Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Spanning trees in graphs

Bachelor Thesis

2022
Terézia Strišovská

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Spanning trees in graphs

Bachelor Thesis

Study Programme:	Applied Informatics
Field of Study:	Applied Informatics
Department:	Department of Applied informatics
Supervisor:	doc. RNDr. Tatiana Jajcayová, PhD.
Consultant:	Mgr. Dominika Mihálová

Bratislava, 2022
Terézia Strišovská

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Terézia Strišovská
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Spanning trees in graphs
Kostry v grafoch

Anotácia: Cieľom práce je navrhnť a zbehnúť experimenty a určiť ohraničenia pre počet kostier v rôznych triedach grafov. Napríklad výsledok Nogu Alona tvrdí, že pre k-regulárne jednoduché grafy s k>0, s n vrcholmi existuje aspoň 2^(n\2) kostier.

Cieľ: Cieľom práce je navrhnť a zbehnúť experimenty a určiť ohraničenia pre počet kostier v rôznych triedach grafov.

Vedúci: doc. RNDr. Tatiana Jajcayová, PhD.
Konzultant: Mgr. Dominika Mihálová
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 01.06.2022

Dátum schválenia: 19.09.2022 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Terézia Strišovská
Study programme:	Applied Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title:	Spanning trees in graphs
Annotation:	The aim of the theses is to run experiments and give bounds for number of spanning trees in variety of classes of graphs. For isntance a result of Noga Alon claims that for k-regular simple graphs with k>0, with n vertices there are at least 2^(n\2) spanning trees.
Aim:	The aim of the theses is to run experiments and give bounds for number of spanning trees in different classes of graphs.

Supervisor:	doc. RNDr. Tatiana Jajcayová, PhD.
Consultant:	Mgr. Dominika Mihálová
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	01.06.2022

Approved: 19.09.2022 doc. RNDr. Damas Gruska, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments: Tu môžete poďakovať školiteľovi, prípadne ďalším osobám, ktoré vám s prácou nejako pomohli, poradili, poskytli dáta a podobne.

Abstrakt

Slovenský abstrakt v rozsahu 100-500 slov, jeden odstavec. Abstrakt stručne sumarizuje výsledky práce. Mal by byť pochopiteľný pre bežného informatika. Nemal by teda využívať skratky, termíny alebo označenie zavedené v práci, okrem tých, ktoré sú všeobecne známe.

Kľúčové slová: jedno, druhé, tretie (prípadne štvrté, piate)

Abstract

The aim of this thesis is to study the number of spanning trees in different classes of graphs. We designed a set of experiments centered mainly around regular graphs and examined graphs with the least and most spanning trees for a given category of graphs. We were using GenReg, a program that enabled us to acquire even large complete sets of unlabeled k-regular graphs on n vertices which were then processed and evaluated with regard to their number of spanning trees. For each experiment, command line tools were implemented, allowing the user to specify properties of the set of graphs they wish to work with. After having gathered enough data, we proposed a hypothesis about the structure and the number of spanning trees in 3-regular graphs on n vertices with minimum number of spanning trees for given n. We also suggested a trait that seems to be shared between 3-regular graphs with maximum number of spanning trees for a given number of vertices.

Keywords: graph, spanning tree, regular graph, GenReg

Contents

Introduction **1**

1 Preliminaries **3**

1.1 Basic terminology and definitions . . . 3

1.1.1 Simple graph . . . 3

1.1.2 Regular graph . . . 4

1.1.3 Paths and cycles . . . 4

1.1.4 Connected graph . . . 5

1.1.5 Subgraph and spanning subgraph . . . 5

1.1.6 Tree . . . 6

1.1.7 Spanning tree . . . 6

1.1.8 Graph isomorphism . . . 7

1.2 Graph representation . . . 8

1.3 Tree isomorphism . . . 8

1.4 Spanning tree enumeration . . . 9

1.5 Matrix Tree Theorem . . . 9

1.6 Graph classes and their bounds for number of spanning trees . . . 10

1.6.1 Trees . . . 10

1.6.2 Cycles . . . 11

1.6.3 Complete graphs . . . 11

1.6.4 Regular graphs . . . 12

2 Implementation **15**

2.1 Methods . . . 15

2.2 Genreg . . . 16

2.3 Programming language . . . 17

2.4 Bash scripts . . . 18

2.5 Algorithms . . . 18

2.5.1 AHU algorithhm . . . 18

2.5.2 Spanning tree enumeration . . . 19

2.5.3 Spanning tree counting . . . 20

2.6 Classes and their functionality 20
2.6.1 SpanningTreeCounter 20
2.6.2 ProcRegular 20
2.6.3 ProcBiregular 21
2.6.4 ColRegular 21
2.6.5 GeneralFunctionsForGraphs 22
2.6.6 ColBiregular 22
2.7 Design of graph generation and processing 22
2.7.1 Serial processing 23
2.7.2 Parallel processing 23
2.7.3 Unsuccessful prototype 24
2.8 Tools for experiments 24
2.8.1 Minimum/maximum numbers of spanning trees 25
2.8.2 Unlabeled spanning trees and other funcionality 26

3 Result analysis **27**
3.1 Regular graphs with minimum number of spanning trees 27
3.2 3-regular graphs 27
3.3 4-regular graphs 30
3.4 Regular graphs with maximum number of spanning trees 31
3.5 Biregular graphs with minimum number of spanning trees 32
3.6 Isomorphism classes 33
3.7 Software efficiency 34

Conclusion **37**

Appendix A **41**
3.8 3-regular graphs with maximum number of spanning trees 41
3.9 4-regular graphs with maximum number of spanning trees 44
3.10 3-regular graphs with maximum number of spanning trees 45

List of Figures

1.1 3-regular graphs on 8 vertices 4
1.2 Paths in a graph 5
1.3 Examples of subgraphs and spanning subgraph 5
1.4 All spanning trees of graph G 7
1.5 Isomorphic graphs 8
1.6 Representants of each isomorphism class of K_5's spanning trees 12
1.7 All connected unlabeled 0, 1 and 2-regular graphs on up to 5 vertices . 12
1.8 Comparison of Alon's and McKays upper bounds for regular graphs . . 14

2.1 Example of output of genreg 17
2.2 Format of results 21
2.3 Scheme for parallel generation and graph processing 24
2.4 Menu of cli interface 26
2.5 Example of output for comparison of spanning trees in two graphs . . . 26

3.1 Building partitions of 3-regular graphs with minimum number of spanning trees 28
3.2 3-regular graphs on 10 and 16 vertices with the lowest number of spanning trees 28
3.3 Scheme of 3-regular graphs with lowest number of spanning trees on $n = 10 + 4i$ vertices 29
3.4 Scheme of 3-regular graphs with lowest number of spanning trees on $n = 16 + 4i$ vertices 29
3.5 Comparison of Alon's and our lower bounds for number of spanning trees in 3-regular graphs 30
3.6 Building partitions of 4-regular graphs with minimum number of spanning trees 31
3.7 New building partitions for biregular graphs 32
3.8 Biregular graph of order $n + 1$ with least spanning trees for $n = 14$, $k = 3$ and $k_1 = 10$ 33

3.9 Biregular graph of order $n + 1$ with least spanning trees for $n = 16$, $k = 3$ and $k_1 = \frac{n}{2}$ 33
3.10 3-regular graphs with minimal number of spanning trees on from 4 to 28 vertices 43
3.11 3-regular graphs with minimal number of spanning trees on from 4 to 28 vertices 45

List of Tables

3.1 Maximum and minimum numbers of spanning trees in 4-regular graphs 31
3.2 Comparison of number of isomorphism classes 34
3.3 Computation times for serial generation of 3-regular graphs 35
3.4 Computation times for parallel generation of 3-regular graphs 35

Introduction

In the first chapter, we will explain terms from graph theory that are necessary for the understanding of our thesis. We will cover also the problem of graph isomorphism, methods for generating and counting spanning trees in graphs. Finally, we will discuss already known bounds for number of spanning trees in various graph families and existing works concerning this topic.

The second chapter will contain information about used technologies, algorithms and the experiments we want to conduct. We will also describe the components of our software and how they are integrated. We will go through the functionality of the software and explain which experiments it will suit for.

In the third chapter, we will introduce the results of our experiments. We will discuss especially regular graphs with highest and lowest number of spanning trees, what can we say about their structure and how could that help us estimate the bounds for number of spanning trees. It will also include evaluation of time performance of our software.

Chapter 1

Preliminaries

This chapter will be devoted to explaining terms and notations from graph theory, introducing concepts, algorithms and previous research in the field of the topic of our thesis.

1.1 Basic terminology and definitions

In this section, we will introduce basic definitions from graph theory that are relevant to our work. Each subsection is dedicated to one or two related terms, providing their definition, further explanation and in some cases illustrated examples.

1.1.1 Simple graph

Graph is a structure, that is present around us in many forms, even though we may not realize it's a graph. Transportation network, family trees and electricity network are just a few examples. We will look at this term from a formal side, defining one specific type of graphs - simple graphs.

Definition. A simple graph is an ordered pair of sets $G = (V, E)$, where V is a non-empty set of vertices (or nodes) of G, and E, the set of edges of G, is a set of two-element pairs(2-combinations) of vertices. Thus, each edge of G can be expressed as $\{u, v\}$, where u and v are distinct vertices, i.e., $u, v \in V, u \neq v$. The vertices u and v determining an edge $\{u, v\}$ are called edpoints of the edge. The edge $\{u, v\}$ is said to join u and v, and the edge is said to be incident to either of its endpoints. Any two vertices in G that are joined bz an edge are said to be adjacent, and are called neighbors. A vertex with no neighbors is called isolated. [3, p. 497]

In other words, simple graph is a graph without loops(edges with equal endpoints) and multiple edges(edges with the same pair of endpoints).

In our work, we will consider only undirected simple graphs, which means that edges are unordered pairs of vertices and thus each edge $\{u, v\}$ can be traversed in both directions - from u to v and from v to u.

1.1.2 Regular graph

Definition. If v is a vertex of a graph G, then the degree of v, denoted $deg(v)$ (or $deg_G(V)$), if we wish to emphasize the dependence on G), is the number of edges incident to v, with any self-loops counted twice. A simple graph in which all vertices have the same degree k is called a regular graph or, more precisely, a k-regular graph. [3, p. 499]

This means that each vertex in a k-regular graph has exactly k neighbors. k may range from 0 to $|V(G)| - 1$. For $k = 0$, the edge set is empty, 1-regular graph consists of disconnected edges and 2-regular graphs contains one cycle or more disjoint cycles. For $k \geq 2$, any k-regular graph contains one or more cycle. Every simple k-regular graph on n vertices has exactly $\frac{n \cdot k}{2}$ edges. Since the sum of degrees of vertices in a simple undirected graphs must be an even number(every edge is counted twice - once in both possible directions), k-regular graphs where k is odd exist only for even number of vertices.

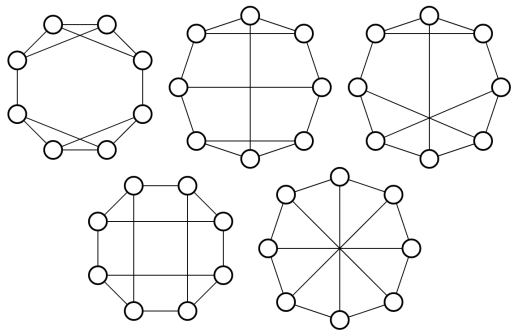


Figure 1.1: 3-regular graphs on 8 vertices

1.1.3 Paths and cycles

Definition. Suppose that $G = (V, E)$ is a graph, and $v, w \in V$ are a pair of vertices. A path in G from v to w is an alternating sequence of vertices and edges: $P =< v_0, e_1, v_1, e_2, v_2, ..., v_{k-1}, e_k, v_k >$, such that the endpoints of edge e_i are the vertices $\{v_{i-1}, v_i\}$, for $1 \leq i \leq k$, $v_0 = v$, and $v_k = w$. We say that path P passes through the vertices $v_0, v_1, v_2, ..., v_{k-1}, v_k$, and traverses the edges $e_1, e_2, ..., e_k$, and that the path has length k, since it traverses k edges. [3, p. 540]

There may exist several different paths from v to w, as we can see in figure 1.2, where there are 4 different paths from 2 to 4 and those are: $2, \{2, 0\}, 0, \{0, 1\}, 1, \{1, 4\}, 4$; $2, \{2, 0\}, 0, \{0, 1\}, 1, \{1, 3\}, 3, \{3, 4\}, 4$; $2, \{2, 3\}, 3, \{3, 4\}, 4$ and $2, \{2, 3\}, 3, \{3, 1\}, 1, \{1, 4\}, 4$.

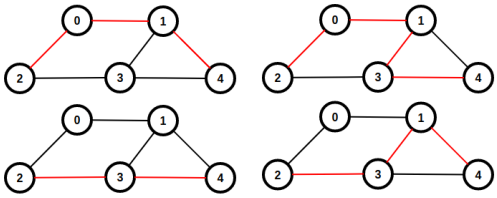


Figure 1.2: Paths in a graph

Definition. A cycle of a graph G, also called a circuit if the first vertex is not specified, is a subset of the edge set of G that forms a path such that the first node of the path corresponds to the last. A graph containing no cycles of any length is known as an acyclic graph. [9]

If we see a cycle of length n as a path that begins and ends in the same vertex and no other vertices are repeated, it is not important in which vertex it starts, since it could start in any of its n vertices and it would still represent the same cycle.

A graph where each vertex is of degree at least 2, this graph must contain a cycle.

1.1.4 Connected graph

Definition. A graph is connected if it has a u, v-path for each pair u, $v \in V(G)$. [1, p. 5]

This means that each vertex of G is reachable from any of its vertices. The number of components(maximal connected subgraphs) in connected graph with $|V(G)| \geq 1$ is exactly 1, and this component contains all the vertices from its vertex set.

1.1.5 Subgraph and spanning subgraph

Definition. A subgraph of a graph G is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$; we write $H \subseteq G$ and say that "G contains H". [1, p. 3]

Definition. A spanning subgraph of G is a subgraph with vertex set $V(G)$. [1, p. 51]

Basically, we can create a subgraph of G by deleting any of its vertices and their incident edges and any of remaining edges. However, in case of spanning subgraph, we may delete only edges, as vertex set must be preserved.

We can see this difference in figure 1.3. Both G_2 and G_3 are subgraphs of G_1, but only G_3 is its spanning subgraphs, since vertex set of G_2 is $V(G_1) \setminus \{4\}$.

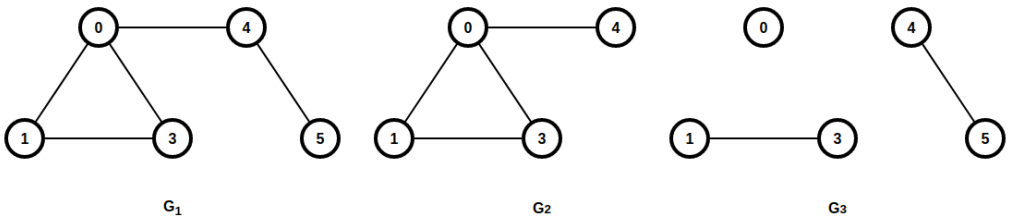


Figure 1.3: Examples of subgraphs and spanning subgraph

1.1.6 Tree

There exist several characterizations of trees and each of them is equivalent. This means that in case we want to prove that a graph is a tree, we can choose any of the characterizations and verify that the graph satisfies it.

The following are examples of characteristics of a tree G on n vertices.

a) G is connected and has no cycles

b) G is connected and has $n - 1$ edges

c) G has $n - 1$ edges and no cycles

d) for $u, v \in V(G)$, G has exactly one u, v-path [1, p. 52]

Since there is only one path between each pair of vertices, if we delete any edge from $E(G)$, the graph becomes disconnected. This means that every edge of a tree is a bridge. [3, p. 573]

When speaking of trees, a vertex of degree 1 is called a leaf, the rest of the vertices are internal vertices. [3, p. 572]

Another important properties of trees are that $|E(G)| = |V(G)| - 1$ and that by adding a new edge to G between two of it vertices which weren't adjacent originally, we create a new graph with exactly one cycle.

1.1.7 Spanning tree

Now we are getting to the core topic of our thesis, spanning trees, which are special cases of trees.

Definition. A spanning tree is a spanning subgraph that is a tree. [1, p. 51]

So when we have a graph G, any of its spanning trees T is a graph such that $V(T) = V(G)$ and T is a maximum possible tree in G, which means that adding an edge to $E(T)$ renders T cyclic.

From the earlier definitions, we can see that a disconnected graph can't have any spanning tree and therefore we will focus on connected graphs in our work.

In figure 1.4, we can see a graph G and all of its spanning trees.

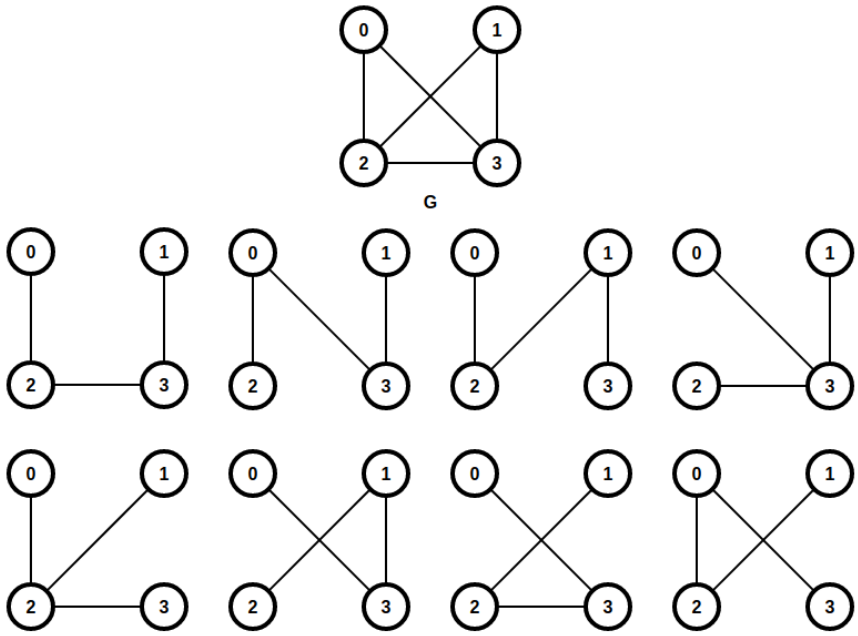


Figure 1.4: All spanning trees of graph G

Also spanning trees have many applications, for example in network, where we want to cover and connect all nodes, but also to minimize the cost of connections between those nodes. A special case of spanning tree is minimum spanning tree, in weighted graphs it's a spanning trees with minimum possible weight of the contained edges. Minimum spanning trees can be used in approximating solution of complex mathematical problems, for example the Traveling Salesman Problem.

1.1.8 Graph isomorphism

Definition. An isomorphism from G to H is a bijection $f : V(G) \to V(H)$ such that $uv \in E(G)$ if and only if $f(u)f(v) \in E(H)$. As a result, if two graphs are isomorphic, they must have the same degree sequence. However, the same degree sequence doesn't implies that two graphs have to be isomorphic. We say "G is isomorphic to H", written $G \cong H$, if there is an isomorphism from G to H. [1, p. 7]

Basically, two graphs, G and H, are isomorphic if they have the same structure, they vertex and edge sets might be different. Isomorphism is then a mapping from vertex set of V to vertex set of H, which renames the vertices of V, creating G' such that it has the same vertex and edge set as H.

Definition. An isomorphism from a graph G to itself is called an automorphism of G. [10, p. 4]

Figure 1.5 shows two isomorphic graphs G_1 and G_2, where one of the possible isomorphisms between them is $\{(0,6),(1,8),(2,7),(3,5),(4,9)\}$.

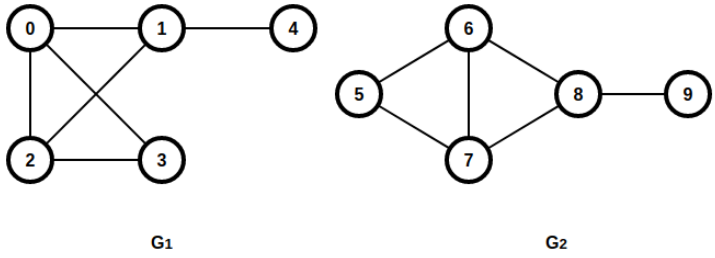


Figure 1.5: Isomorphic graphs

1.2 Graph representation

There exist several ways to represent a graph, and each way may be more suitable for a different situation, depending on the operations to be performed on graphs, the information about the graphs we want to preserve or memory criteria. We will list a few methods, some of which are also used in our work. Suppose we have a graph G with vertex set $V = \{1, 2, 3...n\}$ and edge set E.

First is adjacency matrix, which is a $|V| \times |V|$ matrix where element in i-th row and j-th column holds information about whether vertex i is adjacent to vertex j. This means that in case of undirected graphs, where $ij \in E \rightarrow ji \in E$, the matrix is diagonally symmetric. When working with unweighted graph, the value of the element would be set to 1 if $ij \in E$, otherwise to 0. For weighted graphs, it represents the weight of edge from i to j.

There is also incidence matrix, it is of size $|E| \times |V|$, each column represents one edge in the graph and rows are for vertices. It's entries are from the set $\{0, 1\}$, defined in such manner that $(v, e) = 1$ if vertex v is incident upon edge e, otherwise 0. [15]

Edge list is an array with $|E|$ pairs, each for one edge from E. This method is memory efficient in case of sparse graphs - graphs with only a few edges, however, we may need to store an extra record for the number of vertices in the graph, since there might be isolated vertices and edge sets holds no information about those.

Another representation is adjacency list, which is a collection of lists, one for each vertex of V. List for a given vertex v contains all vertices adjacent to v.

1.3 Tree isomorphism

Generally speaking, the task of determining whether two graphs are isomorphic is non-trivial. There are some cases, when we can exclude the existence of isomorphism easily, for example when the two graphs are not on the same number of vertices or when they have different degree sequence. The problem doesn't have a broadly applicable

known solution with polynomial time, neither is it known to be a NP-complete problem, therefore it may be of intermediate complexity. [11, p. 3304] For some classes of graphs, however, there exist solutions with polynomial complexity.

Algorithms for tree isomorphism work with rooted trees. Although the graphs in our thesis are undirected, simple trees can by rooted by finding central vertex such that the longest path to leaf is minimum over all vertices in the graph.

We will be using AHU algorithm, described more in detail in chapter 2, which solves the problem in linear time proportional to the number of vertices in the trees. The definition of tree isomorphism it uses if the following:

Definition. Two trees are said to be isomorphic if we can map one tree into the other by permuting the order of the sons of vertices [5, p. 84]

Each vertex is then assigned a tuple representing the structure of its subtree.

1.4 Spanning tree enumeration

Finding an arbitrary spanning tree of a graph, or even minimum spanning tree in edge-weighted graphs, is a well known problem with several possible solutions that work in polynomial time. The main factor affecting computation time is the number of edges and vertices in a graph. There are, however, cases when we would like to generate all spanning trees for a given graph. The number of spanning trees in a graph might be exponential to its size, so the time taken for generating all spanning trees is no longer polynomial.

There are a few different approaches to spanning tree enumeration, varying also in efficiency. [13] One of them would be generating combinations of graph's edges of size $n - 1$, candidates for spanning trees, and testing them for acyclicity. Another method starts with initial spanning tree, constructed for example by breadth or depth first search. We can then replace one edge of the current spanning tree by an edge outside of it, making sure no cycle is formed. The resulting graph is a spanning tree as well. We need to employ some policy to ensure no duplicates are generated, but on the contrary with the previous one, this method produces only spanning trees.

For the purposes of our work, we will be implementing algorithm from the first described group, one that was proposed by Cristian E. Onete and Maria Cristina C. Onete. [13]

1.5 Matrix Tree Theorem

Although spanning tree enumeration can serve as a method of counting spanning trees, when we are interested only in the number of spanning trees their listing and time taken

for their enumeration becomes redundant. In these cases, we can use Matrix Tree Theorem, also know as Kirchhoff's theorem, named after Gustav Kirchohoff, which computes this number using determinant of a matrix Q of size $n \times n$ for a graph G with vertex set $\{v_1, v_2, ..., v_n\}$.It works for loopless graphs, so graphs in our work satisfy this condition. Multiple edges are allowed, but since we are working with simple graphs, we can suppose that the number of edges $\{u, v\}$ for any vertex u and v in the graph is always 0 or 1.

Then the elements of the matrix Q are defined as

$$Q_{i,j} := \begin{cases} deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \\ 0 & otherwise \end{cases}$$

Basically, we take a $n \times n$ matrix with vertex degrees on the diagonal, other elements are set to 0, and subtract the adjacency matrix of G.

After this step, we delete any row s and column t of Q, obtaining Q^*. Once we have computed determinant of Q^*, $detQ^*$, the number of spanning trees in G equals $(-1)^{s+t} detQ^*$. [1, p. 67]

1.6 Graph classes and their bounds for number of spanning trees

While the question of number of spanning trees in a particular graph is straightforward - it can be easily answered with use of above described Matrix Tree Theorem, there is another question and that is generalisation of number of spanning trees for a specific class of graphs. In this case, we may not always be searching for one concrete number, but rather for example for a relationship between the number of vertices in a graph and its number of spanning trees, which we could use to determine the number after being given only the number of vertices. Even more general approach would be to estimate upper or lower bounds for the number of spanning trees in a particular class in terms of number of vertices.

We will go through some common classes of graphs and discuss the available information about their number of spanning trees.

1.6.1 Trees

Since any tree T is an acyclic connected graph, by removing any of its edges, it will become disconnected. This means that is has only one spanning subgraph, which is T itself. We already stated, that T is a tree, combined with te fact that T is its own spanning subgraph(this applies to all the graphs, not only trees) it meets the definition of a spanning tree.

Therefore, T has exactly one spanning tree, T, and one isomorphism class containing solely T.

1.6.2 Cycles

C_n are connected graphs with all of its vertices in a single cycle of a length n. If we remove any of its edges, we will get acyclic subgraph G with $|V(G)| = n-1$ and $n-1$ edges, which is by definition a tree. The resulting graph will remain connected, with the same vertex set as C_n, meaning that it is spanning subgraph as well. This makes G a spanning tree of C_n.

C_n has n edges, so the number of its different spanning trees is n, each one of them is obtained by removing a different edge from the edge set of C_n.

However, the number of isomorphism classes of spanning trees in C_n is 1 - all of them are path graphs on n vertices.

1.6.3 Complete graphs

Complete graphs, K_n, have maximum possible number of edges in a graph on n vertices, exactly

$$\frac{n \cdot (n-1)}{2} \tag{1.1}$$

Each vertex $v \in V$ is adjacent to all the vertices $u \in V(K_n) \setminus \{v\}$, so while looking for a spanning trees of K_n, we can instead see this problem as constructing all possible labeled trees on n vertices. These trees cover all the vertices in vertex set of K_n which makes them spanning trees of K_n.

The number of labeled trees on n vertices is defined by Cayley's formula, n^{n-2}.

In 1918, Prüfer found proof for this theorem, using function that would assign each tree a unique code, a sequence of length $n-2$ with entries from $[n]$, which is a set of natural numbers $\{1, ..., n\}$, where $n \in N$. There is a bijection between the set of trees on n vertices and the set of above mentioned sequences, hence both sets have the same cardinality, n^{n-2}.

$f(T)$, which computes Prüfer sequence for a labeled tree T, is defined iteratively. In each step, we delete leaf with the smallest label and add label of its only neighbour to the end of the sequence. This way, we perform $n-2$ iterations, producing a sequence of length $n-2$ and leaving one edge.

Inverse function to f is a function that produces a tree T from each sequence, where $f(T) = s$. Beginning with a forest with all the vertices from $[n]$ and empty set of finished vertices, in $i-$th iteration, edge xy is added and vertex y is marked finished. x is the label of a vertex in i-th position of s and y is the smallest label not yet included

in finished vertices and not appearing in later positions of s. After $n-2$ steps, we remain with two unfinished vertices, which are then joined by an edge. [1, p. 63]

Similarly, the number of non-isomorphic spanning trees of K_n is the number of non-isomorphic trees on n vertices. Every complete graph has a spanning tree that is a path, a star(even though in $K_n, n \leq 3$, there is only one spanning tree that is a path and at the same time a star) and the number of isomorphism classes grows for higher values of n. The sequence of these values for $n \geq 1$ is $1, 1, 1, 2, 3, 6, 11, 23, 47, 106, 235, 551, 1301, 3159, ...$, it can be found to a greater extent in The On-line Encyclopedia of Integer Sequences. [12]

Here is an example for K_5, which has three isomorphism classes. We are showing one spanning tree from each class. After we count cardinality of each isomorphism class, we reach number $125 = 5^{5-2}$, which represents the number of all different spanning trees of K_5.

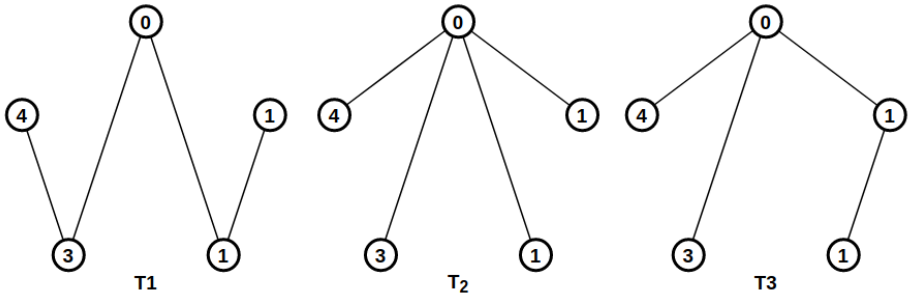


Figure 1.6: Representants of each isomorphism class of K_5's spanning trees

1.6.4 Regular graphs

Regular graphs don't necessary have to be connected, but for the purposes of determining the bounds for number of spanning trees, we will take into account only those with one component. Any disconnected regular graph has no spanning trees, which is information that wouldn't help us specify the relationship between the number of vertices and spanning trees.

Since the papers our thesis is based on suggest bounds only for k-regular graphs where $k \geq 3$, we will first separately discuss 0, 1 and 2-regular graphs. For illustration, figure 1.7 shows all possible unlabeled k-regular graphs for $k = 0, 1, 2$ with $|V(G)| \geq 5$ which are connected.

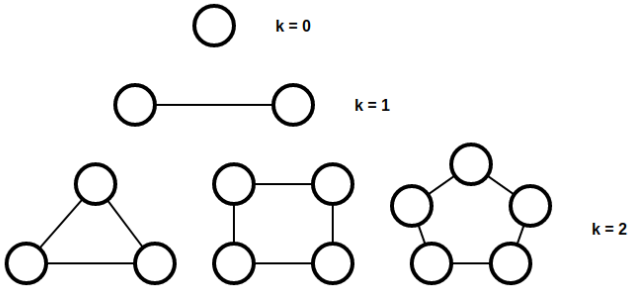


Figure 1.7: All connected unlabeled 0, 1 and 2-regular graphs on up to 5 vertices

$k = 0$ 0-regular graphs contain only isolated vertices and therefore their number of spanning trees is always 0 once their vertex set consists of more than one vertex. And although 0-regular graph on 1 vertex has empy edge set, it is a tree by definition, which means that it is its own spanning tree.

$k = 1$ There is exactly one unlabelled connected 1-regular graph - two vertices connected by a single edge. We can see, that it satisfies one of the earlier mentioned characteristics of a tree. Therefore, it has exactly one spanning tree.

$k = 2$ As we stated in the information about 2-regular graphs, they consist of one or more disjoint cycles. In the cases when the number of disjoint cycles is greater than one, the graph is disconnected, so we will focus on those where only one cycle on n vertices is present. Again, this case case be answered with one of our previously discussed classes, cycle graphs. Then number of different spanning trees of any 2-regular connected graph is hence n.

$k \geq 3$ In 1983, Brendan McKay's article Spanning Trees in Regular Graphs was published. In the introduction, he stated a theorem setting upper bound for k-regular trees on n vertices where $k \geq 3$. According to the theorem, the number of spanning trees of such graph is at most

$$\frac{(\frac{n \cdot k}{n-1})^{n-1}}{n} \tag{1.2}$$

[8, p. 149]. For 3-regular graphs, this means that their number of spanning trees can't be higher than 16, 100, 696, 5080, 38443... for $n = 4, 6, 8, 10, 12...$

Later, in 1990, Noga Alon went further into this problematic in his The Number of Spanning Trees in Regular Graphs [6] and proposed different set of bounds. One of the results presented there is that the number of spanning trees of the currently discussed k-regular graphs is always greater than

$$2^{\frac{n}{2}} \tag{1.3}$$

Another observation introduced is that this number isn't greater than

$$(\frac{1}{n-1}) \cdot k^n \tag{1.4}$$

McKay's theorem, however, presents slightly more narrow and thus better bounds.

McKay's and Alon's upper bounds are compared in figure 1.8 on case for 3-regular graphs.

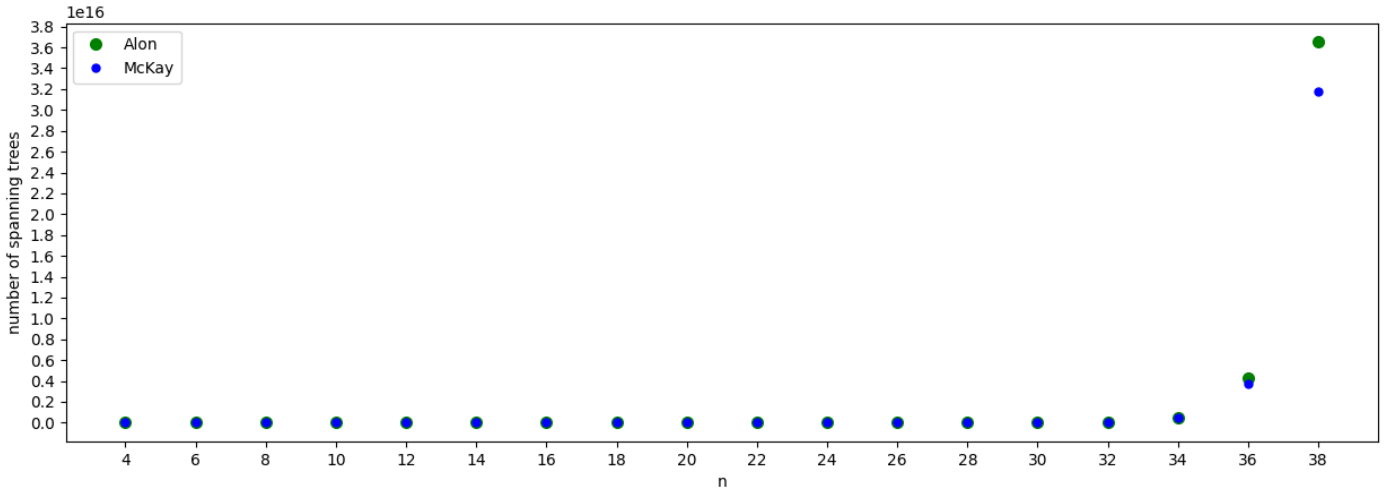


Figure 1.8: Comparison of Alon's and McKays upper bounds for regular graphs

Noga Alon proposed another upper bound, it can however be applied only to higher values of n. As n tends to infinity, the number of spanning trees in every k-regular graph, where $k \geq 3$ is at most

$$((k+1)^{k-2} \cdot (k-1))^{\frac{n}{k+1}} \tag{1.5}$$

We probably won't be able to address it in our experiments, because we will be working with lower values of n. For higher n, the amount of different possible k-regular graphs is becoming too large and difficult to process in reasonable time.

Chapter 2

Implementation

In this chapter, we will discuss the experiments we decided to conduct, technologies and algorithms that we opted for. We will also go through the most important classes in the programs and their functions. Finally, we will introduce the software that implements our experiments.

2.1 Methods

In this section, we will outline experiments which we would like to conduct and analyze in order to give bounds for number of spanning trees is specific classes of graphs. Our thesis was inspired by Noga Alon's article about spanning trees in regular graphs and since it seemed to be a problem which still has room for further research and development, we decided to focus mainly on regular graphs in our experiments as well.

I. Firstly, we would like to process sets of k-regular graphs on up to n vertices (for n ranging from lowest acceptable value for specific k to values where it's still possible to generate and process all graphs in reasonable time with our current software). For each graph, we will compute its number of different spanning trees and we will filter out graphs which have minimum and maximum number of spanning trees overall for n. After having collected enough data, we will separately examine all graphs with min/max number of spanning trees. We would like to find out, whether we can deduce something general about how the numbers or graphs are developing with increasing n. We will be working only with $k = 3$ and $k = 4$. The reason behind this is that for higher k, the set of all unlabeled connected graphs on n vertices is too large even for lower values of n. For information about number of graphs for specific n and k, we were using The On-Line Encyclopedia of Integer Sequences's website [7].

II. Second experiment is derived from the previous one. The only difference is that we won't be working with regular graphs, but biregular. We decided to consider only

those biregular graphs on $n+1$ vertices, where n vertices are of degree k and only one vertex of a different degree k_1. We would like to test different combinations of n, k and k_1 and examine how the results are differing from results for k-regular graphs on n vertices. Besides that, we will be still interested in the development of result data for fixed k and k_1, with n increasing.

III. In the last experiment, we would like to try a different approach. This time, we won't be dealing with numbers of all different spanning trees(those that have at least one different edge). Instead, we will be counting only different unlabeled spanning trees. To achieve this, we will need to generate all spanning trees of a graph and determine which ones are isomorphic. We might even sort the spanning trees into isomorphism classes and study cardinality of each class.

After having obtained sets of unlabeled spanning trees for two graphs, the next step of this experiment would be to compare cardinality of these sets with regard to the number of labeled spanning trees in the graphs. We would be interested in knowing whether a graph with more labeled spanning trees will always have also more unlabeled spanning trees. Another thing worth attention might be the number of unlabeled spanning trees that are present in both graphs. There are many possible graphs on which we could study these characteristics, for the time being, we decided to only work with pairs of graphs taken from the results of experiments described in the first paragraph - i. e. pair of k-regular graphs on n vertices where one of them has minimum, the other maximum number of spanning trees for n and k.

We ordered the experiments according to their priority for our thesis. It is possible, that we won't be able to cover all experiments equally and study all of the results of less prioritized ones in detail. This applies especially for the third experiment, which is actually a set of related experiments of which each could be further specified and developed. These remaining problems might be addressed in possible future works.

2.2 Genreg

To conduct experiments with regular graphs, we will need to generate all graphs on given number of vertices and with given regularity. Regular graphs are a widely studied class of graphs, so a generator with the required features has already been implemented. Genreg is a program for generating all connected non-isomorphic graphs for given number of vertices and regularity. Its principles were introduced in Markus Meringer's work Fast Generation of Regular Graphs and Construction of Cages[4]. It is written in the C programming language. When running Genreg, we have to specify at least two arguments, n and k, where n represents number of vertices of generated graphs and

k is their regularity. Another option, which we aren't using in our experiments, is to choose minimal t, for girth(length of the shortest cycle in a graph) of graphs. When not specified, it is set to 3. Generated graphs can be stored in ASCII file named $n\_k\_t.asc$ or written to standard output. Entry for each graph consists of number of the graph, the graph itself represented as adjacency list, girth of the graph followed by list of its automorphisms and finally size of graph's automorphism group. Genreg also writes out general information about the generation, the most interesting are the total number of generated graphs and the required CPU time. With a designated flag, it is possible to redirect this information to a file with $.erg$ suffix. With $-m$ flag, GenReg also offers an option to split the generation into several parts, lets say j, each will compute a disjoint set of graphs union of which is the set of graphs that would be generated by single generation. For lower j the number of graphs generated by each part is around equal, however, as j grows, there might occur more noticeable differences.

Since the size of files with stored graphs starts to exceed several dozens of GB as n grows, we opted for the results being written to standard output from where it can be redirected to our program. Also the increase in runtime is rapid and for higher values of n, it would soon become impractical to generate graphs in a single generator. For these reasons, we are using the $-m$ flag to divide the generation into sufficient number of partitions which we then run in parallel.

```
Graph 1:

1 : 2 3 4
2 : 1 3 4
3 : 1 2 5
4 : 1 2 5
5 : 3 4 6
6 : 5 7 8
7 : 6 9 10
8 : 6 9 10
9 : 7 8 10
10 : 7 8 9
Taillenweite: 3

9 : 1 2 3 4 5 6 7 8 10 9
7 : 1 2 3 4 5 6 8 7 9 10
3 : 1 2 4 3 5 6 7 8 9 10
1 : 2 1 3 4 5 6 7 8 9 10
1 : 9 10 8 7 6 5 4 3 1 2
1 : 10 9 8 7 6 5 4 3 2 1
Ordnung: 32
```

Figure 2.1: Example of output of genreg

2.3 Programming language

Original versions of our programs were written in Python. However, we soon have to replace it, as it has proven to be too slow for computations on higher number of graphs. Python is interpreted dynamic typed language, with garbage collector. It's syntax is simple and easy to read, thus making it the fist choice to test our ideas.

But due to its dynamic typed and interpreted nature, it is slower than some other available programming languages. Speed was crucial to our work, since we needed to perform computations for large sets of graphs. We have decided for C++, which is, on a contrary to Python, compiled and statically typed language. It is highly optimizable, which is making it suitable for our purposes. We only need to be careful while allocating memory - C++ doesn't have garbage collector, so it's necessary to free memory manually in order to avoid memory leaks. Also, we have to make sure we are using suitable data types, as choosing too short numeric type for calculating the number of spanning trees could result in several hours of computations to have no effect due to the results being overflown.

2.4 Bash scripts

Besides programs in C++ programming language, we employed also bash scripts, which are files with sets of commands that are interpreted by Unix/Linux shell. They allow us to prepare a sequence of commands that will be executed consecutively any time we run the bash script. We use it mainly for simple manipulation with files and process launching.

2.5 Algorithms

2.5.1 AHU algorithhm

For determining whether two trees are isomorphic, we will use a version of AHU(Aho, Hopcroft, Ullman) algorithm which will represent each tree as a string. This algorithm has time complexity $O(n)$, where n is the number of vertices in the tree.

Since AHU algorithm works for rooted trees, to be able to use it, we first have to find a centre of our undirected spanning trees and use it as a root. A center of a tree is a vertex, from which the longest path to a leaf is minimal over all vertices in the tree. The algorithm for identifying centre(s) of a tree is following: First we choose a random vertex v, then we find $v1$, vertex most distant from v and subsequently $v2$, the most distant vertex from $v1$. There can be one or two centres, depending on the parity of distance between $v1$ and $v2$ - the centre is a vertex in the middle of te path from $v1$ to $v2$. We use each centre as a root of tree and change the orientation of edges accordingly.

If the two trees have the same number of centres, we run AHU algorithm on each combination of rooting in both trees and compare the output strings. Trees are isomorphic if any of these combinations produces the same string. However, if the couple

of trees has a different number of centres, there is no need to use AHU algorithm, since they cannot be isomorphic and have the same structure.

After rooting a tree, we can now use modified AHU algorithm to construct a strig capturing the structure of this tree. Non-leaf vertices and their subtrees are represented by string in this format: "0x1", where x is a string we get after putting string codes of all their children of the current vertex in lexicographical order. Each leaf node is assigned "01". Starting at the root, we recursively construct characteristic string for all vertices. The string assigned to root node represents also the structure of the whole tree.

2.5.2 Spanning tree enumeration

The algorithm we used to enumerate all spanning trees of a graph is Onete's algorithm[14]. It is based on operations performed over matrices constructed for a given graph $G(|V(G)| = n$, $|E(G)| = m$ and $E(G) = \{e_1, e_2, ..., e_m\})$ - diagonal matrix $Diag$ and reduced incidence matrix $RInc$.

$Diag$ is of size $m \times m$ defined element wise as following:

$$Diag_{i,j} := \begin{cases} e_i & \text{if } i = j \\ 0 & otherwise \end{cases}$$

The construction of incidence matrix was described earlier in section 1.2. If we remove any of its rows, we obtain $RInc$. The next step if to perform the transpose of $RInc$, producing $RInc^T$. In our implementation of the algorithm, we construct $RInc^T$(of size $m \times (n-1)$) without previously constructing $RInc$ by switching its rows and columns.

The next step is to form matrix U which is a product of multiplication $Diag \times RInc^T$. It is again of size $m \times (n-1)$ and any combination of $n-1$ of its rows contains set of edges that is a candidate for spanning tree of G. Since also non-tree edge sets are formed, we need to perform certain tests to filter those out. First we check whether there is a row with only one nonzero entry among the chosen $n-1$ rows - only combinations that satisfy this condition are left for further examination. Also combinations with less than $n-1$ different edges were omitted. We were using Onete algorithm along with AHU algorithm(we computed characteristic string(s) for every spanning tree) so we decided to use it as the final test, to detect the edge sets that would form a disconnected graph and thus not a tree. When working with tree on n vertices, the characteristic string of such tree consists of exactly $2n$ characters, since each vertex adds to the string characters 0 and 1. If we run AHU algorithm on a non-tree, it will work only with one of its components - the one containing the vertex we chose as start vertex while searching for centre of the graph. This means that the characteristic string won't be computed using all vertices of $V(G)$ and therefore its length is less than $2n$.

2.5.3 Spanning tree counting

When we need to compute the number of spanning trees in a given graph, we use Matrix Tree Theorem. As described in section 1.5, the first step is to construct matrix Q of size $n \times n$, then delete one of its columns and rows and finally compute the determinant of the resulting matrix Q^*. In our program, we merged the first and the second step - we construct matrix of size $n-1 \times n-1$ right away, following the rules for construction of Q and omitting the column and row of Q with index 0.

The method for computing determinant of a matrix is Gauss elimination, with time complexity $O(N^3)$. It is significantly more effective for bigger matrices compared to another method - using idea of Laplace expansion and thus recursively computing determinant of a matrix of a dimension one smaller, whose time complexity is $O(n!)$.

2.6 Classes and their functionality

In this section, we will introduce some of the most important functions and their main functionality. While working with graphs, representation in a form of edge lists was sufficient most of the time. We implemented it as a vector of vectors, where the inner vectors were of size two and represented one edge. When the algorithms required more specific representation, for example to avoid repetitive search through the list of edges, we opted for adjacency lists(with the use of map) or various matrices.

2.6.1 SpanningTreeCounter

SpanningTreesCounter class implements The Matrix Tree Theorem for graphs on n vertices. We represent matrix as two-dimensional vector of integers. It has one public method, *countForGraph*, its only argument is an edge list of a graph for which then runs computations and returns its number of spanning trees of type long long.

2.6.2 ProcRegular

The purpose of class ProcRegular is to allow processing on k-regular graphs on n vertices and filter out the graphs with minimum and maximum number of spanning trees. It has variables for holding values of k, n, current minimum and maximum number of spanning trees, sets of graphs with such numbers of spanning trees. Since it is highly implausible that the final set of any of these graphs will be exceptionally large, at least for the values of n and k that we are able to process in reasonable time, we limited the number of stored graphs to 1000 in each set. Without this restriction, the accumulation of graphs in partial results was causing the program to be terminated by out of memory killer. Another attribute is *counter*, which is an instance of SpanningTreeCounter

initialised with n and k.

Its base function is *processAll* which reads from standard input until it has collected information about all n vertices. At that point *processGraph* is called. Firstly, it parses the input lines and transforms adjacency list to edge list of the graph. Then we pass the edge list to the *counter* in *countForGraph* and compare the returned number of spanning trees with the current values stored in minValue and maxValue. If necessary, we update them accordingly. After the end of input, it writes out the results to a file. In figure 2.2, it is shown how the results are formated. In the first line, there is string mix or min, depending on whether it represents minimum of maximum number of spanning trees, number of graphs collected in this category and finally the number of spanning trees. If not specified, the name of output file is maxMinReg$k$-n.txt.

```
min 1 75
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
max 1 81
[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]
```

Figure 2.2: Format of results

For further purposes, processing unit will be a term used to describe a program which gets n, k and optionally other arguments arguments, creates an instance of ProcRegular or its subclass initialised with these values and calls its *processAll* method.

2.6.3 ProcBiregular

ProcBiregular is a class derived from ProcRegualr and is designed to work with biregular graphs as was described in 2.1. In addition to inherited properties, it stores the degree of the $n+1$st vertex u. It also implements function *edgeCombinations* which tries to remove all possible combinations of $\frac{k_1}{2}$ edges from a given k-regular graph on n vertices. It then adds an edge between the additional vertex to each of the $(k-1)$-regular vertices and computes the number of spanning trees of the resulting graph. In the process, it is checked whether the latest removed edge wasn't incident to any vertex v that has already had an edge removed and thus would lead to a multiple edge between v and u. This method is called from *processGraph* instead of directly computing and evaluating the number of spanning trees of the regular graph. The default name of output file is maxMinReg$k$-n-k_1.txt.

2.6.4 ColRegular

Similarly as ProcRegular, also ColRegular operates with k-regular graphs on n vertices. It is used to collect results from files $prefix$-i, where i ranges from 1 to j. $prefix$ and j are specified in the initialisation. The format of the files is the same as described in

2.6.2. This is done in method *collectResults* - it consecutively reads from files, evaluates their results the same way as ProcRegular, it also uses analogical data structures to store partial results. The final results are written to maxMinReg$k$-n-j.txt.

A program to which we pass n, k(and possible optional arguments) which are used to initialise an instance of ColRegular or its derived class and consequently calls on it *collectAll* method will be referred to as collecting unit.

2.6.5 GeneralFunctionsForGraphs

This class offers various public functions for graphs that can be called from methods of other classes. These functions are static, so every time any of the is called, it requires input data of the graph we wish to operate with. *checkGraphExistence* takes four arguments: n_1, k_1, n_2 and k_2. It returns true if it is possible to construct a graph with n_1 vertices of degree k_1 and v_2 vertices of degree k_2. *countSpanningTrees* returns the number of spanning trees of the given graph, which is passed to the function either as an edge list in a form of string or as a vector of vectors. Isomorphism of two trees is determined by of *isomorphicGraphs*, there are again the same two options as previously in which format to pass the input trees. *spanningTreeBFS* returns edge list of the first spanning tree of a graph that is find by using breadth first search. *generateAllSpanningTrees* returns a map where keys are representants from each isomorphism class of the set of all spanning trees of the given graphs. The value for the specific key is then the cardinality of the corresponding isomorphism class. Finally, *compareUnlabeledSpanningTrees* takes two graphs on the same number of vertices as arguments and prints unlabeled spanning trees that are present in the both graphs.

2.6.6 ColBiregular

Class ColBiregular is derived from ColRegular with slight modifications to enable collection of results for biregular graphs, with additional information about the value of k_1. The name of output file is maxMinBireg$k$-n-k_1-j.txt this time.

2.7 Design of graph generation and processing

While working with genreg, we used two different approaches. Both variants are using pipes, mechanisms for passing data from one process to another. They work in unidirectional way where transferred data are buffered by kernel. The receiving process is suspended until there are some data to read from the pipe. Also the sending process is suspended when the buffer is filled.

2.7.1 Serial processing

The first one is serial generating and processing. This approach is suitable especially for the cases of lower values of n where splitting generations into smaller parts isn't allowed in genreg. The number of generated graphs there doesn't seem to exceed 1000, so the whole process is relatively fast.

Here is a template for the commands that we use in order to generate and process the graphs. In some versions, there are additional arguments, but they aren't crucial for the understanding of the scheme.

GenReg n k -a stdout | processingUnit n k [$k2$]

First we run genreg with desired n and k, writing its data to standard output. We redirect them to standard input of our processing unit through a pipe. The processing unit is launched with the same n and k as its arguments, reads from standard input and processes the data as we described sooner.

After the generation is finished and our unit processed all graphs, the final results are written to a file with a default name corresponding to particular processing module used.

2.7.2 Parallel processing

The other approach we employed is to run several composite units, similar to the one in the previous case, in parallel. We split the generation into desired number of partitions, lets say j, through genreg's $-m\ i\ j$ option. For 3-regular graphs, this option is available when $n \geq 16$, for 4 and 5-regular graphs when $n \geq 12$.

The template for the commands is slightly different in this case:

GenReg n k -a stdout -m i $pocetCasti$ | processingUnit out-graf-i n k &

This command is run j times, with i ranging from 1 to j. They are run in background, so that they could work concurrently and wouldn't block other processes until they are finished. The output of genreg is again redirected through pipe to standard input of processing unit. Besides n and information about degrees of vertices, our unit is given also the name of the file to write their results to - $out - graf - \{i\}.txt$.

However, when the processing is distributed into several parts, the same goes for the results. Each processing module has its own set of partial results which we need to merge and filter to acquite the final results. This is achieved through collecting unit, which reads data from all out files and evaluate them in the same way as processing units do. Only after this step we store the data to final output file. This unit is launched once all genregs and processing units are finished, using *wait* in the bash script. In figure 2.3, we can see how these units work together.

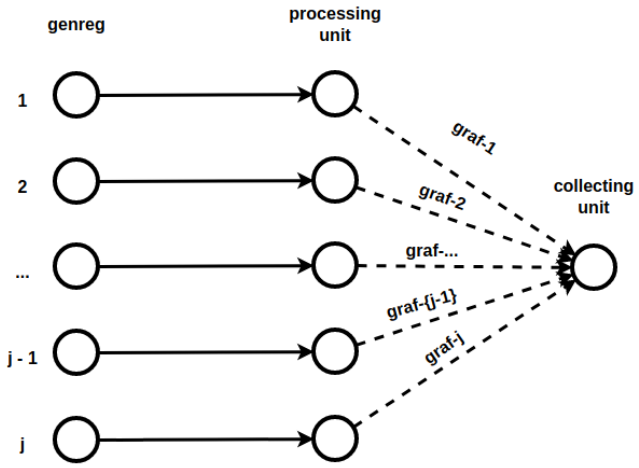


Figure 2.3: Scheme for parallel generation and graph processing

2.7.3 Unsuccessful prototype

In both cases that were described above, the processing of graphs within one processing unit, regardless of whether it is in serial or parallel version, is done sequentially. Graphs are processed one by one and their data is discarded once their evaluation finished. Exception to this is a situation where a graph was used to update the current minimum or maximum number of spanning trees. Originally, there were attempts to make also the processing parallel. We would use a form of batch processing, where we would first gather a certain number, lets say m, of graphs in the main thread and then pass it to a new thread that would run in parallel. The number of concurrently ruining threads, th, would be limited to. The key to this approach was to find the right configuration - number of running threads and number of graphs in one batch. Small values of m were resulting in the process being ineffective, and too high th or m were causing the program to be killed by the out of memory killer due to need to hold a large amount of data. Although it worked for smaller sets of graphs, as the number of vertices was growing, these problems started to be more prominent and we had to search for new methods, since it was difficult to configure the settings in a way that we were able to finish computations and at the same time to accomplish the desired efficiency. Methods introduced in section 2.7.1 and 2.7.2 have proven to be more universally applicable, so we we didn't include this approach in the final product.

2.8 Tools for experiments

The final software for experiments is a set of executable files - binary files from C++ source code(all necessary target files are generated after running make install) and

bash scripts which operate with most of the binary files. Bash scripts expect these files to be stored in /usr/bin/, or to have the paths to them added in PATH. Another prerequisite for successfully running bash scripts is having executable of genreg, under name GenReg, managed the same way as our own binary files. All of the programs are to be run in command line.

2.8.1 Minimum/maximum numbers of spanning trees

Bash scripts handle all experiments which are based on graph generation and processing. Here is a list of available bash scripts with a short description of how are they operated and when to use them. Their output is a file with a list of graphs with specified parameters which have minimum and maximum number of spanning trees for a given number of vertices.

regularSerial This script is designed for the first experiment from 2.1. It is run with two arguments, *regularSerial* n k, where n is number of vertices and k is regularity. After it finishes, we will obtain file maxMinReg$k$-n.txt.

biregularSerial With the next script, we can work with biregular graphs, as was described in the second experiment. *biregularSerial* n k k_1 runs the script for graphs on $n+1$ vertices where all vertices, except one vertex with degree k_1, have k neighbours. The name of output file is maxMinBireg$k$-n-k_1.txt.

regularGirthSerial It wasn't planned to implement this functionality originally, however, after experimenting with regular graphs, it has turned out that it might be helpful to study the number of spanning trees in graphs with limited values of girth. This script is used to examine a set of k regular graphs on n vertices in which girth is always at least t. If t is highest possible for the current combination of n and k, we will be dealing only with graphs with girth t. To run the process, we use the following command and order of arguments: *regularGirthSerial* n k t. The results are written to file maxMinRegGirth$k$-n-t.txt.

Parallel versions All of these three bash scripts are available also in parallel version. They have the same name, only with *Serial* being replaced by *Paralel*. They need one new argument j, which goes as first and it specifies into how many parts we want to split the generation and processing. The structure of names of output files has slightly changed to reflect the use of parallelisation, it is now $< originalName > k - n - [k_1/t] - j.txt$.

2.8.2 Unlabeled spanning trees and other funcionality

graphsFunctions covers the remaining experiments and also has some additional functionality for simple tests on graphs. It is operated through console interface, whose main menu can be seen in figure 2.4. Each option in the menu is assigned a number which needs to be typed in when we want to continue to this section. In each section, single operation on graphs is performed, user will be first requested to write number of vertices of the graph and then its edge list. It expects the vertices to be labeled from 0 to $n - 1$.

Option 1 allows us to compute the number of spanning trees of the given graph. In option 2, we can evaluate whether two trees are isomorphic. Option 3 generates all spanning trees of the given graph. User needs to enter the name of file into which the results will be written. To compress the results, only one representant of each isomorphism class of the spanning trees will be written, along with the cardinality of this class. The last graph-related option compares unlabeled spanning trees of two graphs and outputs those that are present in both graphs.

```
================================
spanning tree counting: 1
tree isomporphism: 2
spanning tree enumeration: 3
spanning tree intersection: 4
exit: 5
================================
```

Figure 2.4: Menu of cli interface

```
number of vertices: 6
edge list of first graph: [(0, 1), (0, 2), (0, 3), (1, 2), (1, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
edge list of second graph: [(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]
first graph:
75 spanning trees generated, 4 isomorphism classes
second graph:
81 spanning trees generated, 3 isomorphism classes
unlabeled spanning trees present in both graphs:
[(0, 1), (0, 2), (0, 3), (1, 4), (2, 5)]
[(0, 1), (0, 2), (0, 3), (3, 4), (3, 5)]
[(0, 1), (0, 2), (1, 4), (2, 5), (3, 4)]

3 total
```

Figure 2.5: Example of output for comparison of spanning trees in two graphs

Chapter 3

Result analysis

In this chapter, we will introduce and analyze the results of our experiments. We will try to answer the question whether it is possible to set closer bounds for the number of spanning trees in regular and biregular graphs. We will also discuss how efficient our software was while working with generated graphs.

3.1 Regular graphs with minimum number of spanning trees

When searching for minimum and maximum number of spanning trees in k-regular graphs, we were focusing mainly on cases where $k = 3$ and $k = 4$, as the number of graphs wasn't rising that fast with increasing n and we were therefore able to examine graphs for a wider range of n.

3.2 3-regular graphs

Firstly, we will have a look at 3-regular graphs with minimum number of spanning trees for given n. We generated and evaluated graphs for n ranging from 4 to 28. For $n = 4, 6, 8$, the structure of the graphs with the lowest number of spanning trees didn't seem to be particularly special. However, for $n \geq 10$, we started to observe some common traits in these graphs. The graphs were divided into a certain number of subgraphs, and these subgraphs were interconnected by single edges in such manner, that these edges were bridges(thus they all had to be contained in any spanning tree of the graph). In such graphs, there exists only one way how to connect two neighbouring subgraphs, lets call them partitions, so the total number of spanning trees is dependent only on the number of spanning trees of each partition. To calculate it, we need to separately count spanning trees in each partition and then multiply the results.

Another characteristic that these graphs shared was the structure of their partitions. Except for case when $n = 12$, only three non-isomorphic partitions were occurring and they seemed to serve as building blocks for 3-regular graphs with minimal number of spanning trees. Each of them was on a different number of vertices: 1, 4 and 5. Their structure can be seen in figure 3.1, with vertices with degree lower than 3 in red colour. The first one is simply a single vertex of degree 0, with one spanning tree. It therefore has to be connected to three other partitions to reach the degree of k. The second one, diamond graph, is formed from a complete graph on 4 vertices from which any edge is removed(all edges are equivalent in complete graphs). Thus it has two vertices of degree already 3 and two vertices of degree 2 - each of them is waiting to be connected to a different partition. Its number of spanning trees is 8. The last type of partitions, with 24 spanning trees, has only one vertex which is available for further connecting, with degree 2. It therefore serves as an endpoint in the structure of the whole graphs.

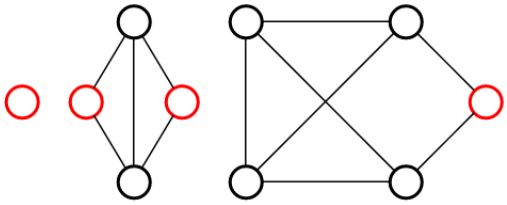


Figure 3.1: Building partitions of 3-regular graphs with minimum number of spanning trees

So far, once n is high enough, the graphs are formed in a simple way. For graph with minimum number of spanning trees on $n + 4$ vertices, we just take a graph with minimum number of spanning trees on n vertices, remove any of its bridges and connect the resulting two vertices with degree 2 to the two vertices of degree 2 in a new diamond graph partition - each of the two vertices in the original graph will be incident to a different vertex in the additional partition. We can easily see that the new graph will have exactly 8 times more spanning trees than the original one. The base graphs for this iterative formation are on 10 and 16 vertices, they have slightly different structure. We thus have two different branches of the formation - each with addition of 4 vertices but one staring with 10 and the other in 16 vertices. In both cases, the number of 5-vertex partitions remains intact. These two base graphs are shown in figure 3.2. The full list of the graphs can be found in Appendix A, section 3.8.

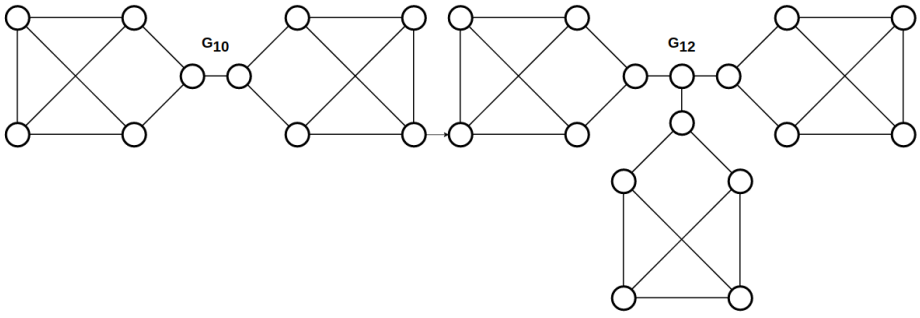


Figure 3.2: 3-regular graphs on 10 and 16 vertices with the lowest number of spanning trees

Based on the predictable structure of the examined graphs, we presume that we can determine the exact minimum possible number of spanning trees in 3-regular graphs on n vertices(for $n = 10 + 4i$ and $n = 16 + 4i$, $i \in \mathbb{N}$) and the structure of such graphs.

Our hypothesis is the following:

If $n = 10 + 4i, i \in \mathbb{N}$, then the graph with lowest number of spanning trees has two 5-vertex and $\frac{n-2\cdot5}{4}$ 4-vertex partitions, thus the number of its spanning trees is

$$24^2 \cdot 8^{\frac{n-2\cdot5}{4}} \tag{3.1}$$

Figure 3.3 shows the structure of there graphs. Bridges are connecting partitions in a single row. The value of x is $\frac{n-2\cdot5}{4}$.

Figure 3.3: Scheme of 3-regular graphs with lowest number of spanning trees on $n = 10 + 4i$ vertices

If, on the other hand, $n = 16+4i$, $i \in \mathbb{N}$, graphs with the lowest number of spanning trees have three 5-vertex, $\frac{n-3\cdot5-1}{4}$ 4-vertex partitions and one 1-vertex partition, lets call it central vertex. The 4-vertex partitions can be distributed to three branches stemming from central vertex in different ways, resulting in more non-isomorphic variants of such graphs. Their number of spanning trees is exactly

$$24^3 \cdot 8^{\frac{n-3\cdot5-1}{4}} \tag{3.2}$$

We can see their structure in figure 3.4, where $x + y + z = \frac{n-3\cdot5-1}{4}$

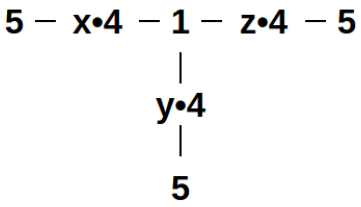


Figure 3.4: Scheme of 3-regular graphs with lowest number of spanning trees on $n = 16 + 4i$ vertices

If our hypothesis is correct, the lower bound for the number of spanning trees in 3-regular graphs it sets is significantly more precise than Alon's $n^{\frac{n}{2}}$. Figure 3.5 compares these two bounds, for n ranging from 14 to 40.

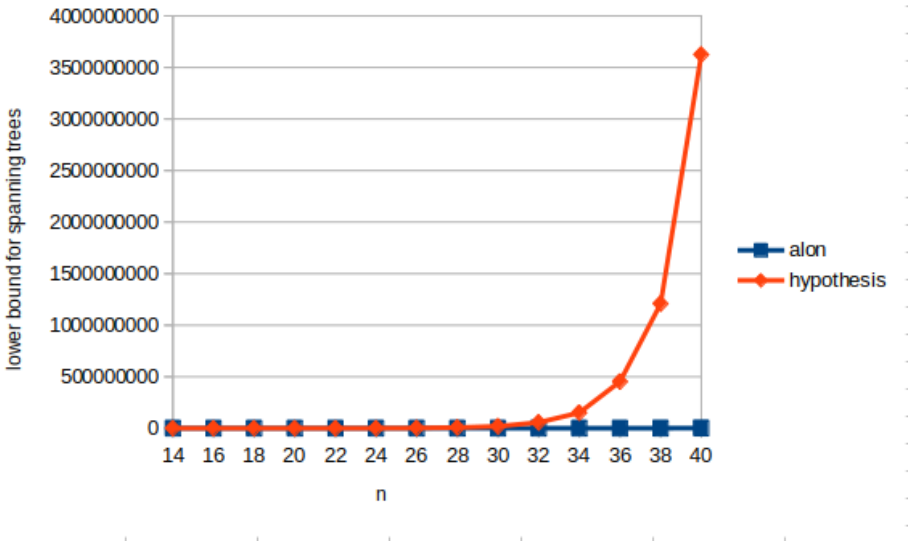


Figure 3.5: Comparison of Alon's and our lower bounds for number of spanning trees in 3-regular graphs

3.3 4-regular graphs

Also in $k = 4$, there was visible a certain pattern among the graphs with the lowest number of spanning trees for each n. We detected two partitions that appeared in these graphs, one on 4 vertices, all of degree 3, and the other one on 5 vertices, where only two vertices were missing one neighbour to reach degree of k. They can be seen in figure 3.6. Besides vertices embedded in partitions, additional vertices were present, but we couldn't find any system in they layout yet. We can only say that they serve as connection points for used partitions. The structure of the graphs wasn't so straightforward and easy to analyse as in the previous case. k-regular graphs where k is even can't have any bridges, so there is higher level of connection between the partitions. This also means that the total number of spanning trees is higher than as if we had partitions connected only by bridges, since now there are more possible ways how to choose spanning trees edges from a set of edges coming out from one partition. Although no bridges were present in these graphs(cut-sets of size one, in connected graphs, a cut-set is a set of edges by whose removal, the graph becomes disconnected), each graph had cut of size two. The exact lowest numbers of spanning trees in 4-regular graphs can be seen in 3.1, there is also column for highest numbers of spanning trees, since they won't be discussed separately. Section 3.9 in Appendix A contains representations of all 4-regular graphs with minimum number of spanning trees.

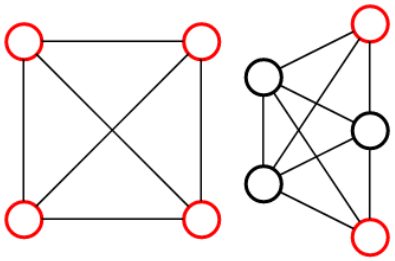


Figure 3.6: Building partitions of 4-regular graphs with minimum number of spanning trees

n	min	max
5	125	125
6	384	384
7	1183	1200
8	3456	4096
9	9600	12480
10	18750	40960
11	40000	130691
12	115000	428652
13	330625	1373125
14	900000	4423680
15	1900000	14466816
16	4000000	47775744
17	11000000	153399024
18	24000000	509607936
19	69000000	1687944960

Table 3.1: Maximum and minimum numbers of spanning trees in 4-regular graphs

3.4 Regular graphs with maximum number of spanning trees

We were examining only 3-regular graphs, again from 4 to 28 vertices. For each value of n, we found exactly one graph with maximum number of spanning trees. These numbers were 16, 81, 392, 2000, 9800, 50421, 248832, 1265625, 6422000, 32710656, 168664320, 862488000 and 4410450000. To gain a better understanding of these graphs, we tried to take a closer look at their structure. They weren't so easy to analyse, compared to previous cases, since they were more interconnected and any patterns or process of formation, if present, weren't so obvious. However, we formed an assumption about their shared characteristic and we later confirmed it using GenReg's option to

specify also minimum girth in generated graphs. All of our graphs have maximum possible girth for their number of vertices. Some of these graphs were cages - regular graphs with lowest possible number of vertices for a certain girth. Those were cases with $n = 4, 6, 10, 14, 24$. Assuming our hypothesis is correct, the search for 3-regular graphs with maximum number of spanning trees will be easier, with help of GenReg's minimum girth specification, we can delimit generated graphs to only those with highest possible girth, thus making the whole process faster. So far, we managed to go through cases with $n \leq 42$. All of these graphs are listed in section 3.10 of Appendix A. McKay's upper bound for number of spanning trees is 3-regular graphs is rising much faster than the actual maximum numbers, also when we are taking into consideration the estimated numbers. However, we don't have any general formula for the upper bound.

3.5 Biregular graphs with minimum number of spanning trees

Another case we studied were graphs on $n+1$ vertices, where n vertices were of degree 3 and the remaining, additional vertex was of different degree k_1. There exist several possible values of k_1 that are acceptable for a specific n and k. We used two different approaches, first was to set k_1 to a fixed value and generate graphs for n from range $< 4, 20 >$, if such graphs could exist. The other approach was to define k_1 with regard to n, more specifically $k_1 = \frac{n}{2}$ and $k_1 = n$. In both approaches the formation of graphs followed similar principles as in 3-regular graphs. The number of building partition increases and all of these new partitions included the vertex with degree k_1. This vertex is shared among new partitions, so they aren't connected by bridges, but by the vertex itself. We can see the most common building partitions in figure 3.7. The red vertices are those which will have degree k_1 in a full graph. In the case with fixed value of k_1, once n was high enough, the graphs were formed just as in 3-regular graphs - by removing one bridge and connecting the incident vertices to the ends of added diamond graph partition. If would be possible to create hypothesis for the minimal number of spanning trees in such graphs. In the second case, k_1 was growing with n, so besides from adding diamond partition, also the set of used biregular partitions was reorganised. Examples of graphs produced by both approaches can be seen in figure 3.8 and 3.9

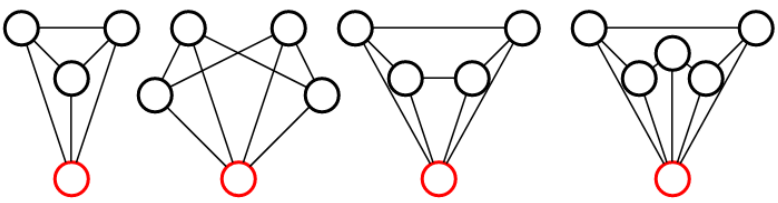


Figure 3.7: New building partitions for biregular graphs

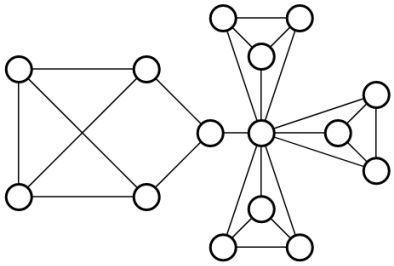


Figure 3.8: Biregular graph of order $n + 1$ with least spanning trees for $n = 14$, $k = 3$ and $k_1 = 10$

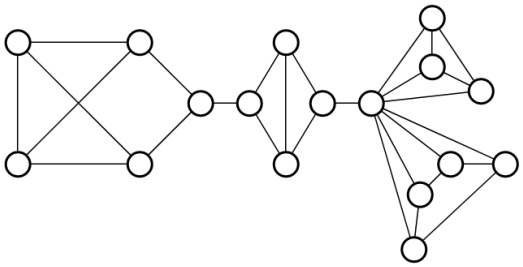


Figure 3.9: Biregular graph of order $n + 1$ with least spanning trees for $n = 16$, $k = 3$ and $k_1 = \frac{n}{2}$

3.6 Isomorphism classes

The last experiment we conducted was to compare graphs with the same n based on their number of labeled spanning trees and also number of isomorphism classes of spanning trees/unlabeled spanning trees of their spanning trees. The test sample we were working with were 3-regular graphs with minimum and maximum number of spanning trees on 4 to 18 vertices. We were also interested in how many unlabeled spanning trees have these pairs of graph in common. The results of this experiment are summarized in table 3.2, where columns min/max are for numbers of unlabeled spanning trees in graph with minimum/maximum number of spanning trees. The last column says how many unlabeled spanning trees are present in both graphs. We found out that higher number of spanning trees does not necessary mean that the graph will have also higher number of isomorphism classes of spanning trees, as shows case where $n = 6$ or $n = 10$.

n	min	max	∩
4	2	2	2
6	4	3	3
8	8	10	7
10	21	20	10
12	81	118	70
14	78	127	19
16	56	707	9
18	300	8438	259

Table 3.2: Comparison of number of isomorphism classes

3.7 Software efficiency

With our software, we can run generation of regular graphs and process its output in a reasonable time for even large cases, with a few milliards graphs. In most cases, the main factor that is affecting computation time is the number of generated graphs. This is something we can't change, we can only ease its effects by running the program with our parallel option on high enough number of cores. Also the number of available cores is an important factor, as splitting the generation and processing into appropriate number of jobs that are run is parallel is something that can significantly reduce the computation time.

For cases with higher number of graphs, we run our program on 42 cores, with generation split into the same number of parts. The biggest set of graphs we worked with were 3-regular graphs on 28 vertices, with total 40497138011 graphs. The generation and computation all together took almost $6,5$ days.

Table 3.3 shows computation times of serial generation and processing of 3-regular graphs on up to 22 vertices. We also added column with the total number of generated graphs. In table 3.4, we can see similar information, but this time, the generation of graphs was split into 8 parts which were run in parallel. The column with number of graphs now represents the highest number of graphs generated by one generator.

The software is the least efficient for biregular graphs, since we didn't add enough optimization when trying different possible combinations how to integrate the additional vertex into each regular graph generated by genreg.

n	time	graphs generated
4	0,010s	1
6	0,010s	2
8	0,010s	5
10	0,010s	19
12	0,013s	85
14	0,034s	509
16	0,185s	4060
18	1,993s	41301
20	28,271s	510489
22	8min 5,469s	7319447

Table 3.3: Computation times for serial generation of 3-regular graphs

n	time	max. graphs generated by one generator
16	0,094s	582
18	0,648s	6388
20	8,356s	73182
22	8min 5,469s	1093961
24	42min 28,092s	19401405

Table 3.4: Computation times for parallel generation of 3-regular graphs

When using genreg's $-m$ option for parallel generation, we encountered a feature that was degrading the time efficiency of our software. Ideally, each generator would work for around the same amount of time and produce around equal number of graphs. The whole process would finish the fastest this way, since the workload would be distributed evenly. However, as we said earlier, with higher j, there are chances of more prominent inequalities. We were working on 42 cores, so we decided for j to be also 42. For generations with lower number of graphs, these inequalities weren't that noticeable, since the process was relatively fast anyway. $n = 28$ and $k = 3$, with total 40497138011 graphs was a problematic case. If distributed evenly, each generator should produce around 964217572 graphs. However, 66% of generators didn't even exceed 900 millions graphs. The fastest one finished with 175283521 graphs and the slowest one with 3490608800, which was 1 milliard more than the second slowest generator. However, for $n = 20$ and $k = 4$, the division performed significantly better(the total number of generated graphs was higher this time, 152808063181). Because of the exponential rise of number of graphs, we didn't have time resources to test $-m$ flag for higher values of n, so the further attributes of this problem remain unexamined.

Conclusion

Our work was centered around examining graphs with minimum and maximum number of spanning trees on a given number of vertices in a specific classes of graphs, in order to propose bounds for the number of spanning trees in the said class.

First we designed a set of experiments, focusing on regular graphs since that field was already studied, which was giving us opportunity to compare our potential results with previously introduced relations. We implemented a set of command line tools allowing us to conduct these experiments. We have opted for a combination of an already existing technology for generating regular graphs - GenReg, C++ programming language for processing the graphs and bash scripts for the management of generation and the respective processing. The greatest challenges during work with large sets of regular graphs were time and memory requirements, which we managed to solve by GenReg's built in functionality and some add-on local mechanisms. Different approaches to working with GenReg were tested and we selected the ones that were the most universal and allowed us to work with a broad range of graph sets. The tools offer mainly scripts for generating k-regular graphs on n vertices, which are then filtered and the graphs with minimum and maximum number of spanning trees are returned as a result. User can specify n, k and in some cases other additional arguments.

We have collected data for 3 and 4-regular graphs for up to highest possible n with regard to being able to run the computations in reasonable time with our assets. As first we analysed the graphs with the highest number of spanning trees for each n. We have discovered, that these graphs are constructed from only a limited number of non-isomorphic subgraphs- building partitions- and are thus very similar in structure. For each value of k, the graphs have their own set of building partitions and specific ways of formation. The case of 3-regular graphs exhibited very predictable patterns, using brides to connect the building partitions in a manner that no cycle was formed outside the partitions. It appears that if we take a graph with minimum number of spanning trees on $n = 10 + 4i$, $i \in \mathbb{N}$ or $n = 16 + 4i$, $i \in \mathbb{N}$ vertices, we can transform it into a graph with minimum number of spanning trees on $n+4$ vertices by simply disconnecting any of the bridges and attaching them to a new partition on 4 partitions. Based on the iterative construction of these graphs, we proposed an equation to compute the lowest

possible number of spanning trees in 3-regular graphs for a given n.

While working with 3-regular graphs with maximum number of spanning trees, we have discovered that this topic seems to be related to the study of cycles in graphs, more specifically girth and cages. In all examined sets of graphs on n vertices, the graph with the most spanning trees has the highest possible girth. Therefore, with growing n, these graphs started lacking cycles of short length.

Similarly as in regular graphs, also biregular graphs with n vertices of degree k and one vertex of degree k_1 display the same behaviour when constructing a graphs with minimum number of spanning trees. We focused on cases where $k = 3$, with varying k_1. The resulting graphs reused building partitions from 3-regular graphs but introduced new ones as well. When increasing n by 4, the formation of the graphs was again very predictable which would allow us to determine the presumed lowest possible number of spanning trees for a specific combination n and k_1 and the structure of the corresponding graph.

Our experiments have opened many questions that were above the time resources available for the work. However, we find these questions to be worth studying more in detail in potential extensions of this thesis.

Although we have a hypothesis defining the exact number of spanning trees and structure of 3-regular graphs on n vertices with lowest possible number of spanning trees for the given n, this hypothesis hasn't been proven yet. We suggest proof by induction in two separate branches - one for $n = 10 + 4i$, $i \in \mathbb{N}$ and the other for $n = 16 + 4i$, $i \in \mathbb{N}$. In induction step would be done by adding 4 vertices to n, reflecting the iterative formation of the graphs. Maybe it would be possible to generalise this hypothesis for k-regular graphs where k is odd number(in graphs with odd regularity, existence of bridges is allowed and bridges seem to be key to the structure of the detected graphs when $k = 3$).

Another interesting addition to our work would be to process sets of 4-regular graphs $n \geq 20$ vertices. These cases will require resources for a higher level of work-load distribution than we had or/and more time available. This way, additional data would be gathered for more thorough examination of structure of graphs with minimum number of spanning trees for given n. The goal of this examination would be to determine whether we can not only identify all building partitions of 4-regular graphs with minimum number of spanning trees, but also to find a pattern in their formation and thus be able to predict their development for even higher n.

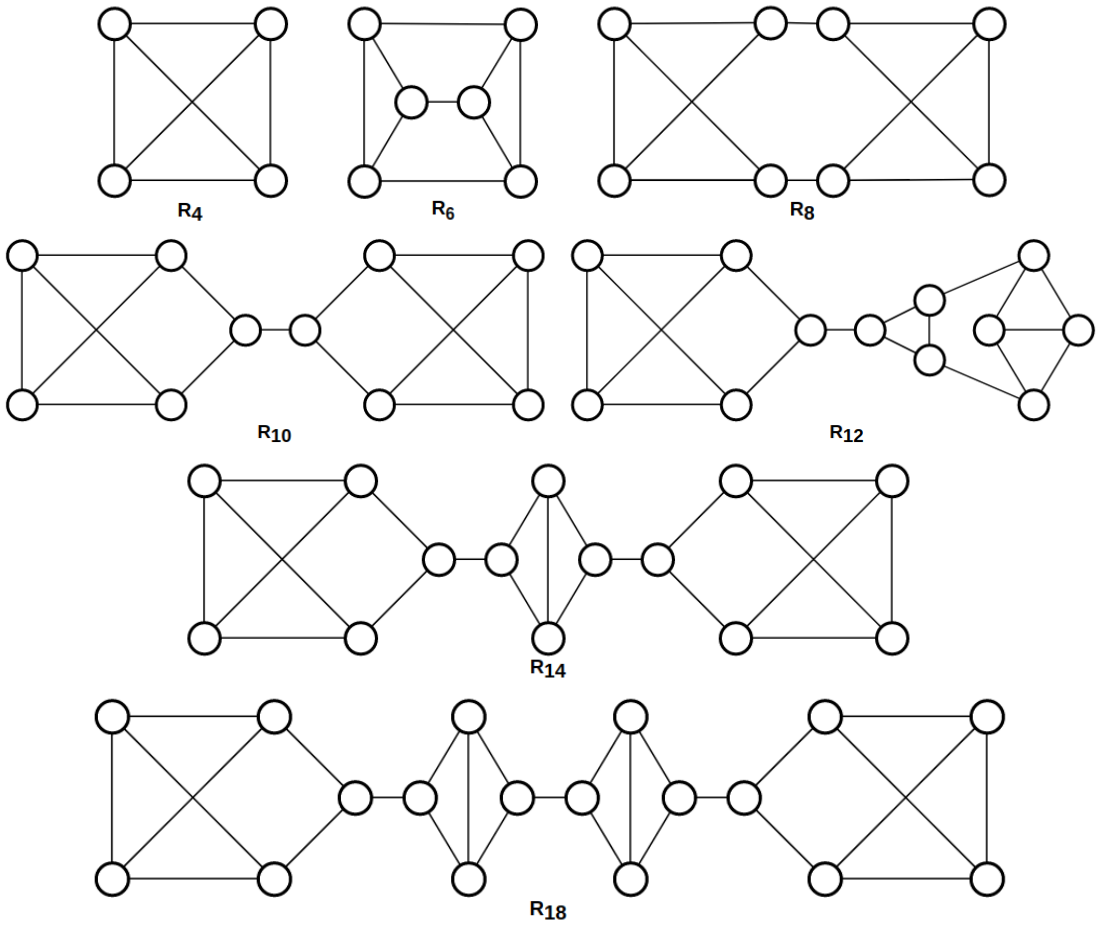
Also the question of 3-regular graphs with maximum number of spanning trees holds potential for further progress. Besides length of cycles, we could focus also on the number of different cycles in a graph with such length. To specify the task for this particular case, we would like to study the relation between the graphs with the most

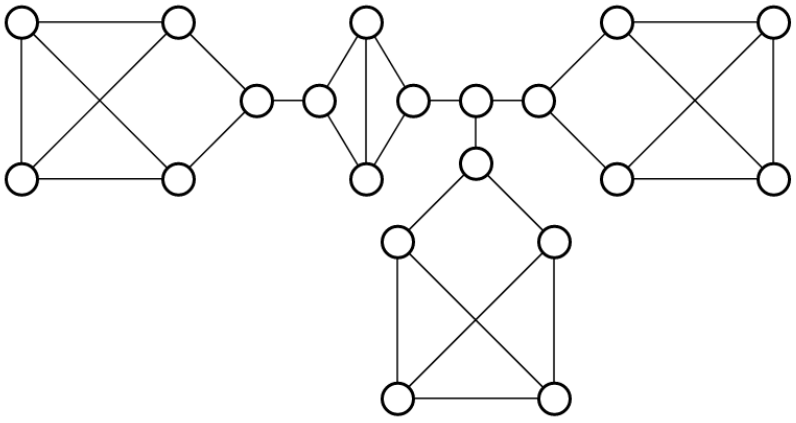
spanning trees for given n and the number of their cycles with the length of their girth. If there is any relation, will they possibly have the least such cycles among all other 3-regular graphs with the same girth?

Appendix A

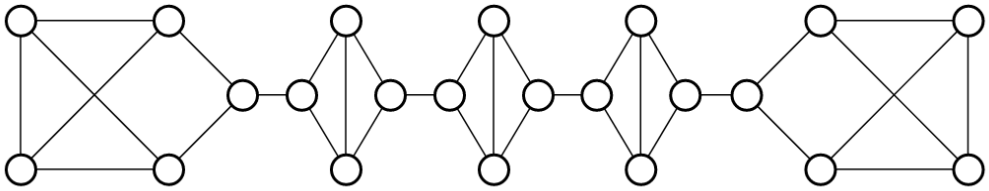
3.8 3-regular graphs with maximum number of spanning trees

Our hyposthesis about 3-regular graphs with minimal number of spanning trees and the structure of these graphs was proven for graphs on up to 28 vertices using genreg. In figure 3.11, we can see all detected graphs with minimum number of spanning trees for given n.

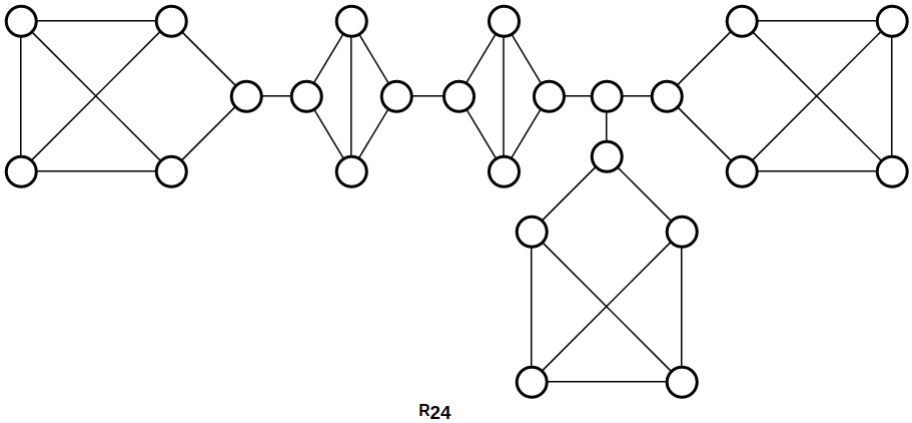




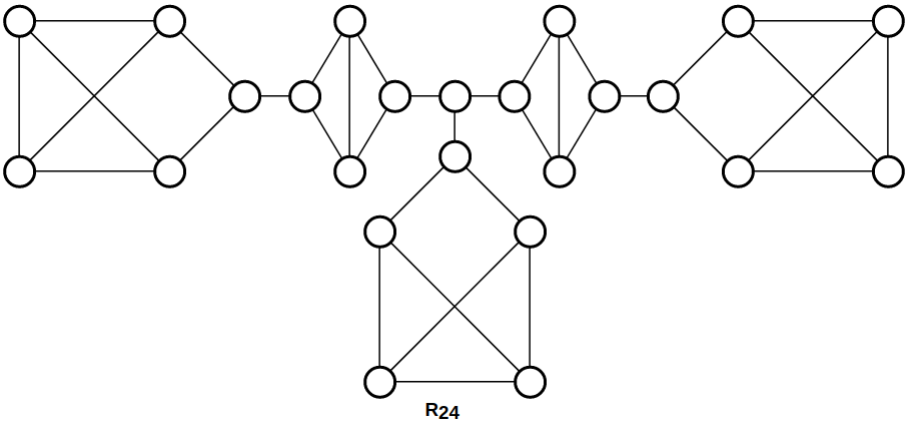
R_{20}



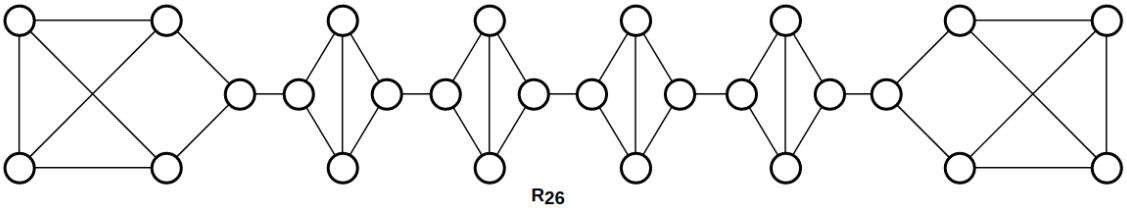
R_{22}



R_{24}



R_{24}



R_{26}

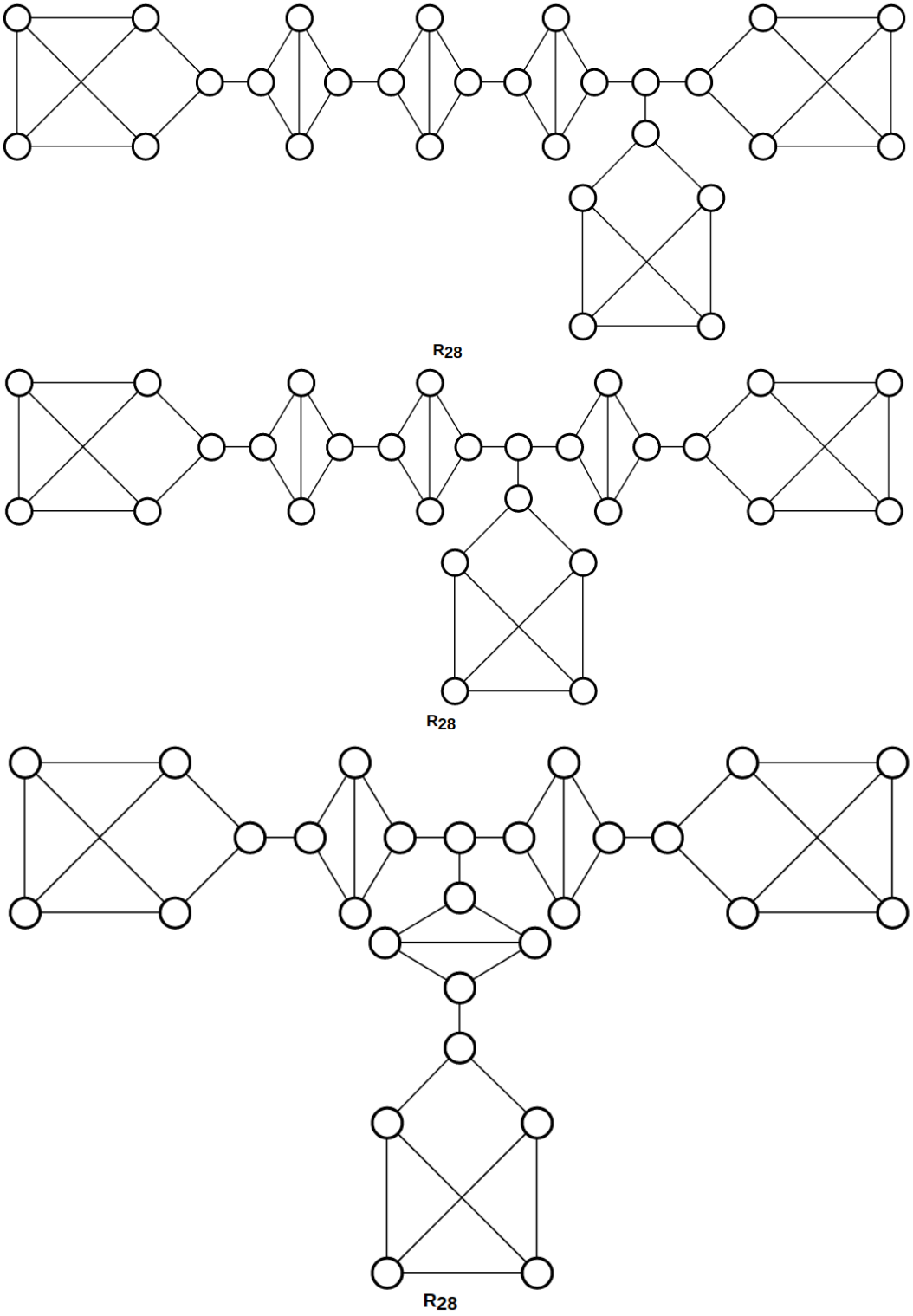
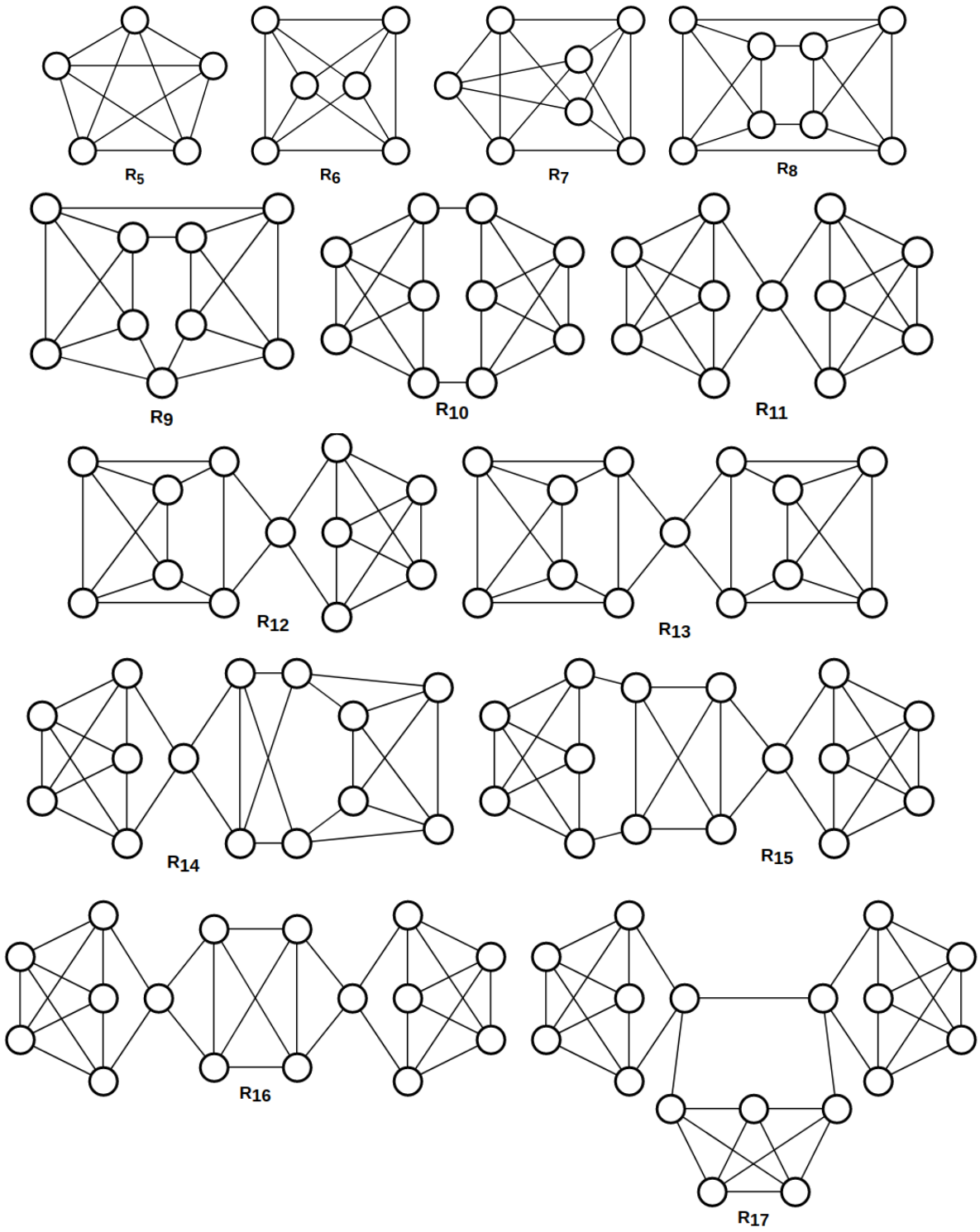


Figure 3.10: 3-regular graphs with minimal number of spanning trees on from 4 to 28 vertices

3.9 4-regular graphs with maximum number of spanning trees



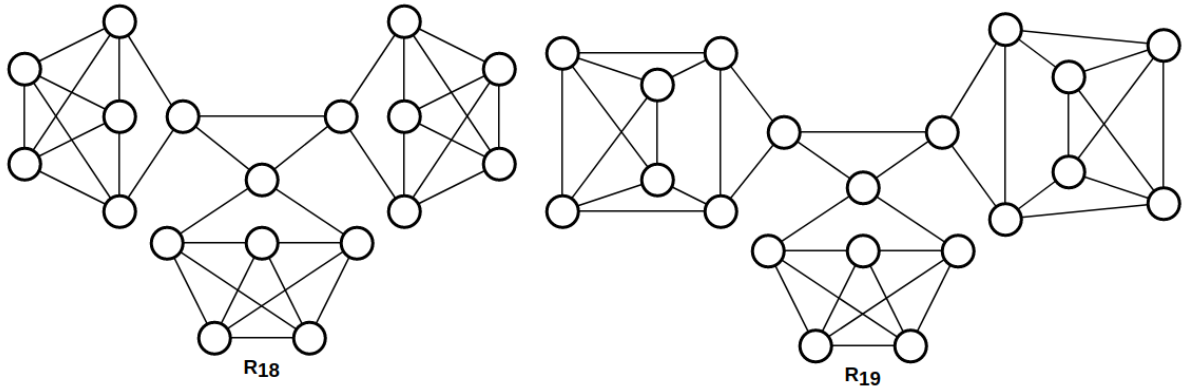


Figure 3.11: 3-regular graphs with minimal number of spanning trees on from 4 to 28 vertices

3.10 3-regular graphs with maximum number of spanning trees

Here is a list of 3-regular graphs which we expect to have the highest number of spanning trees for $4 \leq n \leq 42$. Cases with $n \leq 28$ were confirmed through generation of all 3-regular graphs on n vertices. For $n \geq 30$, we have only estimated data which we obtained by generating only graphs with maximum possible girth for the specific value of n.

For each presented graph, we list its n, number of spanning trees, girth and finally edge list.

n=4, 16 spanning trees, girth 3

[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]

n=6, 81 spanning trees, girth 4

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]

n=8, 392 spanning trees, girth 4

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 4), (2, 6), (3, 5), (3, 7), (4, 7), (5, 6), (6, 7)]

n=10, 2000 spanning trees, girth 5

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 6), (4, 8), (5, 7), (5, 9), (6, 9), (7, 8)]

n=12, 9800 spanning trees, girth 5

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 6), (4, 8), (5, 7), (5, 9), (6, 10), (7, 11), (8, 11), (9, 10), (10, 11)]

n=14, 50421 spanning trees, girth 6

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 10), (6, 12), (7, 11), (7, 13), (8, 10), (8, 13), (9, 11), (9, 12)]

n=16, 248832 spanning trees, girth 6

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 10), (6, 12), (7, 11), (7, 14), (8, 11), (8, 13), (9, 12), (9, 14), (10, 15), (13, 15), (14, 15)]

n=18, 1265625 spanning trees, girth 6

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 10), (6, 12), (7, 14), (7, 15), (8, 11), (8, 13), (9, 16), (9, 17), (10, 16), (11, 14), (12, 17), (13, 15), (14, 17), (15, 16)]

n=20, 6422000 spanning trees, girth 6

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 10), (6, 14), (7, 12), (7, 15), (8, 11), (8, 16), (9, 17), (9, 18), (10, 17), (11, 15), (12, 19), (13, 14), (13, 18), (14, 16), (15, 18), (16, 19), (17, 19)]

n=22, 32710656 spanning trees, girth 6

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 10), (6, 14), (7, 15), (7, 16), (8, 12), (8, 17), (9, 18), (9, 19), (10, 18), (11, 15), (11, 17), (12, 16), (13, 14), (13, 19), (14, 20), (15, 19), (16, 21), (17, 20), (18, 21), (20, 21)]

n=24, 168664320 spanning trees, girth 7

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 14), (10, 18), (11, 16), (11, 20), (12, 15), (12, 19), (13, 17), (13, 21), (14, 21), (15, 22), (16, 19), (17, 23), (18, 23), (20, 22), (22, 23)]

n=26, 862488000 spanning trees, girth 7

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 14), (10, 18), (11, 16), (11, 20), (12, 15), (12, 19), (13, 17), (13, 21), (14, 22), (15, 23), (16, 24), (17, 25), (18, 25), (19, 24), (20, 23), (21, 22), (22, 24), (23, 25)]

n=28, 4410450000 spanning trees, girth 7

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 14), (10, 18), (11, 16), (11, 20), (12, 15), (12, 19), (13, 17), (13, 21), (14, 22), (15, 23), (16, 24), (17, 25), (18, 25), (19, 24), (20, 23), (21, 22), (22, 26), (23, 27), (24, 26), (25, 27), (26, 27)]

n=30, 23066015625 spanning trees, girth 8

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 22), (10, 23), (11, 24), (11, 25), (12, 26), (12, 27), (13, 28), (13, 29), (14, 22), (14, 26), (15, 24), (15, 28), (16, 23), (16, 27), (17, 25), (17, 29), (18, 22), (18, 29), (19, 24), (19, 27), (20, 23), (20, 28), (21, 25), (21, 26)]

n=32, 116461210752 spanning trees, girth 8

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 14), (10, 22), (11, 18), (11, 23), (12, 24), (12, 25), (13, 26), (13, 27), (14, 28), (15, 19), (15, 24), (16, 23), (16, 26), (17, 29), (17, 30), (18, 29), (19, 27), (20, 22), (20, 31), (21, 26), (21, 28), (22, 30), (23, 31), (24, 31), (25, 28), (25, 29), (27, 30)]

n=34, 602763033600 spanning trees, girth 8

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 22), (10, 23), (11, 24), (11, 25), (12, 26), (12, 27), (13, 28), (13, 29), (14, 22), (14, 26), (15, 24), (15, 28), (16, 23), (16, 29), (17, 25), (17, 30), (18, 23), (18, 27), (19, 28), (19, 30), (20, 25), (20, 26), (21, 29), (21, 31), (22, 32), (24, 33), (27, 33), (30, 32), (31, 32), (31, 33)]

n=36, 3096828960300 spanning trees, girth 8

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 22), (10, 23), (11, 24), (11, 25), (12, 26), (12, 27), (13, 28), (13, 29), (14, 22), (14, 26), (15, 24), (15, 28), (16, 23), (16, 29), (17, 25), (17, 30), (18, 22), (18, 31), (19, 28), (19, 30), (20, 23), (20, 27), (21, 24), (21, 32), (25, 33), (26, 34), (27, 33), (29, 35), (30, 34), (31, 33), (31, 35), (32, 34), (32, 35)]

n=38, 16018229311425 spanning trees, girth 8

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 22), (10, 23), (11, 24), (11, 25), (12, 26), (12, 27), (13, 28), (13, 29), (14, 22), (14, 26), (15, 24), (15, 28), (16, 23), (16, 27), (17, 30), (17, 31), (18, 22), (18, 29), (19, 32), (19, 33), (20, 25), (20, 26), (21, 34), (21, 35), (23, 34), (24, 36), (25, 30), (27, 32), (28, 37), (29, 31), (30, 33), (31, 35), (32, 36), (33, 37), (34, 37), (35, 36)]

n=40, 83133130637040 spanning trees, girth 8

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 22), (10, 23), (11, 24), (11, 25), (12, 26), (12, 27), (13, 28), (13, 29), (14, 22), (14, 26), (15, 24), (15, 30), (16, 23), (16, 28), (17, 31), (17, 32), (18, 22), (18, 33), (19, 25), (19, 34), (20, 28), (20, 35), (21, 36), (21, 37), (23, 38), (24, 35), (25, 31), (26, 36), (27, 34), (27, 39), (29, 30), (29, 33), (30, 37), (31, 36), (32, 33), (32, 39), (34, 38), (35, 39), (37, 38)]

n=42, 432714276707327 spanning trees, girth 8

[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6), (2, 7), (3, 8), (3, 9), (4, 10), (4, 11), (5, 12), (5, 13), (6, 14), (6, 15), (7, 16), (7, 17), (8, 18), (8, 19), (9, 20), (9, 21), (10, 22), (10, 23), (11, 24), (11, 25), (12, 26), (12, 27), (13, 28), (13, 29), (14, 22), (14, 30), (15, 26), (15, 31), (16, 24), (16, 32), (17, 33), (17, 34), (18, 23), (18, 32), (19, 27), (19, 35), (20, 36), (20, 37), (21, 38), (21, 39), (22, 36), (23, 40), (24, 38), (25, 35), (25, 41), (26,

38), (27, 33), (28, 30), (28, 39), (29, 32), (29, 37), (30, 35), (31, 40), (31, 41), (33, 36), (34, 39), (34, 40), (37, 41)]

Bibliography

[1] WEST D. B. Introduction to Graph Theory. Prentice-Hall, Inc. 1996. 0-13-227828-6.

[2] CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C., Introduction to Algorithms, Third Edition. The MIT Press. 2009. 978-0-262-03384-8.

[3] STANOYEVITCH A., Discrete Structures with Contemporary Applications. Chapman and Hall/CRC Press. 2011. 978-1-4398-1768-1.

[4] MERINGER M., Fast Generation of Regular Graphs and Construction of Cages. Journal of Graph Theory 30, 137-146, 1999.

[5] AHO A., HOPCROFT J. and ULLMAN J., The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Co., Reading, MA, 1974, pp. 84-85.

[6] ALON N., The Number of Spanning Trees in Regular Graphs. Random Struct. Algorithms 1(2). 1990. 175-182.

[7] The On-line Encyclopedia of Integer Sequences, 2021, Connected regular graphs (with girth at least 3), From https://oeis.org/wiki/User:Jason_Kimberley/A068934/ 29.3.2023

[8] McKay B., Spanning Trees in Regular Graphs. Europ. J. Combinatorics (1983) 4. 1983. 149-160.

[9] Weisstein E. W., Graph Cycle. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GraphCycle.html 24.3.2023

[10] GODSIL C., ROYLE G., Algebraic Graph Theory. Springer Science+Business Media, LLC, 2001.

[11] BABAI L., Group, graphs, algorithms: the Graph Isomorphism Problem. Proceedings of the International Congress of Mathematicians (ICM 2018), Rio de Janeiro, Vol. 3 (3303-3320). 2018. 3303-3320.

[12] The On-line Encyclopedia of Integer Sequences, A000055 Number of trees with n unlabeled nodes., From https://oeis.org/A000055 25.3.2023

[13] CHAKRABORTY M., CHOWDHURY S., CHAKRABORTY J. et al., Algorithms for generating all possible spanning trees of a simple undirected connected graph: an extensive review. Complex Intell. Syst. 5, 265–281 (2019). 2019

[14] Onete CE and Onete MCC (2010) Enumerating all the spanning trees in an unoriented graph—a novel approach, XIth International Workshop on Symbolic and Numerical Methods, Modeling and Applications to Circuit Design (SM2ACD)

[15] Weisstein, Eric W. "Incidence Matrix." From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/IncidenceMatrix.html