a graph invariant Q : G$\to \mathbb{C}[z]$, where $\mathbb{C}$[z] is the polynomial ring over $\mathbb{C}$.* It is a graph invariant whose values are polynomials.

> **Example 2.4.1.** The matching polynomial $m_G(x)$ is a function of the number of k-edge matchings (matchings with exactly k edges) in a graph:
>
> $$m_G(x) = \sum_{k \geq 0} m_k x^k,$$
>
> where m_k represents the number of k-edge matchings.

It is important to note that, in this paper, a graph polynomial given a signature function can refer to either a *local* polynomial or a *global* polynomial.

Definition 2.4.6. *For a symmetric signature $f = [f_0, f_1, .., f_d]$ of arity d, the* ***local polynomial*** *of f is*

$$P_f(x) = \sum_{i=0}^{d} \binom{d}{i} f_i x^i.$$

We can view $P_f(x)$ as the polynomial for a single vertex with d dangling edges (edges with no vertex at the other end) [GLLZ]. Notice how, in fact, the local polynomial does only depend on the signature function, but not on the structure of the graph. For a regular graph, the arity of the local polynomial is the same as the degree d of the graph.

Example 2.4.2. For this paper, we look at the local polynomials for $b-$matchings. Remember that the signature function for a $b-$matching is $f = [1,..,1,0,..,0]$, with $b+1$ 1s. Then:

$$P_f(x) = \binom{d}{0} f_0 x^0 + \binom{d}{1} f_1 x^1 + .. + \binom{d}{d} f_d x^d \Rightarrow$$

$$P_f(x) = \binom{d}{0} x^0 + \binom{d}{1} x^1 + .. + \binom{d}{b} f_b x^b,$$

since $f_i = 1$ for $0 \leq i \leq b$ and $f_i = 0$ for $b < i \leq d$.
The local polynomial for a b-matching is a truncated binomial expansion.

Definition 2.4.7. *The **global polynomial** of a graph G=(V, E) with signature f is:*

$$P_G(x) = \sum_{i=0}^{|E|} Z_i x^i,$$

where

$$Z_i = \sum_{\sigma \in \{0,1\}^{|E|} \text{ and } |\sigma|=i} \prod_{v \in V} f_{|\sigma_{E(v)}|},$$

and *E(v)* is the set of edges adjacent to vertex v [GLLZ].

In the above definition, $|\sigma_{E(v)}|$ is the Hamming weight of the substring of σ restricted on $E(v)$.

Notice that the $P_G(1)$, the global polynomial evaluated at 1, expresses a Holant problem as introduced in Section 1.1

Example 2.4.3. Consider the constraint *f=[1,1,0]* and G the 2-regular graph in Figure 2.2 with 3 vertices and 3 edges.

The vertices of the graph are $\{0,1,2\}$ with the following set of edges:

$$E(0) = \{e_0, e_2\},$$
$$E(1) = \{e_0, e_1\},$$
$$E(2) = \{e_1, e_2\}.$$

Moreover,

$$\sigma \in \{000, 001, 010,$$
$$100, 011, 101, 110, 111\}.$$

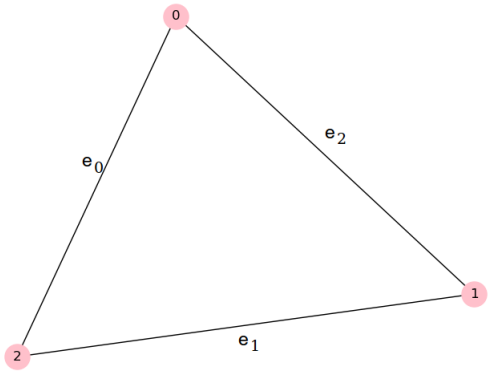

Figure 2.2: (3,2)-graph

As in the introduction, if $E(v) = (e_i, e_j)$ is the set of edges of vertex v, with $i \neq j$ and $i, j \in \{0,1,2\}$, and $\sigma = \sigma_0\sigma_1\sigma_2 \in \{0,1\}^3$, then:

$$\sigma_{E(v)} = \sigma_i\sigma_j \in \{0,1\}^2 \text{ and } |\sigma_{E(v)}| \in \{0,1,2\}.$$

Computing the Z_i coefficients, for $i \in \{0,1,2,3\}$:

$$i = 0, \ \ \sigma \in \{000\}, \ \ Z_0 = f_0 * f_0 * f_0 \Rightarrow Z_0 = 1.$$

$$i = 1, \ \ \sigma \in \{001, 010, 100\}, \ \ Z_1 = f_0 * f_1 * f_1 + f_1 * f_0 * f_1 + f_1 * f_1 * f_0 \Rightarrow Z_1 = 3.$$
$$i = 2, \ \ \sigma \in \{011, 101, 110\}, \ \ Z_2 = f_2 * f_1 * f_1 + f_1 * f_2 * f_1 + f_1 * f_1 * f_2 \Rightarrow Z_2 = 0.$$
$$i = 3, \ \ \sigma \in \{111\}, \ \ Z_3 = f_2 * f_2 * f_2 \Rightarrow Z_3 = 0.$$

Hence, the resulting global polynomial is:

$$P_G(x) = 1 + 3x.$$

Note that two isomorphic graphs with the same constraint function result in an equal global polynomials. Thus, the approaches presented in Section 3.4 to generate $(n,d) - graphs$ try to save only non-isomorphic graphs for given n and d values.

2.5 Coefficients of global polynomials

With the definition of global polynomial in mind, we make and prove some useful observations on the values of the coefficients of the polynomial.

Observation 2.5.1. If $d \leq b$, where d is the degree of the regular graph $G = (V,E)$ with b the size of the matching function, then each coefficient Z_i of P_G is:

$$Z_i = \binom{|E|}{i},$$

where $|E|$ is the number of edges of the graph.

To prove the above statement, remember that

$$Z_i = \sum_{\sigma \in \{0,1\}^{|E|} \text{ and } |\sigma|=i} \prod_{v \in V} f_{|\sigma_{E(v)}|},$$

with *E(v)* the set of edges adjacent to vertex v.

Since $0 \leq |\sigma_{E(v)}| \leq d$, then $f_{|\sigma_{E(v)}|} \in \{f_0, f_1, .., f_d\}$, for any $v \in V$ and $\sigma \in \{0,1\}^{|E|}$. We also know that $f_i = 1$ for $i \leq b$ and $d \leq b$, hence $f_{|\sigma_{E(v)}|} = 1$ for any $v \in V$ and $\sigma \in \{0,1\}^{|E|}$. Then:

$$Z_i = \sum_{\sigma \in \{0,1\}^{|E|} \text{ and } |\sigma|=i} \prod_{v \in V} 1 \Rightarrow Z_i = \sum_{\sigma \in \{0,1\}^{|E|} \text{ and } |\sigma|=i} 1 \Rightarrow$$

$$Z_i = |\{\sigma \in \{0,1\}^{|E|} \text{ and } |\sigma| = i\}|,$$

where $|A|$ denotes the cardinality of the set A.

Thus Z_i is equal to the total number of possibilities to arrange i bits of 1 in a string of $|E|$ bits, which gives us

$$Z_i = \binom{|E|}{i},$$

as claimed.

It follows that, for these cases, the global polynomial of the (n,d)-graph $G = (V,E)$ is a binomial expansion:

$$P_G = \sum_{i=0}^{|E|} \binom{|E|}{i} x^i \Rightarrow P_G = (1+x)^{|E|},$$

where $|E| = \frac{n \cdot d}{2}$.

This observation provides a great advantage when we have to compute the global polynomial for given n, d, b with $d \leq b$. The coefficients can be calculated by computing the corresponding binomial coefficients.

The previous proof does in fact help us to come up with another property of the coefficient:

Observation 2.5.2. If $i \leq b$ and $b < d$, then:

$$Z_i = \binom{|E|}{i}.$$

This relation can be quickly proved:

$$i \leq b \Rightarrow |\sigma| \leq b, \text{ for } \sigma \in \{\sigma \in \{0,1\}^{|E|} \text{ and } |\sigma| = i\} \Rightarrow$$

$$|\sigma_{E(v)}| \leq b, \text{ for } v \in V \Rightarrow f_{|\sigma_{E(v)}|} \in \{f_0, f_1, .., f_b\} \Rightarrow f_{|\sigma_{E(v)}|} = 1 \Rightarrow$$

$$Z_i = \sum_{\sigma \in \{0,1\}^{|E|} and |\sigma| = i} 1 \Rightarrow Z_i = \binom{|E|}{i}, \text{ for } i \in \{0, 1, ..b\}.$$

We can go even further and find an upper bound for the coefficients:

Observation 2.5.3. For each b, d and $0 \leq i \leq |E|$,

$$Z_i \leq \binom{|E|}{i}.$$

The proof of the last claim follows the same approach:

$$Z_i = \sum_{\sigma \in \{0,1\}^{|E|} and |\sigma| = i} \prod_{v \in V} f_{|\sigma_{E(v)}|} \Rightarrow$$

$$Z_i = \sum_{\sigma \in \{0,1\}^{|E|} and |\sigma| = i} \prod_{v \in V} (1 \text{ or } 0) \leq \sum_{\sigma \in \{0,1\}^{|E|} and |\sigma| = i} \prod_{v \in V} 1 = \binom{|E|}{i}.$$

An advantage of these observations is that they can provide us a sanity check of the correctness of the methods to generate graph polynomials described in Section 3.6. By running these algorithms for various (n, b, d), it can easily be checked if the solution is the binomial expansion if $d \leq b$ or if the first b coefficients are the expected binomial coefficients.

2.6 Roots of local and global polynomials

We provide next a lemma, stated and proved in [GLLZ], on the relation between the roots of local and global polynomials, making reference to the $H_\varepsilon - stable$ property of polynomials.

Definition 2.6.1. *A polynomial P(z) is H_ε−**stable** for some $\varepsilon > 0$ if P(z)≠0 as long as Real(z)>−ε, where Real(z) is the real part of the complex number z* [GLLZ].

We make the following observation on the stability of the local polynomial. Let $\{z_1, z_2, .., z_d\}$ be the roots of P_f and pick $z \in \{z_1, z_2, .., z_d\}$ with the maximum real part. If $Real(z) \leq 0$, then P_f is $H_{|Real(z)|} - stable$, as the real part of all the other roots is to the left of $Real(z)$, so $P(z') \neq 0$ for any complex number z' with $Real(z') > Real(z)$. However, if $Real(z) > 0$, then there is no $\varepsilon > 0$ such that P_f is $H_\varepsilon - stable$, as $P(z) = 0$ and $Real(z) > 0 > -\varepsilon$ for any $\varepsilon > 0$.

Lemma 2.6.1. *Let f be a symmetric signature of arity d. If the local polynomial $P_f(z)$ is $H_\varepsilon - stable$ for some $\varepsilon > 0$, then the global polynomial $P_G(z)$ has no zero in the Δ-strip of [0, 1], where Δ is a constant depending only on ε [GLLZ].*

The next theorem, adapted from [LPRS16] to match the requirements of our problem, is a fundamental reference and starting point for the analysis of the roots.

Theorem 2.6.2. *Suppose that there is an angle* $\theta \in [0, \frac{\pi}{2}]$ *such that each nonzero root* z_f *of the local polynomial* P_f *satisfies* $|arg(z_f)| \in [\pi - \theta, \pi]$*, where* $arg(z_f) \in [-\pi, \pi]$*. Then every nonzero root* z_G *of* P_G *satisfies* $|arg(z_G)| \in [\pi - 2\theta, \pi]$ *and* $arg(z_G) \in [-\pi, \pi]$*.*

Assume P_f is an $H_\varepsilon - stable$ local polynomial and z its root with the maximum real part. Define $arg(z)$ the angle between the positive real axis and the line joining z with the origin. From the discussion following Definition 2.6.1, $Real(z) \leq 0$ and hence $arg(z) \in [\frac{\pi}{2}, \pi]$. Let $\theta = \pi - arg(z)$ and $\theta' = \pi - 2\theta$. Then, by Theorem 2.6.2 the global polynomial has no root in the $[-\theta', \theta']$ cone. However, if the polynomial has a root z in the right half-plane, $Real(z) > 0$, then it is obvious that there does not exist an angle $\theta \in [0, \frac{\pi}{2}]$ satisfying the condition of the theorem. For such cases, the theorem cannot be applied.

To exemplify, the plot in Figure 2.3 shows the roots of the local polynomial for a 5-matching signature function of arity $d = 7$, together with the θ and θ' angles. More details on these values are discussed in Section 4.2.

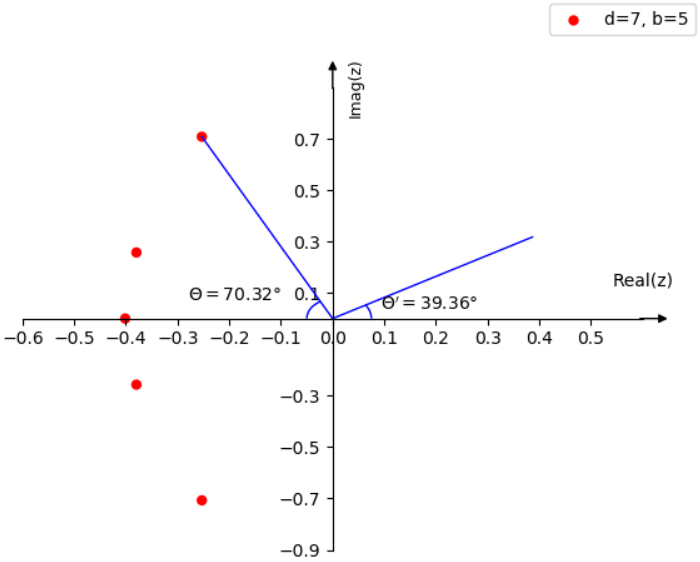

Figure 2.3: Roots of local polynomial for 5-matching of arity 7

The interest on Theorem 2.6.2 is based on its implications on the existence of a zero-free $[0, 1]$ region and, as a results, polynomial-time approximation algorithms for the associated Holant problem. It is obvious that, if the roots of the global polynomial are outside the defined $[-\theta', \theta']$ cone, then there exists a zero-free $[0, 1]$ region, as least as big as the one inside the cone. For this reason, the distribution of the generated roots has been checked in accordance with the corresponding cone defined in Theorem 2.6.2.

Chapter 3

Algorithms and Implementation Decisions

This chapter presents the tools, libraries, algorithms and implementation decisions for creating the environment to run the experiments that are described in Chapter 4. Of particular interest are the approaches taken to generate random graphs (Section 3.4) and to compute the global polynomials (Section 3.6).

3.1 Programming language and libraries

The programming language used for this project has been Python 3.7.0 ([LLCa] for official documentation). Python is a free and open-source programming language. Its portability has enabled us to easily build and run the programs both on the Dice machines provided by the School of Informatics and on a personal laptop. Due to its object-oriented features, the code could be organised in data objects, especially for a better representation of the multivariate polynomials associated with vertices described in Section 3.6.2. Lastly, Python offers great support through its extensive standard and open-source libraries. Python 3.7 is the latest stable version of the programming language. Various optimizations have reduced Python startup time by 10% on Linux and method calls are now up to 20% faster [LLCb].

One Python library used in the implementation of the algorithms to generate graphs, described in Section 3.4, is **random**. It is a module that implements pseudo-random number generators [LLCa]. In particular, *random.choice* returns a random element from the non-empty sequence.

Presented below are three other libraries which have been useful, all part of **SciPy**. SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering ([JOP+01] for official documentation).

- **SciPy library** (component of SciPy stack) is a collection of numerical algorithms and domain-specific toolboxes [JOP+01]. For this project, SciPy has been used for computing the local polynomials, as described in Section 3.5.

- **NumPy** is the fundamental package for scientific computing with Python ([Oli] for official documentation). It is open-source and has powerful tools for manipulating large multidimensional arrays and a large collection of mathematical functions to operate on these arrays. NumPy's popularity comes from the fact that most of its modules are implemented in C or Fortran, then wrapped in Python, allowing faster operations [Oli]. As well as performing different operations on arrays, NumPy has been used for representing the local and global univariate polynomials.
- **Matplotlib** is a Python plotting library which produces publication-quality 2D and 3D figures ([HJ07] for official documentation). For this project, Matplotlib has been used for plotting the graphs generated and roots of the global polynomials, providing a better understanding of the data gathered.

3.2 Root solver

One of the first goals towards answering the experimental questions of this paper was to find a reliable algorithm or library for computing the complex roots of univariate polynomials. This is a well studied topic in Computer Algebra, with known methods and numerical approximations for root finding, such as Newton's Method and Inverse Interpolation. More information on the numerical methods for the root finding problem can be found at [Dat13].

Two Python libraries that provide in-built algorithms for polynomial root finding have been considered:

- **SymPy** is a Python library under SciPy, for symbolic mathematics and computer algebra ([MSP+17] for official documentation). SymPy in its turn uses the library **mpmath**, for real and complex floating-point arithmetic with arbitrary precision ([J+13] for official documentation). SymPy provides the *polyroots* method for computing the roots of a polynomial using the Durand-Kerner method (described in [Kho18]), which performs simultaneous Newton iterations for all the roots, and the *solve* method for algebraic equations, including polynomials. The official documentation does not explicitly say what algorithm *solve* implements, but it suggests that the code uses a series of transformations and existing tools for solving polynomials.
- **NumPy**'s *poly1d* class provides a representation of polynomials together with a *roots* method. The roots are computed using the eigenvalues of the companion matrix (described in [EM95]).

Simple experiments have been carried out to compare the efficiency of the solvers implemented by the libraries, by providing the methods with a list of coefficients. SymPy's *polyroots* involves adding a parameter for the maximum number of iterations until the solution converges. However, the high number of different polynomials of the experiments made it difficult to find the optimal value for this parameter, affecting the reliability of this solution.

The efficiency of Sympy's *solve* and NumPy's *roots* has been tested by comparing the time to compute the roots for polynomials with degrees up to 10. The plot in Figure 3.1 shows that NumPy is faster, with SymPy sometimes being even 10 times slower. These results, together with the polynomials the libraries have been tested on, can be found in Appendix A.

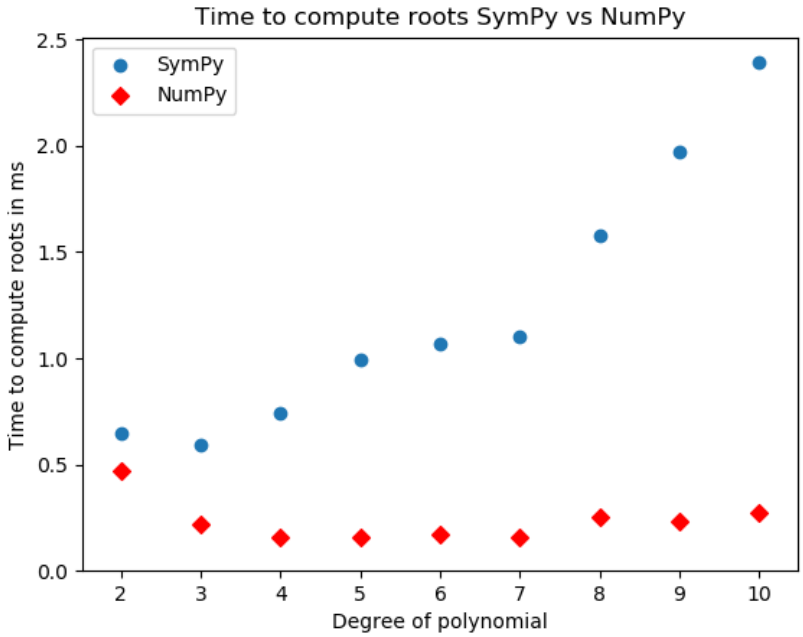

Figure 3.1: SymPy vs NumPy for root solving

Moreover, while SymPy returns results of type *sympy.core.numbers* that needs further processing for getting the numerical values, NumPy's return type is *numpy.complex128*. Evaluating the roots from SymPy to numerical results produces a substantially increased overhead in the running time as well. It was decided that NumPy's type is easier to work with and to separate the real and imaginary parts of a complex number.

Lastly, concern has been expressed regarding the accuracy of the results. For the purpose of comparison, for the following polynomial:

$$p(x) = x^5 - 4x^4 - 7x^3 + 14x^2 - 44x + 120,$$

whose roots are $x \in \{2, 5, 3, -2i, +2i\}$, where $i = \sqrt{-1}$, NumPy's output is:

$$x \in \{2, 5, 3, -4.71844785e^{-16} + 2i, -4.71844785e^{-16} - 2.i\},$$

The returned output has a very small error of e^{-16} corresponding to 10^{-16} for the real part of the complex solutions. In contrast, SymPy returns the the expected roots.

Another limitation of NumPy has been observed for the polynomial corresponding to a binomial expansion. If

$$p(x) = (x+1)^d,$$

NumPy returns d roots distributed close to a circle centered at -1, while SymPy returns the -1 root with multiplicity d. The distance from -1 to NumPy's roots increases as the degree of the expansion becomes larger. For degree 4, for example, the roots are around 0.0002 away from -1, while for degree 25 the largest Euclidean distance between a root and -1 is almost 0.6.

Despite this inaccuracy for the special case of binomial expansion, there have not been observed major discrepancies in the results for other graph polynomials, despite a precision of 8 decimals for NumPy compared to 15 for SymPy. However, these comparisons have been possible for polynomials with degree lower than 10, due to the slow numerical evaluation of SymPy's roots. This numerical transformation constitutes a major cause of the impracticability of SymPy for our experiments.

Considering all the reasons presented above, we have decided to use NumPy's library for root solving.

3.3 Graph representation

After having analysed the possible solutions for graph representation, including adjacency lists, adjacency matrices and Graph objects we created, we have decided to use **NetworkX**. NetworkX is a Python package *"for the creation, manipulation and study of the structure, dynamics and functions of complex networks"* [HSSC08]. It provides data structures for graphs and many standard graph algorithms.

The functionalities of this library have been of substantial help. For example, we have used it to iterate through the edges or vertices of a graph, to save to a file the adjacency list of a graph or to create a graph from its adjaceny list in the text file. The algorithms described in the next section aim to return connected graphs. NetworkX's method *is_connected* has been used to check if a graph is connected. Moreover, we have experimented with more than one graph for given n and d when more then one non-isomorphic such graphs exist. We have used NetworkX's *is_isomorphic* method, which implements the VF2 algorithm [CFSV01]. Lastly, NetworkX has a *draw* method that, together with matplotlib, has facilitated the visualisation of the graphs generated. All the graph figures in this paper have been generated using the *networkx.draw* method.

3.4 Random regular graph generation

Various studies have been made on random regular graphs and the importance of their properties in Combinatorics and Theoretical Computer Science (with some properties presented in [Wor99]), with particular interest being given to the uniform generation of these graphs. While there exist different algorithms (with examples at [NLKB11]) for generating random graphs of n vertices, not necessarily regular, there are practical issues when it comes to their implementation, due to their time complexity or the

low probabilities of generating graphs of high-degrees, resulting in many iterations or restarts. Biases or various heuristics in picking edges or pairing vertices may also result in the returned graph not being chosen from a uniformly random distribution [KHH12].

For the rest of this paper, the notation $G_{n,d}$ refers to the group of all the d-regular graphs of n vertices. Note that the group does only consider connected non-isomorphic graphs, as a graph with $c \geq 2$ connected components is equivalent to c smaller graphs. The complexity of generating uniform regular graphs is a result of the big number of d-regular graphs as n increases. We shall call the number of graphs in the $G_{n,d}$ group as $|G_{n,d}|$. There is no formula for the exact value of $|G_{n,d}|$, but there are asymptotic results by Béla Bollobás [Bol01]. Work towards establishing these values and generating the connected non-isomorphic graphs has been done by Markus Meringer [Mer99], providing the exact number of simple connected non-isomorphic d-regular graphs for $n \leq 16$ and $d \leq 7$. The values for $n \in \{12, 14, 16\}$ and $d = 7$ are provided as examples:

$$|G_{12,7}| = 1547, |G_{14,7}| = 21609301, |G_{16,7}| = 733351105934.$$

Note how increasing n from 14 to 16 results in the graphs group growing almost 33400 times in size. A table of more values as established by Meringer is provided in Appendix B.

Working towards generating simple connected non-isomorphic d-regular graphs for the aims of this project, different approaches have been considered and compared, as summarised in the next sections.

3.4.1 The pairing model

In his paper, Bollobás [Bol01] describes the *pairing model* (also known as the *configuration model*) to generate (n,d)-graphs. His idea is based on having n sets (also referred to as *buckets*) of d elements each, that can be seen as d copies of each vertex. The algorithm, described in Algorithm 1, keeps pairing two elements from different sets. At the end, an edge is added to the graph G between vertex u and vertex v if there is a pair with one end in u's bucket and the other end in v's bucket. If the resulting graph is not simple, the algorithm starts again and is repeated until the graph is simple.

Algorithm 1: The pairing model

Input: n, d;
Output: a uniform random (n,d)-graph G;
create n sets with d elements each;
while *there are elements not paired* **do**
    randomly choose i and j to pair from the $n \cdot d$ elements;
end
for *every pair* (i, j) **do**
    add edge (u,v) to graph G;
    where
        u = bucket number of element i;
        v = bucket number of element j;
end
if *G is not simple* **then**
    restart;
else
    return G;
end

If d is a constant, then the probability of the resulting graph being simple is bounded by a positive constant, uniformly in n. This, however, is no longer true if d is large, for instance $logn$ or n^a, so that most of the time the graph is not simple [HKV06]. It has been shown that the probability to obtain a simple graph is:

$$P(G \text{ is simple}) = e^{\frac{-d}{4}},$$

which quickly approaches 0 as d increases [Wor99], requiring many repetitions until a solution is found. To deal with the issue of many repetitions, our approach in implementing this algorithm involved an early stop heuristic. In particular, if adding an edge for the newly picked pair (i, j) would result in the graph not being simple, restart the algorithm without waiting to match all elements. The improvement saves the time spent by pairing the rest of the elements if the graph so far is already not simple, but the number of times the algorithm restarts is not affected. An analysis of this number of restarts (trials) is provided in Section 3.4.5.

3.4.2 Non-uniform graph generation

Inspired by Bollobás's pairing model, we have initially devised an algorithm that generates regular graphs, with the main difference being the restriction of the pairs to ensure the graph is simple. The motivation behind having a second solution was to reduce the number of trials to return a simple graph. The implementation follows the approach described below in Algorithm 2.

Algorithm 2: The non-uniform pairing model

```
Input: n, d;
Output: a uniform random (n,d)-graph G;
create n sets with d elements each;
for each set s_u from set_0 to set_{n-1} do
    for each element i in s do
        Σ = the set of sets s_v such that i can be paired with at least one element in s_v
          without creating a loop or multi-edge;
        number_of_attempts = 0;
        while a suitable match j for i is not found do
            if number_of_attempts exceeds threshold then
                restart;
            else
                randomly pick set s_v from Σ;
                randomly pick j from s_v;
                number_of_attempts++;
            end
        end
        pair i and j;
    end
end
for every pair (i, j) do
    add edge (u, v) to graph G;
    where
        u = bucket number of element i;
        v = bucket number of element j;
end
return G;
```

Note that in the algorithm above, (i, j) is not a suitable match if j has already been matched. In order to avoid infinite loops in the case of no suitable match being found, a threshold has been added for the number of attempts to run the while loop. If the threshold has been exceeded, the algorithm restarts: $n \cdot d$ attempts are allowed before the algorithm restarts. While this method is faster than the two approaches presented previously, it involves a high degree of bias in the pairing strategy: i is not randomly chosen at all and the possibilities of choosing j are restricted to permitted values that do not cause loops or multiple edges.

3.4.3 Steger and Wormald's refined pairing model

Steger and Wormald propose a refinement of Bollobás's algorithm [SW99], which guarantees that the returned graph is simple. As in the original algorithm, they consider a set of $n \cdot d$ points (or n buckets of d elements each). When picking the points to pair,

the algorithm does not accept pairs that create a loop or would result in a graph with multiple edges. An edge is called suitable if would not create a loop or a parallel (multiple) edge in G. Hence, two points are suitable if they are in different buckets and no currently existing pair contains points in the same two buckets. The algorithm is described below in Algorithm 3.

Algorithm 3: The refined pairing model

Input: n, d;
Output: a uniform random (n,d)-graph G;
create n sets with d elements each;
while *there is no suitable pair* **do**
    randomly choose i and j from the $n \cdot d$ elements;
    if *i and j form a suitable pair* **then**
        pair i and j;
        delete i and j from the sets;
    end
end
if *all elements have been paired* **then**
    for *every pair* (i,j) **do**
        add edge (u,v) to graph G;
        where
            u = bucket number of element i;
            v = bucket number of element j;
    end
    return G;
else
    restart;
end

Their algorithm generates random d-regular graphs which are asymptotically uniform. In contrast with the non-uniform solution that restricts the pairing, Steger and Wormald's refinement checks for suitability after the random choice of j has been made. Notice that, compared to Bollobás's approach which keeps restarting until it generates a simple graph, in this second approach the resulting graph is guaranteed to be simple after one iteration of the algorithm, provided all the $n \cdot d$ elements are paired. This last remark suggests that the algorithm could reach a dead end, if not all $n \cdot d$ elements have been paired but *no suitable pair* can be found. In such cases, the algorithm has to be restarted. Steger and Wormald do not suggest how to check whether there are suitable pairs left. One possible approach, although inefficient, would be to iterate through all the possible pairs and check there is at least one that satisfies the suitability condition.

3.4.4 The faster refined pairing model

Steger and Wormald suggest as well a smart way to choose the pairs (i, j). They also prove that, for fixed d and n with $d = O(\sqrt{n})$, the algorithm generates the graph in running time $O(nd^2)$ [SW99]. However, for d larger than than $\sqrt{n}$, Steger and Wormald state that the algorithm is expected to give $O(d^4)$ complexity. The pairing strategy of this refined model is divided into three phases, summarised below. The algorithms in these phases follow their high-level description of the pairing model.

Algorithm 4: The faster refined pairing model, Phase 1

```
initialize array U with n·d points;
while size of U ≥ 2d² do
    randomly choose i and j from U;
    if i and j form a suitable pair then
        pair i and j;
        delete i and j from U;
    end
end
```

Algorithm 5: The faster refined pairing model, Phase 2

```
while size of U ≥ 2d do
    randomly choose vertices u and v that do not have all points matched yet;
    (u and v do not have degree d yet);
    if (u,v) is not an edge of G yet then
        randomly choose i from the points in u's bucket;
        randomly choose j from the points in v's bucket;
        if i∈U and j∈U then
            pair i and j;
            delete i and j from U;
        end
    end
end
```

Algorithm 6: The faster refined pairing model, Phase 3

construct G'=graph induced by all vertices of G of degree less than d;
H=G's complement;
while *size of* $U \geq 0$ **do**
    randomly choose (u,v) from the edges of H;
    x_u=number suitable points in u;
    x_v=number suitable points in v;
    $x_{uv} = x_u x_v$;
    if *(u,v) is accepted with probability* $\frac{x_{uv}}{d^2}$ **then**
        randomly choose i from the points in u's bucket;
        randomly choose j from the points in v's bucket;
        if $i \in U$ *and* $j \in U$ **then**
            pair i and j;
            delete i and j from U;
            delete (u,v) from the edges of H;
        end
    end
end

Phase 3 presented above involves the construction of the induced graph G' and its complement H, whose definitions are provided next for convenience:

Definition 3.4.1. *A graph G'= (V',E') is a* ***subgraph*** *of G=(V, E) if* $V' \subseteq V$ *and* $E' \subseteq E$. *Given a set* $V' \subseteq V$, *the subgraph of G* ***induced*** *by V' is the graph G'=(V',E'), where* $E' = \{(u,v) \in E : u,v \in V'\}$ [RS02].

Definition 3.4.2. *The* ***complement*** *or* ***inverse*** *of a graph G is a graph H with the same vertices such that two distinct vertices of H are adjacent if and only if they are not adjacent in G* [RS02].

The resulting graph is formed from all the selected pairs, as in the previous approaches. A slight variation of the algorithm presented below has been implemented. The difference comes in adding again a threshold $= n \cdot d$ for the number of times each phase tries to pick a pair. The algorithm restarts when failure to find a suitable pair happens consecutively for the threshold number of attempts.

We have made use of NumPy's arrays for keeping the values of U, Python's dictionaries for mapping a vertex with its set (bucket) of elements and NetworkX's Graph object and *complement* method to generate H. Lastly, all the random choices have been made using Python's *random* library.

3.4.5 Comparisons

The decision on which algorithm to use for generating all the regular graphs has been made after the algorithms described in Sections 3.4.1, 3.4.2 and 3.4.4, or their slight variations such as the above-mention threshold, have been implemented and compared in terms of time efficiency. Note that in the actual implementation, all approaches discard graphs that are not connected and restart the generation. We were in particular interested in how fast they could find a solution and in the number of necessary trials (restarts) and tests have been carried out. Some results for $n \leq 12$ and $d \leq 7$ can be seen in Table 3.1.

The first pairing model requires substantially many trials. For this approach, the algorithm stopped the search and returned no solution if it was not successful after 1000 trials. For large d, closer to n, it was indeed the case that no solution could be found in 1000 trials. The time is in milliseconds and the values have been obtained by running the algorithm 5 times and averaging the results. The average number of trails has been rounded to the nearest integer.

n	d	Initial pairing model		Non-uniform		Faster refinements	
		avg. time	avg. trials	avg.time	avg. trials	avg. time	avg. trials
10	5	178.92	861	2.76	1	35.80	3
10	6	215.78	$\geq$1000, stopped	4.60	1	64.36	3
11	4	71.54	323	3.29	2	4.73	1
11	8	265.08	$\geq$1000, stopped	10.09	2	290.12	6
12	5	278.87	819	7.02	1	27.02	2
12	6	279.83	$\geq$1000, stopped	6.94	1	42.52	2
12	7	320.77	$\geq$1000, stopped	10.41	2	189.31	4

Table 3.1: Time in ms and trials to generate graphs

Compared to the initial pairing model, the Steger and Wormald's faster refined pairing model and the non-uniform implementation manage to find the solutions in an acceptable time. Steger and Wormald's method works reasonably well for $n \leq 12$ and $d \leq 7$. With bigger n and large d, however, this method is not so reliable, due to slow running and many necessary trials. A combination of the two methods could be use for these cases, choosing the approach to take by analysing d with respect to n. The non-uniform implementation could be used when a result is unachievable by Steger and Wormald's methods. The non-uniform approach manages to find graphs as big as 100-regular for 200 vertices in 24.53 seconds and 18 trials.

For the purposes of the experiments in this paper, that do no deal with graphs of large size, we have hence decided to use Steger and Wormald's faster refined pairing method, which ensures asymptotically uniform random generation, relatively fast running time and an acceptable number of trials.

3.5 Local polynomial

Remember that the local polynomial for a symmetric signature $f = [f_0, f_1, .., f_d]$ of arity d is:

$$P_f(x) = \sum_{i=0}^{d} \binom{d}{i} f_i x^i,$$

where d is the degree of the regular graph. We have implemented an algorithm that computes each coefficient $c_i = \binom{d}{i} f_i$ and returns the *numpy.poly1d* from the list of these coefficients. For computing the combination *d takes n*, the **scipy.special** library with its *comb* method has been used.

3.6 Global polynomial

Extensive effort was devoted to implementing an efficient algorithm to compute the global polynomial. We describe next and compare four approaches taken, with the last one being a valuable contribution of this project. For convenience, we present again the definition of the global polynomial, as stated in Section 2.4.

Definition 3.6.1. *The **global polynomial** of a graph G=(V, E) with signature f is:*

$$P_G(x) = \sum_{i=0}^{|E|} Z_i x^i,$$

where

$$Z_i = \sum_{\sigma \in \{0,1\}^{|E|} \text{ and } |\sigma|=i} \prod_{v \in V} f_{|\sigma_{E(v)}|},$$

and *E(v)* is the set of edges adjacent to vertex *v* [GLLZ].

3.6.1 Naive implementation

3.6.1.1 Algorithm overview

The most obvious and straightforward implementation involves iterating through all $\sigma \in \{0,1\}^{|E|}$ to compute each Z_i coefficient, following exactly the formula of Z_i in the above definition. The values in the set $\{0,1\}^{|E|}$ can be computed by iterating through the binary representation of the numbers from 0 to $2^{|E|} - 1$. A dictionary (named *h_dict*) is created initially, with keys from 0 to $|E| - 1$, such that $h\_dict(i)$ stores a list of all $\sigma \in \{0,1\}^{|E|}$ with $|\sigma| = i$, where $|\sigma|$ is the Hamming weight of σ. The algorithm is presented below in Algorithm 7, with an example at the end of the subsection.

We make the important observation first that each edge of the graph is mapped to an integer value from 0 to $|E| - 1$. Hence, σ_e refers to the bit of σ at index equal to the

integer of the edge e. So $|\sigma_{E(v)}|$ is the sum of all these bits for $E(v)$ the set of edges adjacent to a vertex v.

Algorithm 7: Naive implementation for global polynomial

Input: G, an $(n,d)-$graph, f signature function;
Output: a list of coefficients of G's global polynomial;
create h_dict;
for *i from 0 to* $|E|-1$ **do**
  Z_i=0;
  for $\sigma \in h\_dict(i)$ **do**
    $\text{product}_\sigma = 1$;
    for $v \in G.nodes$ **do**
      $|\sigma_{E(v)}| = 0$;
      for $e \in v.edges$ **do**
        $|\sigma_{E(v)}| = |\sigma_{E(v)}| + \sigma_e$;
      end
      $\text{product}_\sigma = \text{product}_\sigma * f_{|\sigma_{E(v)}|}$;
    end
    $Z_i = Z_i + \text{product}_\sigma$;
  end
  return list of Z coefficients;
end

The body of the first *for* loop computes the value of each coefficient Z_i, a sum of products. In the second *for* loop, for each Z_i, we iterate through $\sigma \in \{0,1\}^{|E|}$, with $|\sigma| = i$, and compute $product_\sigma$, a product of the values of f. For each node v in the graph, we multiply $product_\sigma$ with the value of f at index $|\sigma_{E(v)}|$.

3.6.1.2 Complexity analysis

To compute all $|E|$ coefficients, each σ is used exactly once. Its Hamming weight is unique, so each σ appears exactly once in h_dict, in its $|\sigma|$ list. Thus, the first 2 loops result in $2^{|E|}$ iterations (one for each σ). For each of the n nodes, we iterate through its d edges, resulting in $n \cdot d$ constant operations. Hence, the complexity of this approach is $O(nd2^{|E|})$. Remember that for an $(n,d)-$graph, $|E| = \frac{n \cdot d}{2}$, which gives us $O(nd2^{\frac{nd}{2}})$ time complexity to run the algorithm.

3.6.1.3 Example

We provide next a short example of computing the global polynomial with the naive implementation for the (3,2)-graph in Figure 3.2.
Let the signature function be:

$$f = [f_0, f_1, f_2].$$

The vertices of the graph are $\{0,1,2\}$ with the following set of edges:

$$E(0) = \{0,2\}, E(1) = \{0,1\}, E(2) = \{1,2\},$$

mapping $e_0 \to 0, e_1 \to 1, e_2 \to 2$.

The values of *h_dict* are:

$$h\_dict(0) = \{000\}$$
$$h\_dict(1) = \{001,010,100\}$$
$$h\_dict(2) = \{011,101,110,\}$$
$$h\_dict(3) = \{111\}$$

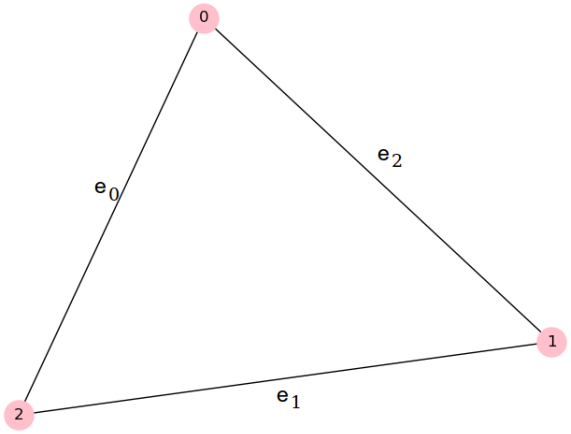

Figure 3.2: (3,2)-graph

- $Z_0 = product_{000} = f_{0+0}f_{0+0}f_{0+0} = f_0f_0f_0 = f_0^3$
- $Z_1 = product_{001} + product_{010} + product_{100} = f_{0+1}f_{0+0}f_{0+1} + f_{0+0}f_{0+1}f_{1+0} + f_{1+0}f_{1+0}f_{0+0} = 3f_0f_1^2$
- $Z_2 = product_{011} + product_{101} + product_{110} = f_{0+1}f_{0+1}f_{1+1} + f_{1+1}f_{1+0}f_{0+1} + f_{1+0}f_{1+1}f_{0+1} = 3f_1^2f_2$
- $Z_3 = product_{111} = f_{1+1}f_{1+1}f_{1+1} = f_2f_2f_2 = f_2^3$

The final answer is:

$$P_G = f_0^3 + 3f_0f_1^2x + 3f_1^2f_2x^2 + f_2^3x^3,$$

with the coefficients:

$$[f_0^3,\ 3f_0f_1^2,\ 3f_1^2f_2,\ f_2^3].$$

If $f = [1,1,0]$, then:

$$P_G = 1 + 3x.$$

3.6.2 Contraction of multivariate polynomials

The complexity of the naive implementation is highly influenced by the need to generate $2^{|E|}$ numbers. Aiming for a more efficient implementation, we have tried a second approach, inspired by proofs in [LPRS16] and [GLLZ] in which each vertex is attached a polynomial and all polynomials are multiplied for a multivariate global result.

3.6.2.1 Algorithm overview

The contraction of multivariate polynomials method involves attaching to each vertex v a multivariate polynomial P_v, whose variables come from the edges of v:

$$P_v = \sum_{i=0}^{d} \sum_{S \subseteq E(v) \text{ and } |S|=i} (\prod_{e \in S} x_e) f_i,$$

where d is the degree of the regular graph, f is the signature function applied to the graph, $E(v)$ is the set of edges associated with v and x_e is a variable corresponding to edge e. In fact, if f is a b-matching with $b \leq d$, then $f_i = 0$ for any $i > b$. Hence, we can restrict the polynomial to:

$$P_v = \sum_{i=0}^{b} \sum_{S \subseteq E(v) \text{ and } |S|=i} (\prod_{e \in S} x_e) f_i.$$

The n vertex polynomials are then multiplied to obtained one resulting multivariate polynomial P:

$$P = \prod_{i=0}^{n} P_i = (((P_0 P_1) P_2) ... P_n),$$

where the vertices are associated with numbers from 0 to $n-1$.

Multiplying two vertex polynomials P_v and P_u is similar to deleting the vertices v and u and adding a new vertex w whose edges are $E(v) \cup E(u)$.

The multiplication of any two multivariate polynomial P_v and P_u is followed by the reduction and contraction stages. Consider $E(v,u)$ the set of variables common to both P_v and P_u. Then $E(v,u)$ is equivalent to the set of edges between v and u (where v or u could have been obtained in any previous multiplication). In the original graph, each edge belongs to exactly two vertices. Hence, each corresponding variable appears in exactly two vertex polynomials. At any computation stage in the multiplication of the n polynomials, any common variable of P_v and P_u cannot appear in P_w, $w \neq v$, $w \neq u$. In the reduction stage, we discard from $P_v P_u$ all the terms with one or more variables in $E(v,u)$ with odd degree. In the contraction stage, we replace in $P_v P_u$ any variable $x_e \in E(v,u)$ with even degree by a universal variable x, with degree $deg(x) = \frac{deg(x_e)}{2}$.

After applying reduction and contraction to $P = \prod_{i=0}^{n} P_i = (((P_0 P_1) P_2) ... P_n)$, it is guaranteed that the resulting polynomial P_G is univariate in x and it is the global polynomial we are looking for.

A more in-depth implementation is presented next and an example is provided at the end of this subsection. Algorithm 8 sketches the main steps of the implementation: creation of the n vertex polynomials and multiplication polynomial by polynomial for

the final result. The strategy to create each polynomial is presented in Algorithm 9 and the multiplication procedure in Algorithm 10.

Algorithm 8: Global polynomial with multivariate polynomials multiplication

Input: G, an $(n,d)-$graph, f signature function;
Output: a list of coefficients of G's global polynomial;
for *each v in G.nodes* **do**
  create P_v;
end
$P_G = P_0$;
for *i from vertex 1 to* $n-1$ **do**
  $P_G = P_G P_i$;
end
return $P_G.coefficients$;

Each P_v polynomial is created as a Polynomial object. The Polynomial object stores a set of variables, which are initially the edges of the vertex, and a list of terms. Each term is represented with a Term object, that stores the value of the f_i coefficient and a variables dictionary, where each variable is mapped to its degree. A Polynomial object can be instantiated with a given list of Term objects and the set of variables, useful when it is the result of a multiplication, or it can compute these values as in Algorithm 9 when no input is given. The use of sets and dictionary guarantees (amortised) constant operations, such as updates, addition or removal of keys.

Algorithm 9: Creating vertex polynomials

Input: G, an $(n,d)-$graph, v vertex of G, f signature function;
Output: a multivariate polynomial P_v;
d = degree of v;
$variables = E(V)$, edges of v;
initialize *terms* (empty list of Terms);
for *i from 0 to d* **do**
  for $S \subseteq E(v)$ *with* $|S| = i$ **do**
    add to *terms* new Term with S list of variables of degree 1 and f_i coefficient;
  end
end

The multiplication in Algorithm 10 integrates the reduction and contraction steps as well. Obtaining the list of common variables between two polynomials, essential in reduction and contraction, can easily be done by intersecting the sets of variables of each polynomial. It is important, however, to ensure that the general variable x (for the final univariate polynomial) is not eliminated in reduction, hence x must not the included in the intersected set.

Algorithm 10: Multivariate polynomials multiplication

Input: P_v, P_u, multivariate polynomials;
Output: P_{vu} result of P_vP_u;
initialise new list *terms*;
$E(v,u) = P_v.variables \cap P_u.variables - \{x\}$;
for $t_v \in P_v.terms$ **do**
  for $t_u \in P_u.terms$ **do**
    $t_{vu} = t_v t_u$;
    if *!(*$\exists x_e \in E(u,v)$ *and* $x_e \in t_{vu}$ *with degree of* x_e *odd* **then**
      if $\exists x_e \in E(u,v)$ *and* $x_e \in t_{vu}$ *with degree of* x_e *even* **then**
        replace x_e^{2k} with x^k in t_{vu};
      end
      add t_{vu} to *terms*;
    end
  end
end
return new Polynomial with *terms* list of Terms and $E(v,u)$ variables set;

Multiplication is done term by term, with two *for* loops: each term of the first polynomial "multiplied" with each term of the second polynomial. A Term has a coefficient value and a dictionary for variables and their degrees. Term by term multiplication involves normal integer multiplication of coefficients (which are values of the signature function) and concatenation of the two dictionaries, such that the keys of the resulting dictionary t_{uv} are $t_v.keys \cup t_u.keys$ and $t_{uv}(x_e) = t_v(x_e) + t_u(x_e)$ (where $t_v(x_e)$ or $t_u(x_e)$ can be 0 if x_e is not common variable).

If $t_v t_u$ contains a common variable with odd degree, $t_v t_u$ is discarded (reduction step). If $t_v t_u$ contains a common variable with even degree, the variable is contracted.

3.6.2.2 Complexity analysis

When creating the individual vertex polynomials, each subset of $E(v)$ is used only once to construct a new Term. Since $|E(v)| = d$, then there are 2^d subsets of $E(v)$. Thus, each P_v has at most 2^d terms. The number of term of a vertex polynomial does in fact depend on the signature function. When the signature function f is a b-matching, the number of terms P_v has is:

$$|P_v.terms| = \sum_{i=0}^{b} \binom{d}{i} \leq 2^d.$$

Each P_v takes $b\sum_{i=0}^{b}\binom{d}{i}$ operations to construct, where b comes from the time to create each term with at most b variables. Therefore, generating n multivariate polynomials results in $O(nb\sum_{i=0}^{b}\binom{d}{i})$ time complexity, which is equal to $O(nd2^d)$ for $b=d$.

In the first multiplication, P_0 and P_1 have each at most 2^d terms, and a term has a maximum of d variables (when $b=d$). Computing the product t_{vu} of two terms does at most $2d$ operations (iterating through d variables in t_v and then through d in t_u). P_0P_1 can have at most $(2^d)^2$ terms, and a term has a maximum of $2d$ variables. Hence, computing $P_0P_1...P_n$ takes at most $O(nd2^{nd})$ time, with the worst case happening when $b=d$. However, in our experiments, the approach presented in this sections is taken when b is smaller than d, with the special case $b=d$ being dealt separately, as presented in Section 2.5. Hence, the complexity becomes at most $O(nd(\sum_{i=0}^{b}\binom{d}{i})^n)$. Slight improvements are also made due to the reduction step that makes the number of terms smaller than $(\sum_{i=0}^{b}\binom{d}{i})^k$ after the k-th multiplication.

3.6.2.3 Example

We provide next a short example of the multivariate polynomial multiplication for the (3,2)-graph in Figure 3.3.

Let the signature function be:

$$f=[f_0,f_1,f_2].$$

The vertices of the graph are $\{0,1,2\}$ with the following set of edges:

$$E(0)=\{e_0,e_2\},\ E(1)=\{e_0,e_1\},\ E(2)=\{e_1,e_2\}.$$

Then the vertex polynomials are:

$$P_0=f_0+x_{e_0}f_1+x_{e_2}f_1+x_{e_0}x_{e_2}f_2,$$

$$P_1=f_0+x_{e_0}f_1+x_{e_1}f_1+x_{e_0}x_{e_1}f_2,$$

$$P_2=f_1+x_{e_1}f_1+x_{e_2}f_1+x_{e_1}x_{e_2}f_2.$$

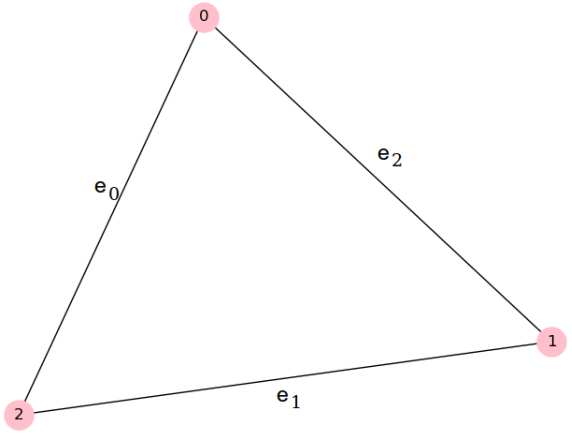

Figure 3.3: (3,2)-graph

The common variable between P_0 and P_1 is e_0. The result of multiplying P_0 and P_1 is:

$$P_0P_1=f_0^2+x_{e_1}f_0f_1+x_{e_2}f_0f_1+x_{e_1}x_{e_2}f_1^2+xf_1^2+xx_{e_1}f_1f_2+xx_{e_2}f_1f_2+xx_{e_1}x_{e_2}f_2^2.$$

The final answer is $P_G = (P_0P_1)P_2$:

$$P_G = f_0^3 + 3f_0f_1^2x + 3f_1^2f_2x^2 + 3f_2^3x^3,$$

with the coefficients:

$$[f_0^3,\ 3f_0f_1^2,\ 3f_1^2f_2,\ f_2^3].$$

If $f = [1,1,0]$, then:

$$P_G = 1 + 3x.$$

3.6.3 Kronecker substitution

The next question was whether it would be possible to find a more efficient multiplication of multivariate polynomials, using univariate polynomials. Although point-wise multiplication of univariate polynomials of degree δ takes $O(\delta^2)$ time, the Fast Fourier Transform (also called *FFT*) can be used for $O(\delta lg(\delta))$ univariate polynomial multiplication (where *lg* is logarithm base-2) [RS02].

3.6.3.1 Algorithm overview

The existence of efficient algorithm such as Fast Fourier Transform have motivated us to try to transform the problem of multivariate polynomials multiplication to a problem of univariate polynomials multiplication, by using Kronecker substitution to create a create univariate polynomials from each multivariate ones, a method explained in [Pan94]. The steps to reach the final polynomial are summarised below:

1. Create a multivariate polynomial P_v for each vertex v as in Algorithm 8.
2. Similar to Algorithm 9, multiply polynomials sequentially, this time using the Kronecker substitution for multiplication of two polynomials: $((P_0P_1)P_2)...P_n$
3. Apply contraction to the final $P = \prod_{i=0}^{n} P_i$ to get P_G.

We expand in Algorithm 11 the Kronecker substitution method, tailored to the structure of our data. An example is presented at the end of this subsection. The same Polynomial class to represent the univariate polynomials as in the solution of Section 3.6.2 has been used.

Algorithm 11: Multivariate polynomials multiplication using Kronecker substitution

Input: P_v, P_u, multivariate polynomials;
Output: P_{vu} result of $P_v P_u$;
$E(v,u) = P_v.variables \cup P_u.variables$;
for *each* $x_{e_i} \in E(v,u)$ **do**
    $\text{degree}_v(x_{e_i})$=max degree of variable x_{e_i} in P_v;
    $\text{degree}_u(x_{e_i})$=max degree of variable x_{e_i} in P_u;
    $\pi(x_{e_i}) = \text{degree}_v(x_{e_i}) + \text{degree}_u(x_{e_i}) + 1$;
end
for *each* $x_{e_i} \in E(v,u)$ **do**
    substitute $x_{e_{i-1}}^{\pi(x_{e_{i-1}})}$ for x_{e_i} in P_v;
    substitute $x_{e_{i-1}}^{\pi(x_{e_{i-1}})}$ for x_{e_i} in P_u;
end
P' = multiplication of P_u and P_v;
P_{vu} = inverse substitution of P';
apply reduction to P_{vu};
return P_{vu};

The multiplication method consists first of finding a bound on the degree of the variables occurring in the result of the multiplication. The first *for* loop computes this bound π for each variable appearing in P_v, P_u or both. The variable are then substituted in P_v and P_u, such that, if $\{x_{e_0}, x_{e_1}, .., x_{e_k}\}$ is the set of all variables, each $x_{e_i} \in \{x_{e_1}, .., x_{e_k}\}$, from higher index to lower, is replaced with $x_{e_{i-1}}^{\pi(x_{e_{i-1}})}$.

The π dictionary matches each variable x_{e_i} with its degree bound. We define π_0 such that:

$$\pi_0(x_{e_0}) = 1$$
$$\pi_0(x_{e_i}) = \pi(x_{e_{i-1}})\pi_0(x_{e_{i-1}}).$$

Thus, π_0 computes substitution degrees relative to e_0.

Let one of the terms of P_v, for example, be:

$$t = x_{e_i}^{deg(x_{e_i})} x_{e_{i+1}}^{deg(x_{e_{i+1}})} .. x_{e_j}^{deg(x_{e_j})}.$$

The substitution transforms t in:

$$x_{e_0}^{\pi_0(x_{e_i})deg(x_{e_i}) + \pi_0(x_{e_{i+1}})deg(x_{e_{i+1}}) + .. + \pi_0(x_{e_j})deg(x_{e_j})}.$$

At the end of this process, P_v and P_u are univariate in x_{e_0} and are multiplied using *numpy.fft* method, that takes $P_v's$ and $P_u's$ list of coefficients as arguments (where a coefficient c_i of P_v is 0 if $x_{e_0}^i$ does not exist in P_v). The function *numpy.fft* returns a list of coefficients of the univariate polynomial result P'.

$$P' = \sum_{i=0}^{m} c_i x_{e_0}^i,$$

where $m = \max(\text{degree of } x_{e_0} \text{ in univariate } P_v) + \max(\text{degree of } x_{e_0} \text{ in univariate } P_u)$.

An inverse substitution step follows next, defined as a separate function, whose result is the multivariate polynomial we would have obtained by multivariate multiplication the original P_v and P_u. Each $x_{e_0}^i$ in P' becomes:

$$x_{e_0}^i \rightarrow x_{e_k}^{\lfloor \frac{i}{\pi_0(x_{e_k})} \rfloor} x_{e_{k-1}}^{\lfloor \frac{i\%\pi_0(x_{e_k})}{\pi_0(e_{k-1})} \rfloor} ..x_{e_1}^{\lfloor \frac{((i\%\pi_0(x_{e_k}))\%..)\%\pi_0(x_{e_2})}{\pi_0(x_{e_1})} \rfloor} x_{e_0}^{((i\%\pi_0(x_{e_k}))\%..)\%\pi_0(x_{e_1})},$$

where $\%$ is the *modulo* operation and $\lfloor x \rfloor$ is the *floor* of x.

Hence, each $c_i x_{e_0}^i$ in P' is transformed into a Term object for the $P_{vu}.terms$ list. Reduction is then applied to discard the common variables of P_v and P_u with odd degree.

Note that is this solution, contraction as in the multivariate multiplication approach is applied only once, as the last step, to $P = \prod_{i=0}^{n} P_i$. The result P is guaranteed to have all variables with even degree (due to the reduction step in multiplication). By contraction, each variable $x_{e_i}^{2k}$ in P becomes x^k in P_G.

3.6.3.2 Complexity analysis

Although we have tried this approach for a better time complexity, no significant progress has been made. The substitution for two initial polynomials can be done in $O(d2^d)$ complexity, with each polynomial having at most 2^d terms of maximum d variables each, using dictionaries for constant computation of each π value. The overall complexity is decided by the efficiency of the multiplication following the substitution, which is influenced by the degrees of the univariate polynomials.

If $\delta = max(degree(P_v), degree(P_u))$, then the multiplication takes $O(\delta lg\delta)$. time. However, δ increases significantly fast. In Example 3.6.3.3, we start with three multivariate polynomials and three variables of degree 1 each and in the second multplication, $\delta = 9$. While this is still a small number, we see substantial increases as n, d and b get bigger. For example, the last multplication for a $(6,3)$-graph with a 2-matching has $\delta = 12390$, while the fourth multplication of $(6,4)$-graph with 3-matching has $\delta = 1110997$. Since FFT involves computing the δ-root of unity, we cannot be sure of the accuracy of the results for δ of the order of millions. Therefore, Kronecker substitution is not a good solution for our problem.

3.6.3.3 Example

We provide next a short example of the Kronecker substitution solution for the (3,2)-graph in Figure 3.4.

Let the signature function be:

$$f = [1,1,0].$$

The vertices of the graph are $\{0,1,2\}$ with the following set of edges:

$$E(0) = \{e_0, e_2\}, E(1) = \{e_0, e_1\}, E(2) = \{e_1, e_2\}.$$

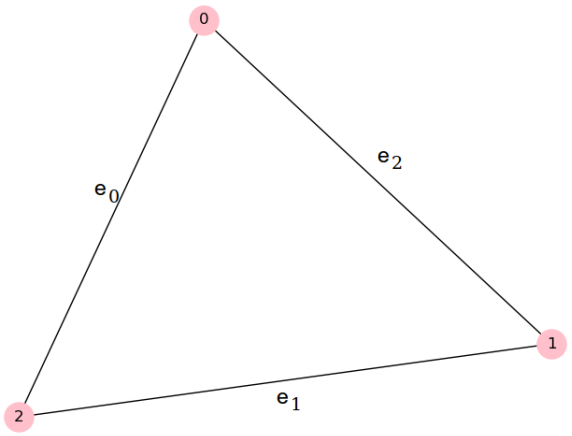

Figure 3.4: (3,2)-graph

Then the vertex polynomials are:

$$P_0 = 1 + x_{e_0} + x_{e_2},$$
$$P_1 = 1 + x_{e_0} + x_{e_1},$$
$$P_2 = 1 + x_{e_1} + x_{e_2}.$$

We do the multiplication of P_0 and P_1 first:

$$\pi(x_{e_0}) = \deg(x_{e_0}) \text{ in } P_0 + \deg(x_{e_0}) \text{ in } P_1 + 1 \Rightarrow \pi(x_{e_0}) = 1 + 1 + 1 \Rightarrow \pi(x_{e_0}) = 3.$$

Similarly,

$$\pi(x_{e_1}) = 2,\ \pi(x_{e_2}) = 2.$$

Then

$$\pi_0(x_{e_1}) = 3 \Rightarrow x_{e_1} \to x_{e_0}^3,$$
$$\pi_0(x_{e_2}) = 6 \Rightarrow x_{e_3} \to x_{e_0}^6.$$

After substitution,

$$P_0 = 1 + x_{e_0} + x_{e_0}^6,$$
$$P_1 = 1 + x_{e_0} + x_{e_0}^3,$$
$$P_0P_1 = 1 + 2x_{e_0} + x_{e_0}^2 + x_{e_0}^3 + x_{e_0}^4 + x_{e_0}^6 + x_{e_0}^7 + x_{e_0}^9.$$

Applying inverse substitution:

$$P_0P_1 = 1 + 2x_{e_0} + x_{e_0}^2 + x_{e_1} + x_{e_0e_1} + x_{e_2} + x_{e_0e_2} + x_{e_1e_2},$$

which is equal to $(1 + x_{e_0} + x_{e_2})(1 + x_{e_0} + x_{e_1})$ as expected. After reducing the terms where the common variable x_{e_0} is of odd degree:

$$P_0P_1 = 1 + x_{e_0}^2 + x_{e_1} + x_{e_2} + x_{e_1e_2},$$

Substitution for P_0P_1 and P_2 results in:

$$P_0P_1 = 1 + x_{e_0}^2 + x_{e_0}^3 + x_{e_0}^9 + x_{e_0}^3 x_{e_0}^9,$$
$$P_2 = 1 + x_{e_0}^3 + x_{e_0}^9,$$
$$P_0P_1P_2 = x_{e_0}^{21} + x_{e_0}^{18} + x_{e_0}^{15} + 3x_{e_0}^{12} + x_{e_0}^{11} + 2x_{e_0}^9 + x_{e_0}^6 + x_{e_0}^5 + 2x_{e_0}^3 + x_{e_0}^2 + 1.$$

Inverse substitution, reduction and contraction give us:

$$P_0P_1P_2 = x_{e_2}^2 + x_{e_1}^2 + x_{e_0}^2 + 1 \Rightarrow P_G = 1 + 3x.$$

3.6.4 Recursive implementation

The first three approaches discussed so far have been implemented. However, their slow execution, as a result of the polynomial-time complexity, did not allow us to run experiments for the n,d,b boundaries that have been initially proposed. More details on the behaviour of these approaches is presented in Section 3.6.5.

In need of a better algorithm, Dr. Heng Guo suggested a recursive formula to compute the coefficients Z_i of the global polynomials of $G = (V,E)$ an $(n,d)-$graph with a $b-$matching.

3.6.4.1 Algorithm overview

We define $M(\pi,i)$ a value depending on π and i, with $\pi : V \to \{0,1,..,b\}$ a constraint function on the number of edges incident to a vertex v. Then $M(\pi,i)$ is the number of matches with i edges such that each vertex v is matched with $\pi(v)$ edges.

If π_0 is the function such that $\pi_0(v) = 0, \forall v \in V$, then:

- $M(\pi_0,0) = 1$, since there is only one way to match each vertex with no edge when there are 0 edges.
- $M(\pi_0,k) = 0, 0 < k \leq |E|$, since we cannot use k edges to match each vertex with 0 edges.

Moreover, $M(\pi,i) = 0$ for $i < 0$ since matches cannot have negative size.

Our goal is, thus, to compute:

$$Z_i = M(\pi_b,i),$$

where $\pi_b(v) = b$, $\forall v \in V$.

The value of this coefficient can be obtained using the following recurrence relation:

$$M(\pi,i) = \sum_{S \in \rho_r(v)} M(\pi^{v \leftarrow 0 \text{ and } \forall u \in S, u \leftarrow \pi(u)-1}, i-|S|),$$

where:

$$v \text{ is an arbitrarily chosen vertex with } \pi(v) = r > 0,$$
$$\pi^{v \leftarrow \alpha} \text{ is a copy of } \pi \text{ with } \pi(v) = \alpha \text{ instead of the original value,}$$
$$\rho_r(v) = \{S \subseteq N(v), \text{ such that } |S| \leq r \text{ and } \forall s \in S, \pi(s) > 0\},$$
$$N(v) \text{ the set of vertices adjacent to } v \text{ (neighbors).}$$

Intuitively, the recursion comes from considering that v can be matched with:

- 0 edges (0 adjacent vertices), resulting in recurrence term $M(\pi^{v \leftarrow 0}, 0)$.
- 1 edge (1 of its neighbors), resulting in the recurrence terms $M(\pi^{v \leftarrow 0, u \leftarrow \pi(u)-1}, i-1)$ for every neighbor u with $\pi(u) > 0$. Updating $\pi(v) = 0$ ensures that v is not considered again in the matching process down the recurrence of the currently considered term.

- ...and similarly up to...

- r edges, where $r = \pi(v)$, resulting in the recurrence terms $M(\pi^{v \leftarrow 0, u_1 \leftarrow \pi(u_1)-1, .., u_r \leftarrow \pi(u_r)-1}, i-r)$ for every possible way to pick $\{u_1, .., u_r\}$ neighbors of v with $\pi(u_i) > 0$.

Algorithm 12 presents the main body of the solution, calling the recursive function *get_m_coefficient* for each Z_i. In this solution, we use the optimization technique of *memoization* to speed up the computations. Memoization involves saving values that are returned by a function call and retrieving them from the saved cache every time they are needed again, avoiding the extra time necessary to re-run the function with the same parameters. A *saved_m_coefficients* dictionary to save $M(\pi, i)$ values already calculated as a result of a *get_m_coefficient* function call has been used. The keys of the dictionary, returned by an auxiliary function, are represented by a String formed from the concatenation of the *get_m_coefficient* input.

Algorithm 12: Global polynomial coefficients using the recursive solution substitution

---

Input: G, an $(n,d)-$graph, b maximum matching constraint;
Output: a list of coefficients of G's global polynomial;
initialize π;
initialize global saved_m_coefficients dictionary;
for $v \in G.nodes$ **do**
    $\pi(v) = b$;
end
for *i from 0 to* $\frac{n \cdot d}{2}$ **do**
    Z_i = get_m_coefficient(G.nodes, π, i);
end

The body of the *get_m_coefficient* function is presented in Algorithm 13. The function checks first if it is in any of the base cases where it should return 0 or 1. For simplicity, the solution keeps a list V of vertices that can be picked at each recursion level, such that for each $v \in V, \pi(v) > 0$. Thus, V is actually a subset of the keys of dictionary π. Then π_0 defined earlier corresponds to an empty V list. If the function call is not a base case, the algorithm checks if it can retrieve the value from the *saved_m_coefficients* dictionary. Otherwise, it picks one v with $\pi(v) = r$ from V (simply done by retrieving the first element of the list) and iterate through the sets in $\rho_r(v)$ to generate each M term in the recursive relation of the current $M(\pi, i)$. The ρ set is computed using Python's $itertools.combinations(iterable, r)$ that returns subsets of length r of elements from the input $iterable$ (list of neighbours), in lexicographic order.

Algorithm 13: The recursive *get_m_coefficient* function

Input: V, a list of vertices that are allowed to be picked, π, i;
Output: value of $M(\pi, i)$;
if *i <0* **then**
  return 0;
end
if *V is empty* **then**
  if *i == 0* **then**
    return 1;
  else
    return 0;
  end
end
k = key(V, π, i);
if *k in saved_m_coefficients.keys* **then**
  return saved_m_coefficients(k)
end
pick $v \in V$ with $\pi(v) > 0$;
r = $\pi(v)$, M=0;
π' = copy of π, $\pi'(v) = 0$;
for *k from 0 to r* **do**
  $\rho_k(v)$ = subset of $\rho_r(v)$ such that for each $S \in \rho_r(v)$, $|S| = k$;
  for $S \in \rho_k(v)$ **do**
    initialize V' list of vertices not allowed to be picked anymore;
    π'' = copy of π';
    for $u \in S$ **do**
      $\pi''(u) = \pi''(u) - 1$;
      if $\pi''(u) = 0$ **then**
        add u to V'
      end
    end
    M = M + get_m_coefficient(V-$\{v\}$-V', π'', i-k)
  end
end
saved_m_coefficients(k)=M;
return M;

It is important to create a deep copy of the π object for every new call of the recursive function. The deep copy creates a new dictionary with the same values as the original one, but each with their own memory location. This step is essential, as the dictionary is updated, corresponding to constraints in matching vertices, for every new call of *get_m_coefficient* and we do not want a change of dictionary in one call to propagate to another call (what would happen if the function calls would share the same object).

3.6.4.2 Complexity analysis

We analyse the complexity of the algorithm presented above. In order to get one $M(\pi, i)$ value, the recursive formula tells us that we need to do at most $|\rho_r(v)|$ other function calls, $r \leq b \leq d$. For a vertex v with at $N(v)$ the set of neighbors, $|N(v)| = d$ and $\rho_r(v) \subseteq N(v)$, then $|\rho_r(v)| \leq 2^d$. In fact, $|\rho_r(v)| \leq |\rho_b(v)| = \sum_{i=0}^{b} \binom{d}{i}$ and $\sum_{i=0}^{b} \binom{d}{i} = 2^d$ when $b = d$. The number of internal states for one i, whose values are saved in *saved_m_coefficients* for fast retrieval, depends on the total number of π functions that we can have. Since π is a function defined on the set of vertices V, $|V| = n$ with values in $\{0, 1, 2, .., b\}$, then the number of π functions is $(b+1)^n$. Moreover, setting up each new copy of the π dictionary for a new recursive call takes $n \cdot b$ steps (n keys to be copied, at most b values to be update). Enumerating all Z_i coefficients, for $i \leq |E| = \frac{n \cdot d}{2}$, results in an algorithm of time complexity $O(n^2 d b 2^d (b+1)^n)$.

3.6.4.3 Example

We provide a short example of the recursive solution for the (3,2)-graph in Figure 3.5.

Let the signature function be a 1-matching,

$$f = [1, 1, 0].$$

The vertices of the graph are $\{v_0, v_1, v_2\}$ with the following neighbor sets:

$$N(v_0) = \{v_1, v_2\}, N(v_1) = \{v_0, v_2\}, N(v_2) = \{v_0, v_1\}.$$

The initial π function is:

$$\pi(v_0) = 1, \pi(v_1) = 1, \pi(v_2) = 1.$$

For convenience, we notate π using the format:

$$\pi(v_0 : 1, v_1 : 1, v_2 : 1).$$

We need to find Z_0, Z_1, Z_2, Z_3.

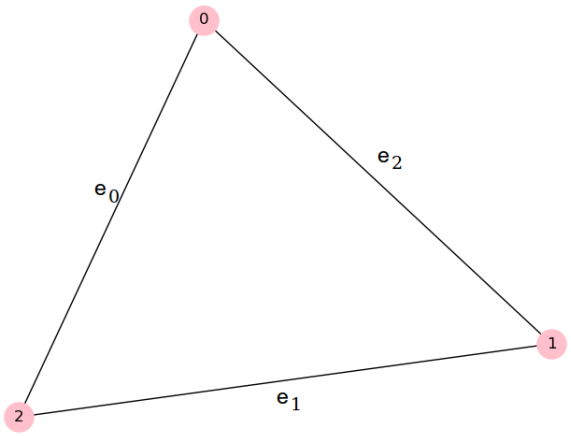

Figure 3.5: (3,2)-graph

- To compute Z_0:

$$Z_0 = M(\pi(v_0:1, v_1:1, v_2:1), 0) \overset{pick-v_0}{=} M(\pi(v_0:0, v_1:1, v_2:1), 0) +$$
$$+ M(\pi(v_0:0, v_1:0, v_2:1), -1) + M(\pi(v_0:0, v_1:1, v_2:0), -1),$$

where

$$M(\pi(v_0:0, v_1:1, v_2:1), 0) \overset{pick-v_1}{=} (M(\pi(v_0:0, v_1:0, v_2:1), 0) + M(\pi(v_0:0, v_1:0, v_2:0, -1)),$$

where

$$M(\pi(v_0:0, v_1:0, v_2:1), 0) \overset{pick-v_2}{=} (M(\pi(v_0:0, v_1:0, v_2:0), 0) = 1, [*]$$

and

$$M(\pi, -1) = 0.$$

Hence:

$$Z_0 = 1.$$

- To compute Z_1:

$$Z_1 = M(\pi(v_0:1, v_1:1, v_2:1), 1) \overset{pick-v_0}{=} M(\pi(v_0:0, v_1:1, v_2:1), 1) +$$
$$+ M(\pi(v_0:0, v_1:0, v_2:1), 0) + M(\pi(v_0:0, v_1:1, v_2:0), 0),$$

where

$$M(\pi(v_0:0, v_1:1, v_2:1), 1) \overset{pick-v_1}{=} (M(\pi(v_0:0, v_1:0, v_2:1), 1) + M(\pi(v_0:0, v_1:0, v_2:0, 0) =$$

$$\overset{pick-v_2}{=} (M(\pi(v_0:0, v_1:0, v_2:0), 1) + 1 = 0 + 1 = 1.$$

As shown in [*],

$$M(\pi(v_0:0, v_1:0, v_2:1), 0) = 1$$

and with a similar approach:

$$M(\pi(v_0:0, v_1:1, v_2:0), 0) = 1.$$

Hence:

$$Z_1 = 3$$

- Similar procedure leads to:

$$Z_2 = 0,$$

$$Z_3 = 0.$$

The global polynomial is:

$$P_G = 1 + 3x.$$

3.6.5 Comparisons

The approaches presented in this section have been implemented. Apart from comparing their running time for deciding which algorithm to use for the experiments, having more solutions has offered a sanity check that their implementation is correct by outputting the same results for the same input. The cases when $b \geq d$ have not been considered, since the solution is straightforward using binomial expansion.

Table 3.2 presents the time in seconds to generate the global polynomial for a given $(n,d)-$graph and a 3-matching. The same graph is used for all 4 tests for given n and d and the time to generate the graph is not considered. For these tests, the generation has been stopped in the program did not finish running in 6 minutes.

Algorithm	n	d	Time
Naive	6	3	0.01062
Multivariate multiplication	6	3	0.01544
Kronecker substitution	6	3	0.05690
Recursion	6	3	0.02321
Naive	6	4	0.07107
Multivariate multiplication	6	4	0.08040
Kronecker substitution	6	4	2.84611
Recursion	6	4	0.05032
Naive	6	5	0.42795
Multivariate multiplication	6	5	0.62920
Kronecker substitution	6	5	>360, stopped
Recursion	6	5	0.10413
Naive	7	4	0.39130
Multivariate multiplication	7	4	0.35944
Kronecker substitution	7	4	>360, stopped
Recursion	7	4	0.17318
Naive	7	6	99.94407
Multivariate multiplication	7	6	68.17199
Kronecker substitution	7	6	>360, stopped
Recursion	7	6	1.12883
Naive	8	4	4.69729
Multivariate multiplication	8	4	5.74345
Kronecker substitution	8	4	>360, stopped
Recursion	8	4	0.71434
Naive	8	5	31.01767
Multivariate multiplication	8	5	59.28918
Kronecker substitution	8	5	>360, stopped
Recursion	8	5	0.89132

Table 3.2: Time in s to compute global polynomial for a 3-matching

The scrips for these tests run on a Dice machine, with processor Intel Core i5-6500 CPU 3.2GHz x 4. Similar results could be observed using the personal laptop with processor Intel Core i5-6200U CPU 2.30GHz x 4.

As expected, Kronecker substitution is not a reliable solution to use, performing significantly worse compared to the three other approaches. The naive and the multivariate solutions have similar performance, while the recursion is remarkably faster.

The execution of these tests stops at $n = 8$, since the naive and the multivariate algorithms cannot run on neither Dice nor personal laptop for $n \geq 9$, due to the memory limits of the machines. However, the recursive method could run on Dice machines in less than 6 minutes for $n \leq 12$ and $d \leq 7$. Table 3.3 shows the time in seconds required to compute the global polynomial for the given configuration, with the purpose of highlighting the efficiency of the algorithm for the required bounds.

n	d	b	Time
12	6	4	88.06828
12	6	5	138.09138
12	7	4	208.96234
12	7	5	347.07679

Table 3.3: Time in s to compute global polynomial using recursion

Dealing with the memory limit of the Dice machines and with the realitvley high number of experiments, as presented in Chapter 4, the algorithms were run on the *student.compute* servers of the university as well, with processor Intel(R) Xeon(R) CPU E5-2690 CPU 3.00GHz x 10, available for extensive computational jobs. While these servers are more powerful, they incur some overhead with scheduling different tasks that affects the total running time of a process. Table 3.4 presents the time in seconds taken to run the algorithm for different n, d, b configurations using $student.compute$.

n	d	b	Time
12	7	4	594.50146
12	7	5	1030.21120
16	9	4	5203.46432
16	9	5	13824.14057
18	5	3	19646.47276
18	5	4	44417.53407

Table 3.4: Time in s to compute global polynomial using student.compute servers

Pushing the boundaries to $n = 18$ resulted in an algorithm that run almost 12 hours. In contrast, the multivariate polynomial for $n = 9, d = 6, b = 5$ took a bit over 12 hours to run on these servers, while this configuration can be run in 12 seconds by the recursive algorithm. Considering the running time of the algorithms, it is obvious that the recursive approach has the best performance. As a result, it has been used to run all the experiments presented in the Chapter 4.

Chapter 4

Experiments and results

This chapter presents the experiments that have been made and analyses their results. The aim of an experiment is to return the roots of a graph polynomial from a d-regular graphs of n vertices (defined in Section 2.2), given a b-matching signature function (defined in Section 2.3). Based on Theorem 2.6.2, we expect to see a distribution of the roots outside a $[-\theta', \theta']$ cone, where θ' is an angle that depends on the local polynomial.

4.1 Graphs

The graphs have been generated using the algorithm described in Section 3.4.4. The high number of regular graphs (as presented in Appendix B), the low probability to generate random regular graph and the complexity of the global polynomial algorithms have influenced the decision to set a limit for the number of vertices n and the degree d for our experiments. Although the initial goal was to analyse the roots of 7-regular graphs with a maximum of 12 vertices, the more efficient algorithm for generating the global polynomial described in Section 3.6.4 has allowed us to slightly increase the limit. In the end, a maximum of 20 non-isomorphic regular graphs have been used from each $G_{n,d}$ group of the proposed goal, $4 \leq n \leq 12$ and $2 \leq d \leq 7$ (with less than 20 if $|G_{n,d}| < 20$) and one graph for the following n and d pairs, extending the boundaries initially set as goal:

- $9 \leq n \leq 12$, $8 \leq d \leq 11$
- $13 \leq n \leq 14$, $2 \leq d \leq 6$
- $15 \leq n \leq 20$, $2 \leq d \leq 5$.

The adjacency lists of the generated graphs have been saved to separate files. The graphs can be reconstructed from the files using the NetworkX library, to ensure that the resuts of the experiments can be reproduced on the same data.

4.2 Local polynomials

The local polynomial is a truncated binomial expansion that depends on the degree d and the b-matching signature function. For every local polynomial, the corresponding $\theta' = \pi - 2\theta$ angle defined in Theorem 2.6.2 has been computed. The θ angle depends on the maximum of $|arg(z_f)|$, where $\{z_f\}$ is the set of roots of the local polynomials. Figure 4.1 shows an example of the θ and θ' angles for the local polynomials of degree $d = 7$ with matchings of 4 and 5. The complex roots of the polynomials are scattered in the plots, with the real part on the $x-axis$ and the imaginary part on the $y-axis$.

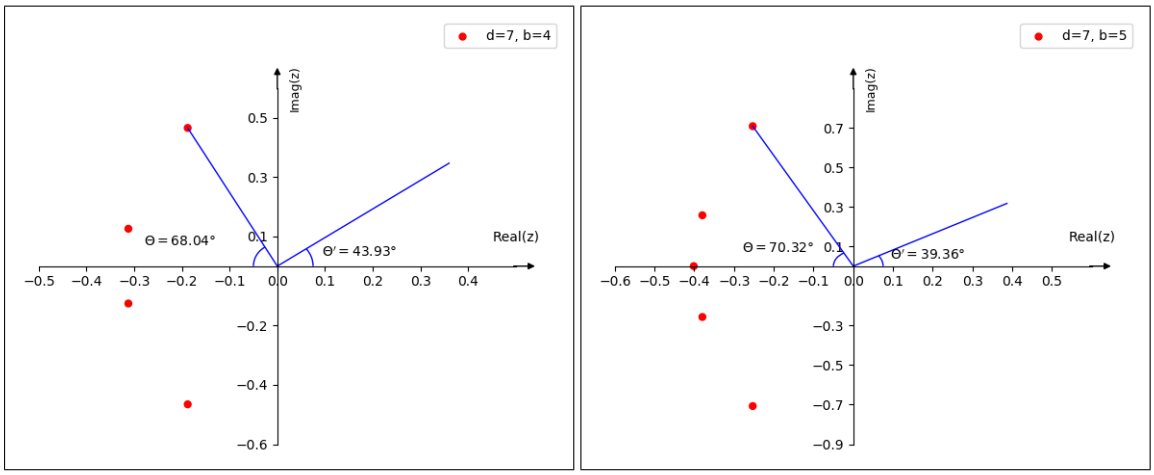

Figure 4.1: θ and θ' angle of local polynomials

Table 4.1 provides the θ and θ' angles for all the local polynomials that have been generated, with d and b corresponding to the boundaries of our experiments. The values have been rounded to the nearest two decimals. As a result, the sum $2\theta + \theta'$ might be slightly different than $180°$. The polynomials are presented in Appendix C.

The cone with the greatest angle $\theta' = 120°$ is associated with 3-regular graphs with a 2-matching constraint, while the cone with the smallest angle $\theta' = 13.97°$ corresponds to the largest configurations, from 11-regular graphs with a 5-matching constraint.

We have only consider the cases when $d > b \geq 2$. For $d \leq b$, the local polynomial is:

$$P_f(x) = (1+x)^d,$$

with the root $x = -1$ and $\theta = 0$, $\theta' = 180°$. For $b = 1$, the local polynomial is:

$$P_f(x) = 1 + dx,$$

with the root $x = -\frac{1}{d}$ and $\theta = 0$, $\theta' = 180°$. These two cases do not give a meaningful limit for the roots of the global polynomial.

Moreover, for the boundaries we have experimented with, $d \leq 11$ and $b \leq 8$, the roots of the local polynomials lie on the left half-plane. This property is not true for $d \geq 12$ and certain values of b, implying that the local polynomial is not H_ε-stable (from the discussion following Definition 2.6.1) and that there is no angle $\theta \in [0, \frac{\pi}{2}]$ satisfying

$|arg(z_f)| \in [\pi - \theta, \pi]$, for z_f roots of the polynomial. Therefore, we cannot apply Lemma 2.6.1 or Theorem 2.6.2 for the d and b values that give a polynomial with roots in the right half-plane.

d	b	θ	θ'
3	2	30.00°	120°
4	2	35.26°	109.47°
4	3	45.00°	90.00°
5	2	37.76°	104.48°
5	3	52.79°	74.43°
5	4	54.00°	72.00°
6	2	39.23°	101.54°
6	3	56.65°	66.71°
6	4	63.30°	53.40°
6	5	60.00°	60.00°
7	2	40.20°	99.60°
7	3	58.98°	62.04°
7	4	68.04°	43.93°
7	5	70.32°	39.36°
7	6	64.29°	51.43°
8	2	40.89°	98.21°
8	3	60.56°	58.89°
8	4	70.96°	38.09°
8	5	75.67°	28.66°
8	6	75.34°	29.33°
8	7	67.50°	45.00°
9	2	41.41°	97.18°
9	3	61.69°	56.52°
9	4	72.95°	34.10°
9	5	79.01°	21.97°
9	6	81.15°	17.71°
9	7	79.10°	21.80°
9	8	70.00°	40.00°
10	2	41.81°	96.38°
10	3	62.55°	54.90°
10	4	74.40°	31.19°
10	5	81.32°	17.36°
11	2	42.13°	95.74°
11	3	63.22°	53.56°
11	4	75.51°	28.98°
11	5	83.01°	13.97°

Table 4.1: θ and corresponding θ' angles of local polynomials.

4.3 Global polynomials

A total of 800 global polynomials have been generated, not including the cases where $b \geq d$. Global polynomials have been computed for all the selected graphs (described in Section 4.1), for b-matching signature functions The initial goal has been to experiment with $b \in \{2,3,4,5\}$. However, the interesting distribution of the roots of the polynomials, described in Section 4.4 has encouraged us to check the patterns for higher values of b as well. Hence, we have increased b to 8 for $d \leq 9$ and $n \leq 11$ and computed one polynomial for each possible configuration of these increased boundaries. For further clarification, the table in Appendix E of the degrees of the global polynomials, discussed in 4.3.2, highlights the applied (n,d,b) configurations.

The experiments have been automated with Python scripts and all the results have been saved for future references. Each result file contains the adjacency list of the graph, the list of coefficients of the global polynomial and the list of the roots of the polynomial. Due to the complexity of the algorithms, polynomials for the graphs with $d \geq 8$ or $n \geq 12$ have been computed on the *student.compute* servers of School of Informatics, whose performance was presented in Section 3.6.5.

4.3.1 Values of the coefficients

As described at the end of Section 2.5, the coefficients Z_i for $0 \leq i \leq b$ correspond to the binomial coefficient $\binom{|E|}{b}$, where $|E| = \frac{n \cdot d}{2}$ is the number of edges of the graph. As a result, $Z_0 = 1$ and $Z_1 = \frac{n \cdot d}{2}$ for any graph. Moreover, since the coefficients represent a number of matchings, then each coefficient is a positive integer.

The results have shown that the coefficients of the polynomials tend to become large with the increase in the number of vertices (but they respect the upper limit $\binom{|E|}{i}$ proved in Section 2.5). This fact raises the concern of the accuracy of the root solving algorithms. Unfortunately, we acknowledge that the inability to establish the precision of NumPy's polynomial solver is a limitation of this project. Example 4.3.1 shows the scale of these coefficients for a 2-matching on 10 and 12-vertices graphs, with an increase in the matching size or number of vertices resulting in even larger coefficients. The graphs used in this example can be seen in Appendix D.

Example 4.3.1. The list of coefficients for (10,7)-graph with a 2-matching, from the leading coefficient to Z_0, is:

$$[23772, 350973, 913185, 972120, 552706, 190197, 42210, 6195, 595, 35, 1],$$

while the list of coefficients for (12,7)-graph with a 2-matching, from the leading coefficient to Z_0, is:

$$[160220, 3366410, 12399489, 18761468, 15326848,$$

$$7717277, 2577795, 595266, 96810, 11060, 861, 42, 1].$$

4.3.2 Degree of the polynomials

It has been notice that, for all the non-isomorphic graphs in $G_{n,d}$ (which have been used for experiments) and a given b, the resulting polynomials have the same degree. A complete list of the degrees is provided in Appendix E. Analysing the degree of the polynomials has resulted in the following interesting experimental observations:

Observation 4.3.1. The degree $deg(P_G)$ of the global polynomial of a d-regular graph, $d \geq 2$ with number of vertices n even and a b-matching, $2 \leq b \leq min(5, d)$, is:

$$deg(P_G) = n + \frac{n}{2}(b-2).$$

Observation 4.3.2. The degree $deg(P_G)$ of the global polynomial of a d-regular graph, $d \geq 2$, with number of vertices n odd and a b-matching, $2 \leq b \leq min(5, d)$, is:

$$deg(P_G) = n + \frac{n-1}{2}(b-2) + \lfloor \frac{b-2}{2} \rfloor.$$

Note that these observations are true for all the experiments we have conducted, however we cannot say whether it holds for graphs that have not been considered.

4.3.3 Polynomials for graphs in the same group

Although isomorphic graphs have the same global polynomial, the converse of this property is not true. We have found the following examples of non-isomorphic graphs that resulted in the same global polynomial:

- One pair of graphs with $n = 10$, $d = 4$, $b = 2$.
- Two pairs of graphs with $n = 10$, $d = 4$, $b = 3$.
- One pair of graphs with $n = 12$, $d = 3$, $b = 2$.

Note that for this observation we rely on the accuracy of NetworkX's *is_isomorphic* function to check for isomorphism. The adjacency lists of these graphs are presented in Appendix F.

4.4 Roots of the global polynomials

Additional Python scripts have been written to plot and analyse the roots. In particular, we have been interested in the distribution of the roots, according to Theorem 2.6.2. It has been confirmed that all the roots satisfy the theorem and are not present in the corresponding $[-\theta', \theta]$ cone. This result has been proved by checking that, for each root z_G of the local polynomial, $|arg(z_G)| \in [\theta', \pi]$. Moreover, since the local polynomials for our constraints are H_ε-stable, the roots are also consistent with Lemma 2.6.1.

Figure 4.2 shows an example of the distribution of the roots for a (12,5)-graph with a 4-matching. The complex roots of the polynomials are in a scatter plot, with the real part on the $x-axis$ and the imaginary part on the $y-axis$. Note that the complex roots of a polynomial appear in conjugate pairs. The graph and the coefficients of the polynomial for this example are presented in Appendix G and the distribution of these roots in discussed in the next subsection.

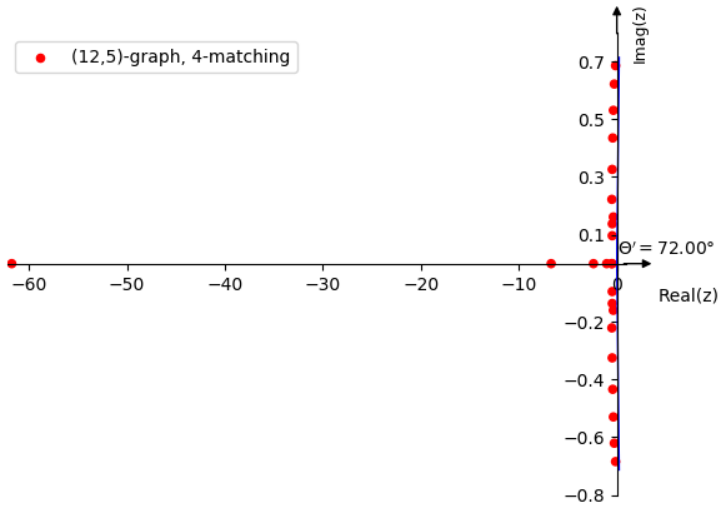

Figure 4.2: Root of global polynomial of a (12,5)-graph with a 4-matching

4.4.1 Distribution of the roots relative to the unit circle

In Figure 4.2, the complex roots are clustered around a tight region with a small negative real part, with two real roots clearly outside this area. The generated polynomials for $b \leq 5$ have actually displayed the same behaviour: all the complex roots with non-negative real part have the modulus less than 1. While there exist real roots both inside and outside the unit circle, for each configuration there exists at least one real root significantly far outside this range.

Much research is devoted to understanding the geometrical properties of the roots of different type of polynomials, as a function of their coefficients, with papers such as [SV95] and [HN08] stating that the complex roots of random polynomials asymptotically tend to be uniformly distributed close to the unit circle. However, we do not have an explanation why, for the global polynomials of matchings of size up to 5, the complex roots in $\mathbb{C} - \mathbb{R}$ lie inside the unit circle.

This observation has raised the question whether the assumption that, for polynomials obtained from matchings of any size, the complex roots follow the same distribution. Motivated by this question, we have generated a few examples for $b \in \{6,7,8\}$ that have contradicted the assumption. In particular, the following (n,d,b) configurations have resulted in graph polynomials with complex whose distance from the origin is bigger than 1:

- $(8,7,6),(9,8,7),(10,7,6),(10,8,7),(10,9,8),(11,8,7)$.

For examples, Figure 4.3 show the distribution for the global polynomial of $(10,9)$-graph with an 8-matching. In the right subfigure, the extreme roots (far from the origin) have been purposely removed for a better visualisation of the tight area near the origin.

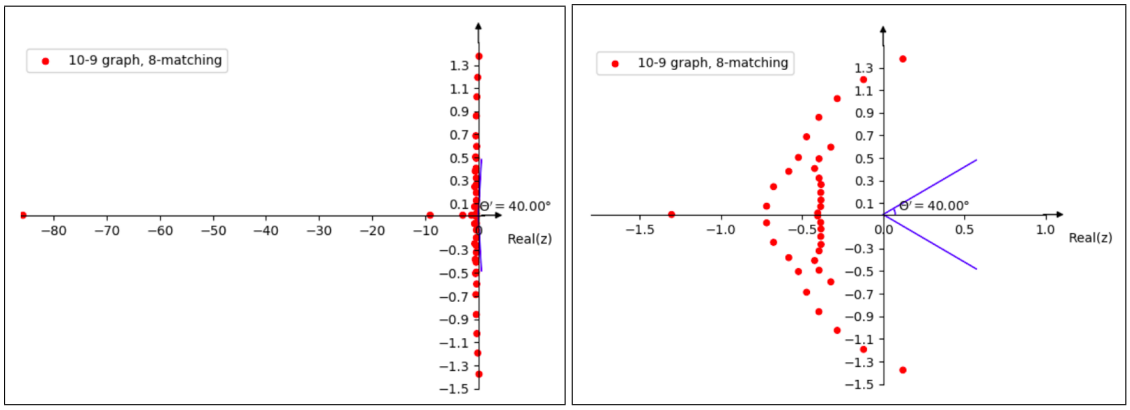

Figure 4.3: Roots of global polynomial

The examples of configurations with complex roots whose modulus is bigger than 1 suggest that such a behaviour appears for configurations of larger values. In fact, we have noticed that the polynomials of the same graph have complex roots with the imaginary part further away from the origin with the increase of the matching constraint function. An example is provided in Figure 4.4 that displays the roots of polynomials

of a $(12,6)$-graph. A constraint function with a bigger b limit results in a more significant variance of the imaginary part of the complex roots, rather than the real parts. Please notice that is essential to view a coloured version of the plot. Moreover, as in the previous example, the extreme roots have been removed.

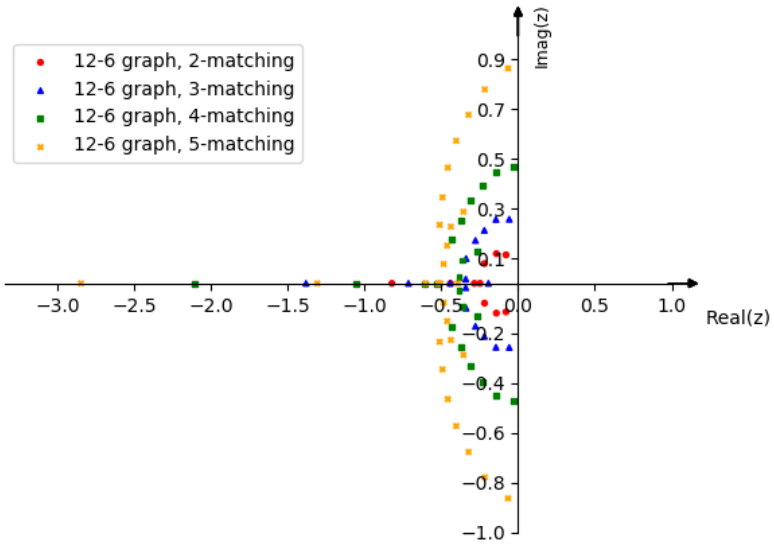

Figure 4.4: Roots of global polynomials for the same graph, increasing the constraint

4.4.2 Bounds of the roots

Figures 4.2 and 4.3 clearly show a considerable distance between the closest and the furthest root to the origin. Trying to understand the limit of the modulus of the roots, we present the Kakeya's theorem stated in [GG14] and proved by Kakeya in [Kak12]:

Theorem 4.4.1. *If $p(z) = \sum_{i=0}^{\delta} c_i z^i$ is a polynomial of degree δ with real and positive coefficients, then all the zeros of p lie in the annulus (region bounded by two concentric circles)*

$$R_1 \leq |z| \leq R_2,$$

where $R_1 = min_{0\leq j\leq \delta-1} \frac{c_j}{c_{j+1}}$ and $R_2 = max_{0\leq j\leq \delta-1} \frac{c_j}{c_{j+1}}$.

Interestingly, our tests have shown us that, for each graph considered:

$$R_1 = \frac{c_0}{c_1} = \frac{1}{|E|} = \frac{2}{n\cdot d} \text{ and } R_2 = \frac{c_{\delta-1}}{c_\delta},$$

with

$$R_1 < 1 < R_2,$$

where $\delta = deg(P_G)$.

When the difference $R_2 - R_1$ is significant, we can intuitively only expect to see a high distance between the closest and the furthest root from the origin, which has been

noticed in our results. For the graph used in Figure 4.2, for example, $R_1 \approx 0.033$ and $R_2 \approx 76.61$.

The big difference between $min(|Real(z_i)|)$ and $max(|Real(z_i)|)$ for z_i roots, the fact that $min(|Real(z_i)|)$ is very close to the origin and that many roots in $\mathbb{C} - \mathbb{R}$ are clustered inside a tight region close to the origin compared to the real roots result in a difficulty to properly visualise the maximum angle between the roots and the real axis. For this reason, the real roots with absolute value bigger than $avg(|Real(z_i)|)$ have not been shown in the rest of the plots.

Lastly, taking into consideration the fact that the global polynomials have positive coefficients, it is obvious that there are no positive real roots for any of the (n,d,b) configurations.

4.4.3 Position of the roots with respect to the local cone

The roots of the polynomials generated for our experiments are consistent with Theorem 2.6.2, being distributed outside a $[-\theta', \theta']$ cone, where the values of the angle θ' have been previously provided in Table 4.1. The table suggests that the angles with the greatest values, imposing a tighter limit of the roots, correspond with 2-matchings. The plots we have produced for a better visual analysis of the results suggest that the roots are close to the limit for $b = 2$ and relatively far from it for $b \in \{3,4,5,6,7,8\}$. Although it is not feasible to include the plots for the full range of produced polynomials, we provide in the Figures 4.5 to 4.8 two examples from graphs of larger size.

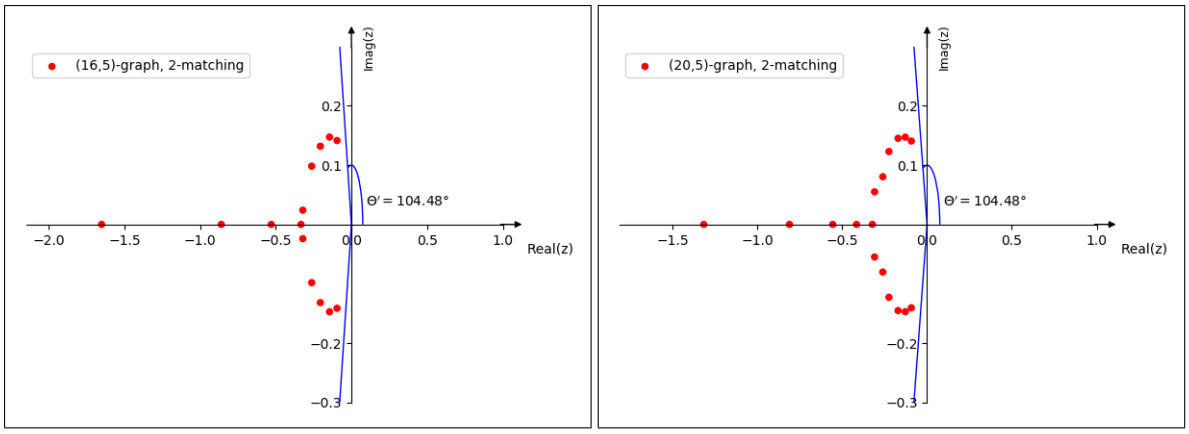

Figure 4.5: Roots of global polynomials, 2-matching

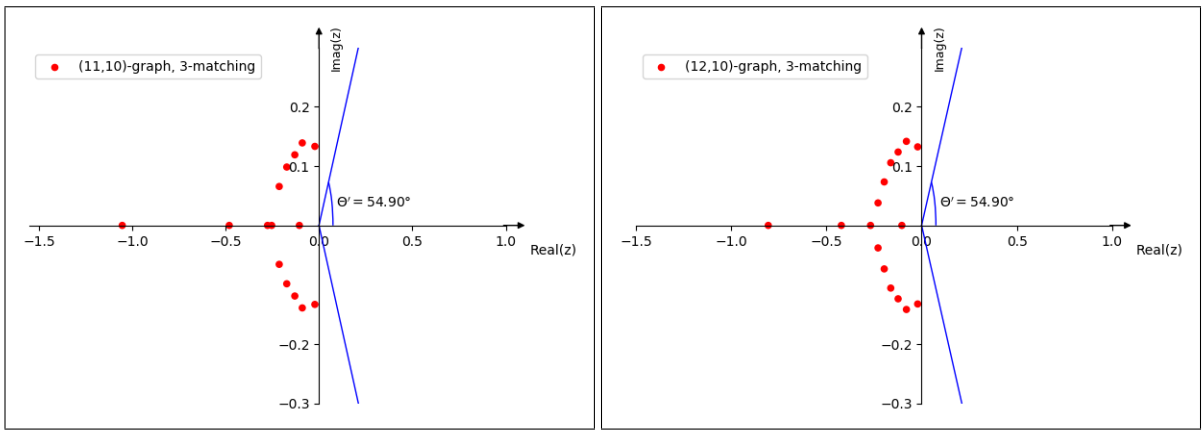

Figure 4.6: Roots of global polynomials, 3-matching

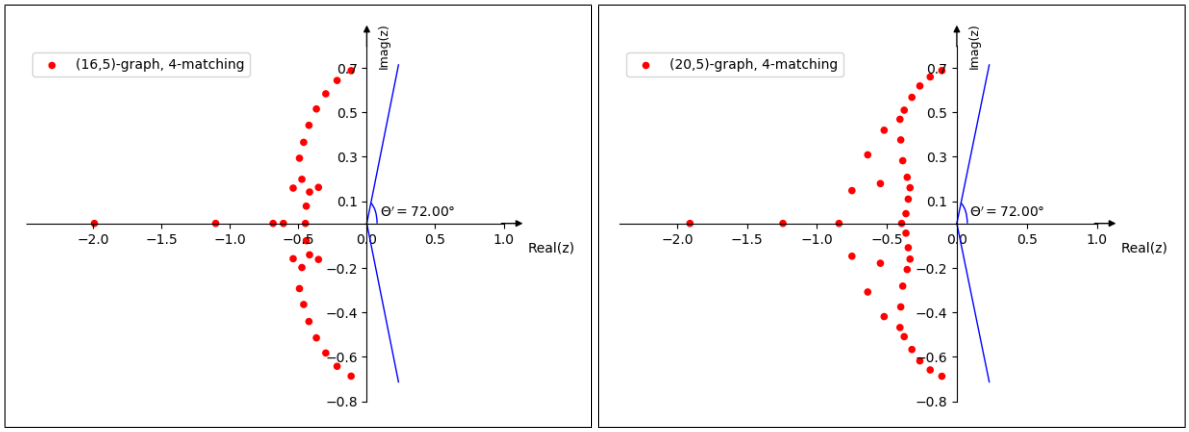

Figure 4.7: Roots of global polynomials, 4-matching

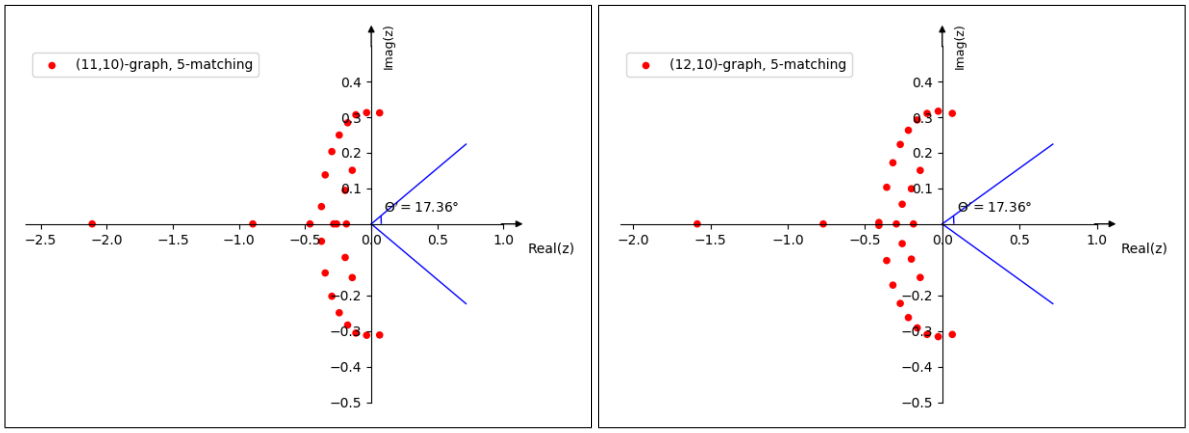

Figure 4.8: Roots of global polynomials, 5-matching

We have provided in Appendix H the distance in angles from the corresponding limit given by the local polynomial to the root with the maximum argument, averaged for all the polynomials of graphs in the same group. The table suggests that the distance is not monotone with the increase of the matching limit. In fact, it increases up to a point and then starts decreasing. The distance is, however, bigger for regular graphs

of higher degree, with the same number of vertices and constraint function. We also remind that the global polynomial for $b \geq d$ is a binomial expansion with root -1.

Lastly, as it could be noticed in the Figures 4.5 to 4.8 and inferred from the table in Appendix H, the roots of polynomials from graphs with the same degree and same constraint have similar distributions and bounds of the roots in $\mathbb{C} - \mathbb{R}$, resulting in similar distances from the associated cone, considering that the limit does not the depend on the number of edges. An example for the polynomials of 8-regular graphs with 5-matchings is provided in Figure 4.9, where the number of edges has been varied from 9 to 12. The roots clearly follow a similar pattern, making it difficult to properly see the differences. Please consider having a coloured version of the plot for a better visualisation.

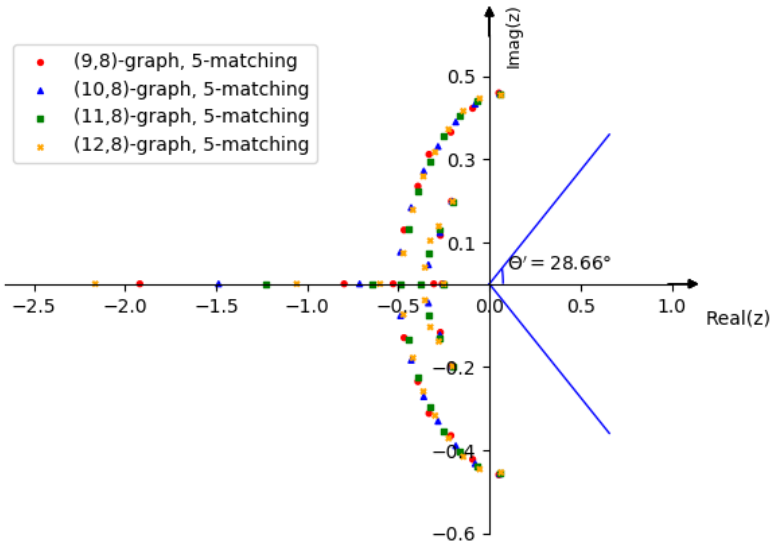

Figure 4.9: Roots of global polynomials for the same (d, b) configuration

We have also noticed that for $b \in \{2, 3\}$, the roots in $\mathbb{C} - \mathbb{R}$ tend to form and arc, while this is not the case for $b \geq 4$. Compare, for example, the distribution of the roots in Figure 4.6 with Figure 4.8.

4.4.4 Roots of non-isomorphic graphs in the same group

Visualisation through plotting has shown that the distribution of the roots of the global polynomials of two non-isomorphic graph in the same $G_{n,d}$ group are very similar. See, for example, in Figure 4.10 the roots of two polynomials of graphs in $G_{12,7}$ for a 5-matching. The differences in the two plots are very difficult to spot. This behaviour could be explained by the fact that two polynomials of non-isomorphic graphs in the same group have coefficients of very similar magnitude and relatively low differences compared to their significantly high values. For the plots in Figure 4.10, for examples, with roots of polynomials of degree 30, a subset of the coefficients of the left polynomials is

$$Z_{30} = 160220, Z_{26} = 8531510892, Z_{16} = 150459467640,$$

while for the same indices, the right polynomial has the coefficients

$$Z_{30} = 160447, Z_{26} = 8535509103, Z_{16} = 150459470190.$$

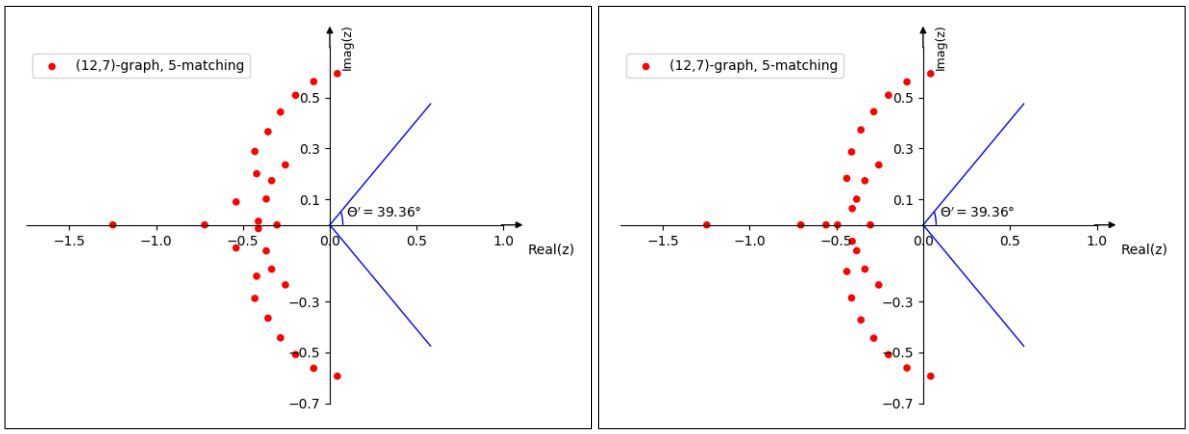

Figure 4.10: Roots of non-isomorphic graphs in the same group, 5-matching

4.4.5 Roots in the right half-plane

Lastly, in their paper [LPRS16], Lebowitz, Pittel, Ruelle and Speer discuss zeros of graph-counting polynomials and give an example for 4-matchings, stating that they do not have a polynomial with roots in the right half plane. Although the first of our results, for small values of n, were consistent with this affirmation, we have actually found an important number of counter-examples. Table 4.2 presents the configurations that resulted in global polynomials with roots in both the left and the right half-planes (note that a 5-matching has not been run for $n = 12, d = 11$). This behaviour can be observed only for bigger values for n, d and b. Furthermore, these results were true for all the graphs in a G_n, d we experimented with for the values in Table 4.2. It would be interesting to experiment with even bigger values to see how the patterns continues. These observations leave space for more analysis in this area.

n	d	b
8	7	5
9	8	4, 5, 6, 7
10	7	5
10	8	4, 5, 6, 7
10	9	4, 5, 6, 7, 8
11	8	4, 5, 6, 7

n	d	b
11	10	4,5
12	7	4,5
12	8	4,5
12	9	4,5
12	10	4,5
12	11	4

Table 4.2: Values with global polynomials with roots in the left and right half-planes

Chapter 5

Conclusion

5.1 Summary

This paper has presented experimental results of the roots of graph polynomials of regular graphs with b-matching signature function. Different approaches to generate the graphs and compute the global polynomials have been implemented and compared, as described in Section 3.4 and 3.6, and the experiments we have originally aimed for have been successfully carried out. Moreover, the algorithm chosen for returning the global polynomials for all the experiments have allowed an increase in the number of vertices of the graphs from 12, the initial goal, to 20. The complexity of the algorithms, however, has imposed a superior limit on the size of the graphs, with the longest run of the global polynomial generator taking almost 12 hours.

Analysis of the generated global polynomials has shown that their degree complies with a general formula. We have discovered as well non-isomorphic graphs in the same group having the same associated global polynomials. Furthermore, the results were consistent with Lemma 2.6.1 and Theorem 2.6.2, implying a bound on the argument of the complex roots depending on the associated local polynomial, and, consequently, the absence of zeros in a region containing the $[0, 1]$ interval, proving the existence of deterministic polynomial-time approximation algorithms for these Holant problems. In addition, empirical observations on the interesting distribution of the complex roots close to the origin have been made, presenting polynomials with roots outside the unit circle, and examples of graphs that disprove Lebowitz, Pittel, Ruelle and Speer's assumption that the roots lie in the left plane have been found.

We acknowledge the limitation of the built-in root solving algorithm used and the fact that it might influence the accuracy of the roots. We have also relied on the correctness of NetworkX's function for checking for graph isomorphism. However, we do not have knowledge of other similar experiments previously carried out to refer to for comparing our solutions and the resulting roots.

5.2 Future work

The experiments described in this paper represent a small fraction of the work that could be done in this area and there is space for plenty of further research. It would be interesting to see how the distribution of the roots of graph polynomials of regular graphs varies with the increase in the number of vertices and edges or by extending the restrictions to larger matching signature functions. More computational power than our resources would be necessary for an advancement in this direction.

Furthermore, similar experiments to those presented here could be carried out for other type of graphs and signature functions to check how the patterns we could notice are affected.

Taking a completely different approach, we propose for the second part of the project a study on the roots of reliability polynomials. Following the definition in [BC92], the reliability of a graph $G = (V,E)$ is the probability that G is connected, given that edges are independently operational with probability p and fail with probability $1-p$. Summing over the probability of all subsets of connected edges whose value is stated in [BM17] gives the reliability polynomial:

$$Z_{rel}(G,p) = \sum_{S \subseteq E,(V,S)\ connected} p^{|S|}(1-p)^{|E|-|S|},$$

where p is the probability of each edge failing independently and E is the set of edges of the graph G.

The study of graph reliability is of particular interest, for example for computing the probability that a complex network system is functional given the failure probabilities of its elements, as suggested in [EYK14]. Research on reliability polynomials has provided upper bounds for the modulus of the roots [BM17] and has disproved the conjecture that the roots lie inside the unit circle [RS04]. Of particular interest for us is, however, the absence of zeros in a region containing the $[0,1]$ interval that would imply the existence of a polynomial-time approximation algorithm.

We plan to accomplish similar sub-goals to the ones followed in this paper, starting with a more in-depth literature review for a better understanding of the research on reliability polynomials and their roots, continuing eventually with the generation of new graphs and with the more challenging task of enumerating the connected subsets and finishing with the implementation of an algorithm to return the polynomials whose roots are to be analysed.

Bibliography

[BC92] Jason I Brown and Charles J Colbourn. Roots of the reliability polynomials. *SIAM Journal on Discrete Mathematics*, 5(4):571–585, 1992. Available at https://epubs.siam.org/doi/pdf/10.1137/0405047.

[BM17] Jason Brown and Lucas Mol. On the roots of all-terminal reliability polynomials. *Discrete Mathematics*, 340(6):1287–1299, 2017. Available at https://doi.org/10.1016/j.disc.2017.01.024.

[Bol01] Béla Bollobás. *Random graphs*. Cambridge University press, 2nd edition, 2001.

[CFSV01] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001. Available at https://www.researchgate.net/publication/200034365_An_Improved_Algorithm_for_Matching_Large_Graphs.

[CG19] Jin-Yi Cai and Artem Govorov. Perfect matchings, rank of connection tensors and graph homomorphisms. pages 476–495, 2019. Available at https://doi.org/10.1137/1.9781611975482.30.

[CGW16a] J. Cai, H. Guo, and T. Williams. A complete dichotomy rises from the capture of vanishing signatures. *SIAM Journal on Computing*, 45(5):1671–1728, 2016. Available at https://doi.org/10.1137/15M1049798.

[CGW16b] Jin-Yi Cai, Heng Guo, and Tyson Williams. The complexity of counting edge colorings and a dichotomy for some higher domain Holant problems. *Research in the Mathematical Sciences*, 3(1):18, 2016. Available at https://doi.org/10.1186/s40687-016-0067-8.

[CH11] Yves Crama and Peter L. Hammer. *Boolean Functions: Theory, Algorithms, and Applications*. Cambridge University Press, 1st edition, 2011.

[CLX09] Jin-Yi Cai, Pinyan Lu, and Mingji Xia. Holant problems and counting CSP. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 715–724. ACM, 2009. Available at https://dl.acm.org/citation.cfm?id=1536511.

[Dat13] B Nath Datta. Numerical methods for the root finding problem. *Northern Illinois University*, 2013. Available at http://www.math.niu.edu/~dattab/MATH435.2013/ROOT_FINDING.pdf.

[EM95] Alan Edelman and H. Murakami. Polynomial roots from companion matrix eigenvalue. *Math. Comp. 64 (1995), 763-776*, 1995. Available at https://doi.org/10.1090/S0025-5718-1995-1262279-2f.

[EYK14] Stephen Eubank, Mina Youssef, and Yasamin Khorramzadeh. Using the network reliability polynomial to characterize and design networks. *Journal of complex networks*, 2(4):356–372, 2014. Available at https://academic.oup.com/comnet/article/2/4/356/2841268.

[FLS07] Michael Freedman, László Lovász, and Alexander Schrijver. Reflection positivity, rank connectivity, and homomorphism of graphs. *Journal of the American Mathematical Society*, 20(1):37–51, 2007. Available at https://doi.org/10.1090/S0894-0347-06-00529-7.

[Gan18] Ghurumuruhan Ganesan. Existence of connected regular and nearly regular graphs. *arXiv e-prints*, page arXiv:1801.08345, January 2018. Available at https://arxiv.org/pdf/1801.08345.pdf.

[Gey11] Charles Geyer. Introduction to Markov Chain Monte Carlo. *Handbook of markov chain monte carlo*, 20116022:45, 2011. Available at https://doi.org/10.1201/b10905.

[GG14] Robert B Gardner and NK Govil. Eneström–Kakeya theorem and some of its generalizations. In *Current Topics in Pure and Computational Complex Analysis*, pages 171–199. Springer, 2014.

[GLLZ] Heng Guo, Chao Liao, Pinyan Lu, and Chihao Zhang. Zeros of Holant problems: locations and algorithms. *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2262–2278. Available at https://doi.org/10.1137/1.9781611975482.137.

[HJ07] D Hunter John. Matplotlib: a 2d graphics environment comput. *Sci. Eng*, 9:90–5, 2007. Available at https://ieeexplore.ieee.org/document/4160265.

[HKV06] J H. Kim and Van Hanh Vu. Generating random regular graphs. *Combinatorica*, 26:683–708, 12 2006. Available at https://dl.acm.org/citation.cfm?id=780576.

[HLZ16] Lingxiao Huang, Pinyan Lu, and Chihao Zhang. Canonical paths for MCMC: from art to science. pages 514–527, 2016. Available at https://arxiv.org/abs/1510.04099.

[HN08] Christopher P Hughes and Ashkan Nikeghbali. The zeros of random polynomials cluster uniformly near the unit circle. *Compositio Mathematica*, 144(3):734–746, 2008. Available at https://doi.org/10.1112/S0010437X07003302.

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using NetworkX. 2008. Available at https://www.osti.gov/biblio/960616.

[J+13] Fredrik Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*, December 2013. http://mpmath.org//.

[JOP+01] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001. http://www.scipy.org/.

[Kak12] Sôichi Kakeya. On the limits of the roots of an algebraic equation with positive coefficients. *Tohoku Mathematical Journal, First Series*, 2:140–142, 1912. Available at https://www.jstage.jst.go.jp/article/tmj1911/2/0/2_0_140/_article/-char/ja/.

[KHH12] Hendrike Klein-Hennig and Alexander K Hartmann. Bias in generation of random graphs. *Physical Review E*, 85(2):026101, 2012. Available at https://arxiv.org/abs/1107.5734.

[Kho18] Dmitry I. Khomovsky. Generalizations of the Durand-Kerner method. *arXiv e-prints*, page arXiv:1806.06280, June 2018. Available at https://arxiv.org/abs/1806.06280.

[Kur05] James F Kurose. *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India, 2005.

[Lan02] Serge Lang. *Algebra*. New York: Springer-Verlag, 3rd edition, 2002.

[LLCa] Python Software Foundation LLC. Python language reference, version 3.7. https://docs.python.org/3/.

[LLCb] Python Software Foundation LLC. What's new in Python 3.7 - Optimization. https://docs.python.org/3.7/whatsnew/3.7.html#optimizations.

[LPRS16] JL Lebowitz, Boris Pittel, D Ruelle, and ER Speer. Central limit theorems, Lee–Yang zeros, and graph-counting polynomials. *Journal of Combinatorial Theory, Series A*, 141:147–183, 2016. Available at https://doi.org/10.1016/j.jcta.2016.02.009.

[MCOD06] Richard C Murphy, Scott M Carter, Mario G Ornelas, and Shrikant Deshpande. *System and method for dynamic resource configuration using a dependency graph*. Google Patents, December 19 2006.

[Mer99] Markus Meringer. Fast generation of regular graphs and construction of cages. *Journal of Graph Theory*, 30(2):137–146, 1999. Available at https://doi.org/10.1002/(SICI)1097-0118(199902)30:2<137::AID-JGT7>3.0.CO;2-G.

[MSP+17] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic

computing in Python. *PeerJ Computer Science*, 3:e103, 2017. Available at https://peerj.com/articles/cs-103/?utm_source=TrendMD&utm_campaign=PeerJ_TrendMD_1&utm_medium=TrendMD.

[NLKB11] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th international conference on extending database technology*, pages 331–342. ACM, 2011. Available at https://dl.acm.org/citation.cfm?id=1951406.

[Oli] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. http://www.numpy.org.

[Pal10] Katarzyna Paluch. Popular b-matchings, December 2010. Available at https://arxiv.org/abs/1101.0021.

[Pan94] Victor Y Pan. Simple multivariate polynomial multiplication. *Journal of Symbolic Computation*, 18(3):183–186, 1994. Available at https://doi.org/10.1006/jsco.1994.1042.

[PR17] Viresh Patel and Guus Regts. Deterministic polynomial-time approximation algorithms for partition functions and graph polynomials. *SIAM Journal on Computing*, 46(6):1893–1919, 2017. Available at https://doi.org/10.1137/16M1101003.

[RS02] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2002.

[RS04] Gordon Royle and Alan D Sokal. The Brown–Colbourn conjecture on zeros of reliability polynomials is false. *Journal of Combinatorial Theory, Series B*, 91(2):345–360, 2004. Available at https://doi.org/10.1016/j.jctb.2004.03.008.

[SV95] Larry A Shepp and Robert J Vanderbei. The complex zeros of random polynomials. *Transactions of the American Mathematical Society*, 347(11):4365, 1995. Available at https://repository.upenn.edu/cgi/viewcontent.cgi?article=1414&context=statistics_papers.

[SW99] Angelika Steger and Nicholas C Wormald. Generating random regular graphs quickly. *Combinatorics, Probability and Computing*, 8(4):377–396, 1999. Available at https://pdfs.semanticscholar.org/a11c/daa94e777b0d9752a326224f742aa3f71c3b.pdf.

[Val79a] Leslie G Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979. Available at https://doi.org/10.1016/0304-3975(79)90044-6.

[Val79b] Leslie G Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979. Available at https://doi.org/10.1137/0208032.

[Ver96] Anatolii Moiseevich Vershik. Statistical mechanics of combinatorial partitions, and their limit shapes. *Functional Analysis and Its Applications*, 30(2):90–105, 1996. Available at `https://link.springer.com/article/10.1007%2FBF0250944a9?LI=true`.

[Wor99] Nicholas C Wormald. Models of random regular graphs. *London Mathematical Society Lecture Note Series*, pages 239–298, 1999. Available at `http://users.monash.edu.au/~nwormald/papers/regsurvey.pdf`.

Appendices

Appendix A

SymPy vs NumPy

Degree	SymPy	NumPy
2	0.65	0.47
3	0.59	0.22
4	0.74	0.16
5	0.99	0.16
6	1.07	0.17
7	1.10	0.16
8	1.58	0.25
9	1.97	0.23
10	2.39	0.27

Table A.1: Time in ms to compute roots of polynomials with SymPy and NumPy

Degree	Polynomial
2	$4x^2+3x+2$
3	$4x^3-3x^2-25x-6$
4	$7x^4+8x^3-23x^2+29$
5	$x^5-4x^4-7x^3+14x^2-44x+120$
6	$x^6+6x^5+15x^4+20x^3+15x^2+6x+1$
7	$x^7+7x^6+21x^5+35x^4+35x^3+21x^2+7x+1$
8	$x^8+8x^7+28x^6+56x^5+70x^4+56x^3+28x^2+8x+1$
9	$8x^9+138x^8+468x^7+756x^6+744x^5+489x^4+220x^3+66x^2+12x+1$
10	$70x^{10}+567x^9+1638x^8+2606x^7+2688x^6+$ $+1932x^5+994x^4+364x^3+91x^2+14x+1$

Table A.2: Polynomials used for testing root solving libraries

Appendix B

Number of non-isomorphic connected d-regular graphs

The number of regular graphs as stated in Table B.1 have been computed by Meringer and presented in [Mer99].

Vertices	Degree 2	Degree 3	Degree 4	Degree 5	Degree 6	Degree 7
3	1	0	0	0	0	0
4	1	1	0	0	0	0
5	1	0	1	0	0	0
6	1	2	1	1	0	0
7	1	0	2	0	1	0
8	1	5	6	3	1	1
9	1	0	16	0	4	0
10	1	19	59	60	21	5
11	1	0	256	0	266	0
12	1	85	1544	7848	7849	1547
13	1	0	10778	0	367860	0
14	1	509	88168	3459383	21609300	21609301
15	1	0	805491	0	1470293675	0
16	1	4060	8037418	2585136675	113314233808	733351105934

Table B.1: Number of non-isomorphic connected d-regular graphs

Appendix C

Local Polynomials

d	b	Polynomial
3	2	$1+3x+3x^2$
4	2	$1+4x+6x^2$
4	3	$1+4x+6x^2+4x^3$
5	2	$1+5x+10x^2$
5	3	$1+5x+10x^2+10x^3$
5	4	$1+5x+10x^2+10x^3+5x^4$
6	2	$1+6x+15x^2$
6	3	$1+6x+15x^2+20x^3$
6	4	$1+6x+15x^2+20x^3+15x^4$
6	5	$1+6x+15x^2+20x^3+15x^4+6x^5$
7	2	$1+7x+21x^2$
7	3	$1+7x+21x^2+35x^3$
7	4	$1+7x+21x^2+35x^3+35x^4$
7	5	$1+7x+21x^2+35x^3+35x^4+21x^5$
7	6	$1+7x+21x^2+35x^3+35x^4+21x^5+7x^6$

Table C.1: Local polynomials

d	b	Polynomial
8	2	$1+8x+28x^2$
8	3	$1+8x+28x^2+56x^3$
8	4	$1+8x+28x^2+56x^3+70x^4$
8	5	$1+8x+28x^2+56x^3+70x^4+56x^5$
8	6	$1+8x+28x^2+56x^3+70x^4+56x^5+28x^6$
8	7	$1+8x+28x^2+56x^3+70x^4+56x^5+28x^6+8x^7$
9	2	$1+9x+36x^2$
9	3	$1+9x+36x^2+84x^3$
9	4	$1+9x+36x^2+84x^3+126x^4$
9	5	$1+9x+36x^2+84x^3+126x^4+126x^5$
9	6	$1+9x+36x^2+84x^3+126x^4+126x^5+84x^6$
9	7	$1+9x+36x^2+84x^3+126x^4+126x^5+84x^6+36x^7$
9	8	$1+9x+36x^2+84x^3+126x^4+126x^5+84x^6+36x^7+9x^8$
10	2	$1+10x+45x^2$
10	3	$1+10x+45x^2+120x^3$
10	4	$1+10x+45x^2+120x^3+210x^4$
10	5	$1+10x+45x^2+120x^3+210x^4+252x^5$
11	2	$1+11x+55x^2$
11	3	$1+11x+55x^2+165x^3$
11	4	$1+11x+55x^2+165x^3+330x^4$
11	5	$1+11x+55x^2+165x^3+330x^4+462x^5$

Table C.2: Local polynomials

Appendix D

The graphs used in Example 4.3.1

Adjacency list:
0: 1, 2, 3, 4, 7, 8, 9
1: 0, 2, 3, 4, 5, 6, 7
2: 0, 1, 3, 5, 6, 8, 9
3: 0, 1, 2, 5, 7, 8, 9
4: 0, 1, 5, 6, 7, 8, 9
5: 1, 2, 3, 4, 6, 8, 9
6: 1, 2, 4, 5, 7, 8, 9
7: 0, 1, 3, 4, 6, 8, 9
8: 0, 2, 3, 4, 5, 6, 7
9: 0, 2, 3, 4, 5, 6, 7

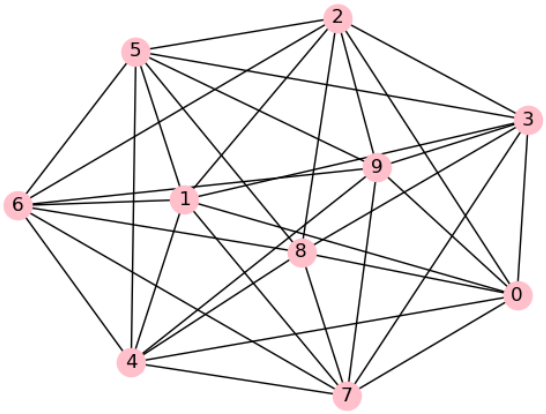

Figure D.1: (10,7)-graph

Adjacency list:
0: 1, 2, 4, 6, 9, 10, 11
1: 0, 3, 5, 7, 8, 9, 11
2: 0, 5, 6, 7, 8, 9, 10
3: 1, 4, 5, 8, 9, 10, 11
4: 0, 3, 5, 6, 8, 9, 10
5: 1, 2, 3, 4, 6, 8, 10
6: 0, 2, 4, 5, 7, 10, 11
7: 1, 2, 6, 8, 9, 10, 11
8: 1, 2, 3, 4, 5, 7, 11
9: 0, 1, 2, 3, 4, 7, 11
10: 0, 2, 3, 4, 5, 6, 7
11: 0, 1, 3, 6, 7, 8, 9

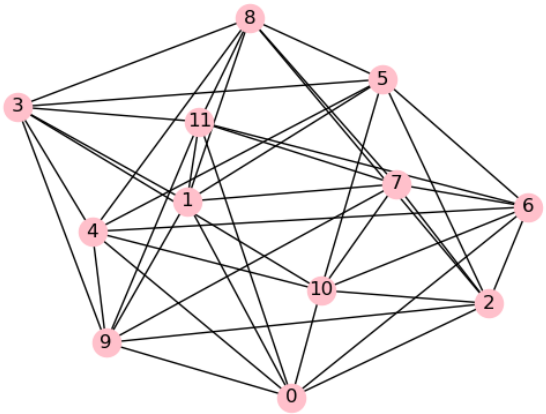

Figure D.2: (12,7)-graph

Appendix E

Degrees of global polynomials

In the table below, a "$-$" (in the cases $b > d$) signifies that the configuration has not been considered.

n	d	$b=2$	$b=3$	$b=4$	$b=5$	$b=6$	$b=7$	$b=8$
4	3	4	6	-	-	-	-	-
5	4	5	7	10	-	-	-	-
6	3	6	9	-	-	-	-	-
6	4	6	9	12	-	-	-	-
6	5	6	9	12	15	-	-	-
7	4	7	10	14	-	-	-	-
7	6	7	10	14	17	21	-	-
8	3	8	12	-	-	-	-	-
8	4	8	12	16	-	-	-	-
8	5	8	12	16	20	-	-	-
8	6	8	12	16	20	24	-	-
8	7	8	12	16	20	24	28	-
9	4	9	13	18	-	-	-	-
9	6	9	13	18	22	-	-	-
9	8	9	13	18	22	26	30	-
10	3	10	15	-	-	-	-	-
10	4	10	15	20	-	-	-	-
10	5	10	15	20	25	-	-	-
10	6	10	15	20	25	30	-	-
10	7	10	15	20	25	30	35	-
10	8	10	15	20	25	30	35	-
10	9	10	15	20	25	30	35	40

Table E.1: Degrees of global polynomials for the generated graphs

n	d	$b=2$	$b=3$	$b=4$	$b=5$	$b=6$	$b=7$	$b=8$
11	4	11	16	22	-	-	-	-
11	6	11	16	22	27	33	-	-
11	8	11	16	22	27	33	38	44
11	10	11	16	22	27	not generated	not generated	not generated
12	3	12	18	-	-	-	-	-
12	4	12	18	24	-	-	-	-
12	5	12	18	24	30	-	-	-
12	6	12	18	24	30	36	-	-
12	7	12	18	24	30	not generated	42	-
12	8	12	18	24	30	not generated	not generated	48
12	9	12	18	24	30	not generated	not generated	not generated
12	10	12	18	24	30	not generated	not generated	not generated
12	11	12	18	24	not generated	not generated	not generated	not generated
13	4	13	19	26	-	-	-	-
13	6	13	19	26	32	39	-	-
14	2	14	-	-	-	-	-	-
14	3	14	21	-	-	-	-	-
14	4	14	21	28	-	-	-	-
14	5	14	21	28	35	-	-	-
14	6	14	21	28	not generated	42	-	-
15	4	15	22	30	-	-	-	-
16	3	16	24	-	-	-	-	-
16	4	16	24	32	-	-	-	-
16	5	16	24	32	40	-	-	-
17	4	17	25	34	-	-	-	-
18	3	18	27	-	-	-	-	-
18	4	18	27	36	-	-	-	-
18	5	18	27	36	45	-	-	-
19	4	19	28	38	-	-	-	-
20	3	20	30	-	-	-	-	-
20	4	20	30	40	-	-	-	-
20	5	20	30	40	50	-	-	-

Table E.2: Degrees of the global polynomials for the generated graphs

Appendix F

Pairs of graphs with the same global polynomial

- $n = 10$, $d = 4$, $b = 2$ and $b = 3$:

Adjacency list:
0: 1, 3, 4, 7
1: 0, 2, 4, 9
2: 1, 5, 6, 7
3: 0, 4, 8, 9
4: 0, 1, 3, 5
5: 2, 4, 6, 8
6: 2, 5, 7, 9
7: 0, 2, 6, 8
8: 3, 5, 7, 9
9: 1, 3, 6, 8

Adjacency list:
0: 1, 2, 4, 7
1: 0, 3, 5, 6
2: 0, 5, 7, 9
3: 1, 4, 5, 8
4: 0, 3, 6, 9
5: 1, 2, 3, 7
6: 1, 4, 8, 9
7: 0, 2, 5, 8
8: 3, 6, 7, 9
9: 2, 4, 6, 8

- $n = 10$, $d = 4$, $b = 3$:

Adjacency list:
0: 3, 5, 7, 8
1: 2, 3, 5, 8
2: 1, 3, 4, 8
3: 0, 1, 2, 7
4: 2, 5, 6, 9
5: 0, 1, 4, 9
6: 4, 7, 8, 9
7: 0, 3, 6, 9
8: 0, 1, 2, 6
9: 4, 5, 6, 7

Adjacency list:
0: 2, 5, 7, 8
1: 2, 4, 6, 9
2: 0, 1, 3, 4
3: 2, 4, 6, 8
4: 1, 2, 3, 7
5: 0, 6, 7, 9
6: 1, 3, 5, 9
7: 0, 4, 5, 8
8: 0, 3, 7, 9
9: 1, 5, 6, 8

- $n = 12$, $d = 3$, $b = 2$:

Adjacency list:
0: 1, 6, 8
1: 0, 7, 10
2: 3, 4, 8
3: 2, 9, 11
4: 2, 8, 9
5: 7, 10, 11
6: 0, 10, 11
7: 1, 5, 9
8: 0, 2, 4
9: 3, 4, 7
10: 1, 5, 6
11: 3, 5, 6

Adjacency list:
0: 6, 9, 10
1: 4, 8, 11
2: 7, 8, 11
3: 5, 7, 9
4: 1, 9, 10
5: 3, 7, 10
6: 0, 8, 11
7: 2, 3, 5
8: 1, 2, 6
9: 0, 3, 4
10: 0, 4, 5
11: 1, 2, 6

Appendix G

Example of graph and associated global polynomial

We provide the adjacency list of a $(12,5)$-graph and its global polynomial from a 4-matching, whose coefficients are plotted in Figure 4.2.

Adjacency list:
0: 1, 2, 4, 8, 11
1: 0, 3, 7, 8, 11
2: 0, 4, 5, 8, 10
3: 1, 4, 5, 9, 10
4: 0, 2, 3, 6, 10
5: 2, 3, 6, 7, 9
6: 4, 5, 7, 9, 11
7: 1, 5, 6, 9, 10
8: 0, 1, 2, 10, 11
9: 3, 5, 6, 7, 11
10: 2, 3, 4, 7 ,8
11: 0, 1, 6, 8, 9

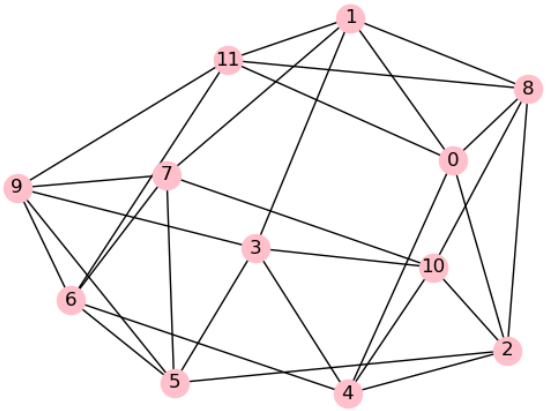

Figure G.1: (12,5)-graph

Coefficients, from most significant to least significant:

[131, 10452, 163867, 1186773, 5219518, 15939850, 36516551, 65886906,
96708252, 118054362, 121688192, 107001195, 80771550, 52509120,
29408121, 14155380, 5825325, 2032200, 593475, 142494, 27405, 4060, 435, 30, 1]

Appendix H

Average distance from the roots to the limit angle

In the table below, a "−" signifies that the configuration has not been considered. The values have been rounded to the nearest two decimals.

n	d	$b=2$	$b=3$	$b=4$	$b=5$	$b=6$	$b=7$	$b=8$
4	3	17.61°	-	-	-	-	-	-
5	4	22.75°	27.16°	-	-	-	-	-
6	3	13.96°	-	-	-	-	-	-
6	4	21.26°	25.65°	-	-	-	-	-
6	5	24.79°	35.19°	33.25°	-	-	-	-
7	4	20.10°	24.40°	-	-	-	-	-
7	6	25.82°	39.11°	43.22°	37.49°	-	-	-
8	3	12.50°	-	-	-	-	-	-
8	4	19.22°	23.45°	-	-	-	-	-
8	5	22.81°	33.06°	31.13°	-	-	-	-
8	6	24.96°	38.18°	42.30°	36.58°	-	-	-
8	7	26.39°	41.37°	48.38°	48.82°	40.62°	-	-
9	4	18.68°	22.82°	-	-	-	-	-
9	6	24.24°	37.39°	41.50°	35.80°	-	-	-
9	8	26.73°	42.80°	51.51°	54.88°	52.95°	43.03°	-
10	3	11.74°	-	-	-	-	-	-
10	4	18.20°	22.31°	-	-	-	-	-
10	5	21.58°	31.67°	29.77°	-	-	-	-
10	6	23.71°	36.78°	40.89°	35.21°	-	-	-
10	7	25.13°	39.97°	46.99°	47.44°	39.26°	-	-
10	8	26.16°	42.17°	50.89°	54.27°	52.34°	42.43°	-
10	9	26.94°	43.77°	53.59°	58.65°	59.67°	56.11°	44.93°

Table H.1: Average distance from the roots to the limit angle

n	d	$b=2$	$b=3$	$b=4$	$b=5$	$b=6$	$b=7$	$b=8$
11	4	17.75°	21.82°	-	-	-	-	-
11	6	23.24°	36.26°	40.36°	34.70°	-	-	-
11	8	25.69°	41.63°	50.36°	53.74°	51.82°	41.92°	-
11	10	27.06°	44.45°	55.05°	61.21°	-	-	-
12	3	10.79°	-	-	-	-	-	-
12	4	17.36°	21.38°	-	-	-	-	-
12	5	20.67°	30.63°	28.75°	-	-	-	-
12	6	22.84°	35.79°	39.90°	34.26°	-	-	-
12	7	24.25°	38.97°	46.00°	46.46°	-	-	-
12	8	25.29°	41.19°	49.92°	53.31°	-	-	-
12	9	26.05°	42.77°	52.61°	57.69°	-	-	-
12	10	26.65°	43.99°	54.60°	60.77°	-	-	-
12	11	27.14°	44.95°	56.13°	-	-	-	-
13	4	16.82°	20.79°	-	-	-	-	-
13	6	22.60°	35.52°	39.63°	33.99°	-	-	-
14	3	10.15°	-	-	-	-	-	-
14	4	16.75°	20.66°	-	-	-	-	-
14	5	19.95°	29.81°	27.97°	-	-	-	-
14	6	22.15°	35.00°	39.11°	-	-	-	-
15	4	16.31°	20.20°	-	-	-	-	-
16	3	9.71°	-	-	-	-	-	-
16	4	15.93°	19.80°	-	-	-	-	-
16	5	19.52°	29.30°	27.46°	-	-	-	-
17	4	15.91°	19.76°	-	-	-	-	-
18	3	9.23°	-	-	-	-	-	-
18	4	15.48°	19.31°	-	-	-	-	-
18	5	19.16°	28.89°	27.04°	-	-	-	-
19	4	15.28°	19.08°	-	-	-	-	-
20	4	15.70°	19.50°	-	-	-	-	-
20	3	9.93°	-	-	-	-	-	-
20	5	18.96°	28.65°	26.82°	-	-	-	-

Table H.2: Average distance from the roots to the limit angle