On the Complexity of Boolean Networks

Author

Edgar Alberto Zúñiga Pérez

Supervisors

Dr. Edgardo Ugalde Saldaña
Dr. Jesús Urías Hermosillo

A thesis presented for the degree of
Master of Science in Physics

Universidad Autónoma de San Luis Potosí
Posgrado en Física
Facultad de Ciencias
San Luis Potosí, S.L.P., México
Septiembre, 2019

On the Complexity of Boolean Networks

Edgar Alberto Zúñiga Pérez

Abstract

The Boolean Networks are a model which has proven to be useful to model real-world systems. The Random Boolean Network model introduced by Kauffman in 1969 has been extensively used to model regulatory genetic networks and other types of systems. In this thesis, we propose the existence of a correlation between the complexity of a Boolean Network and the complexity of its constituents, i.e., the complexity of its topology and its set of updating functions. This hypothesis was tested by performing a series of experiments with the help of the implementation to approximate Kolmogorov complexity called Block Decomposition Method (BDM). First, we present a method to measure the complexity of the individual components of a Boolean Network and then, we propose a representation which can be used to measure the complexity of a Boolean Network. The results showed that this hypothesis was correct for Random Boolean Networks with small topologies given a sufficiently large set of Boolean Networks. However, it could not be generalized to larger topologies because of the enormous computational time required by the implementation of the BDM to approximate Kolmogorov complexity. Finally, the difficulties to measure the complexities of Random Boolean Networks with larger topologies inspired us to propose a novel method to measure Kolmogorov complexity. We have called this method the Block Decomposition Method with Neural Networks (BDMNN) and is based on the use of Neural Networks to perform a regression that approximates Kolmogorov complexity. These Neural Networks were trained by using random sequences for which its complexity was computed using the original BDM implementation to approximate Kolmogorov complexity. Our implementation was evaluated by performing some experiments with random sequences of bits. The results showed that our implementation is faster and requires less computational power to approximate Kolmogorov complexity than the original implementation. The only cost to be paid is a decrease in the accuracy of the results, however, we expect this error can be easily reduced with some little modifications to the method.

Dedication

I dedicate this thesis to all my teachers.

Acknowledgments

I want to thank to the Universidad Autónoma de San Luis Potosí, especially to the staff of its Physics Institute for their support and facilities to write this thesis. The knowledge that I acquired from my teachers in the graduate program was priceless and its dedication has paid off with this work. Finally, I also would like to thank the Consejo Nacional de Ciencia y Tecnología (CONACYT) for their financial support throughout my studies to get the master's degree.

Contents

List of Figures

List of Tables

This page intentionally left blank.

Chapter 1

Introduction

The Boolean Networks are a type of discrete dynamical system which can be used to model a lot of real-world networks. They have been used especially to model genetic regulatory networks by means of the Random Boolean Network model introduced by Kauffman in 1969. These models have proven to be worth to model not only genetic networks but also many other types of systems. It is expected that the ability of a Boolean Network to model any complex system, like a living system, must be intrinsically related to the complexity of any of its constituents (its topology and its set of updating functions). If its constituents are not complex enough, then the complexity of the Boolean Network also will not be enough. In other words, there should exist a correlation among the complexity of a Boolean Network and the complexity of its topology (or its set of updating functions).

In this thesis, we will try to prove the previous hypothesis of correlation. To do so, in Chapter 2, we will begin by giving a little review to graph theory since this is the mathematical tool which allows the description of the topology of any type of network. We will present some basic definitions and the terminology which will be used extensively in the subsequent chapters. A section will be devoted to the study of directed digraphs since they are used as the topology for Boolean Networks. Besides, some distributions of random graphs which will be used later will be studied.

Afterward, in Chapter 3, the model of a Boolean Network will be studied with an emphasis on the Random Boolean Network model proposed by Kauffman. We will study how these models are built from randomly choosing a topology and a set of updating functions. We will talk about the features of these constituents and the possible updating schemes which can be chosen for the dynamics of the network. Finally, we will review some real-world applications of this model and the software which can be used to study them.

In Chapter 4, a review of algorithmic complexity will be given. This chapter is the cornerstone of this thesis. Here, the definition of Kolmogorov complexity will be given. Then, we will study some approaches which have been used to approximate this complexity. Thereupon, we will review some theorems which have been used to create an implementation which can be used to estimate K-complexity. Following, some details about this implementation called Block Decomposition Method (BDM) will be given. This chapter finishes by giving some applications of Kolmogorov com-

plexity.

The second part of this thesis starts with Chapter 5. In this chapter, we will present the experiments performed to prove our hypothesis about the existence of a correlation between the complexity of a Boolean Network and the complexity of its constituents. Firstly, we establish the methods used to measure Kolmogorov complexity. This is done by performing some previous experiments with random sequences of bits and random graphs. Then, we move on to measure the complexity of random digraphs, i.e., the complexity of the topologies of a Boolean Network. We will check distinct types of representations and its effect in the complexity which is obtained. After, we will present a method to measure the complexity of a set of Boolean functions, the other constituent of a Boolean Network. Once we have learned how to measure the complexity of the individual constituents of a Boolean Network, we will continue by proposing a way to represent a Boolean Network as a sequence of bits. This representation is then used to measure the complexity of Boolean Networks and to test our hypothesis of correlation by performing some experiments with Random Boolean Networks.

The results of these experiments showed that the representations used were able to capture the features which give complexity to the mathematical objects which were studied. The hypothesis of correlation was proven to be correct for Random Boolean Networks with small topologies given an ensemble of random networks sufficiently large.

Unfortunately, these results could not be generalized to Boolean Networks with larger topologies because of the enormous computational time demanded by the implementation used to approximate Kolmogorov complexity. Inspired by this trouble, in Chapter 6, we propose a novel method to approximate Kolmogorov complexity by using Neural Networks. These Neural Networks were used to perform a regression which approximates Kolmogorov complexity. The Neural Networks were trained by using random sequences of bits which complexity was computed using the original BDM implementation. We have called this method the Block Decomposition Method with Neural Networks (BDMNN). Therefore, in this chapter, we will give a brief introduction to Machine Learning and Neural Networks. Then, the methodology and the details to create this implementation are presented. Finally, this implementation is evaluated by performing some experiments to compare its results with the results of the original implementation. The results showed that our method is faster and the only cost which must be paid is a little reduction in the accuracy of the approximation, nevertheless we expect the error can be easily reduced by enhancing the method through some little modifications which we talk about in the last part of this chapter.

The codes and algorithms used to perform the experiments of this thesis can be consulted in the Appendix section.

Chapter 2

Review of Graph Theory

In this chapter, a brief overview of graph theory will be given. This is not an extensive study of the whole subject, and only the basic definitions and terminology will be presented, especially those that will be useful in the next chapters. This chapter was written such that it could be used as a brief introduction to graph theory to the novices in the subject as well. If the reader has a basic background in graph theory, he can jump to the next chapter.

2.1 What is a Graph?

A graph[1] is defined as follows [1]:

Definition 2.1. A *graph* G with n *vertices* and m *edges* consists of a *vertex set* $V(G) = \{v_1, ..., v_n\}$ and an *edge set* $E(G) = \{e_1, ..., e_m\}$, where each edge is an unordered pair of vertices. We write uv for the edge $\{u, v\}$.

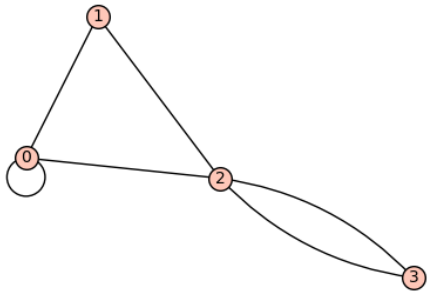
The terminology to refer to the vertices u and v which are the endpoints of an edge e is the following [2]:

Definition 2.2. If $uv \in E(G)$, then u and v are joined by the edge e and they are said to be *adjacent*. Furthermore, u and v are said to be incident with e, and e is said to be incident with u and v.

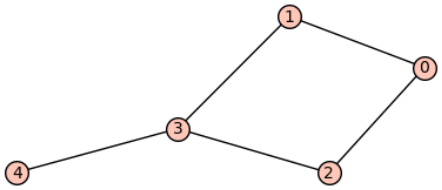
There are some special types of edges to be considered and that will help us to classify graphs [2]:

Definition 2.3. Two or more edges joining the same pair of vertices are called *multiple edges*, and an edge joining a vertex to itself is called a *loop*.

[1]Mathematicians prefer the term graph, although some physicists and engineers often refer to them simply as networks. In both cases, they are referring to the same type of object. We will use both terms indistinctly, though the reader should not confuse a Network with the Boolean Networks or Neural Networks which will be studied later.



(a) $G_1 = (V_1(G_1), E_1(G_1))$ where $V_1 = \{0, 1, 2, 3\}$ and $E_1 = \{00, 01, 02, 12, 23, 32\}$



(b) $G_2 = (V_2(G_2), E_2(G_2))$ where $V_2 = \{0, 1, 2, 3, 4\}$ and $E_2 = \{01, 02, 13, 23, 34\}$

Figure 2.1: (a) The graph G_1 is not a simple graph because it has a loop and two multiple edges, while in (b) the graph G_2 is a simple graph.

We can classify a graph like simple or not [3]:

Definition 2.4. A graph with no loops or multiple edges is called a *simple graph*.

Definition 2.5. A graph with loops or multiple edges is called a *multigraph* [2] [3].

It is customary to give a visual representation of a graph on paper, drawing a dot for each vertex and a curve or line for each edge joining its endpoints. The previous definitions are illustrated in Fig. 2.1.

The way in that the vertices and edges are drawn or labeled is not relevant, what matters is the information of which pairs of nodes[4] form an edge and which not [3]. Although two graphs may look different, they can be equal (see Fig. 2.2):

Definition 2.6. Two graphs are *equal* if they have equal vertex sets and equal edge sets. And two graph diagrams are *equal* if they represent equal vertex sets and equal edge sets.

A graph can be "contained" inside another graph. This "smaller" graph is called a subgraph which is defined as follows [5]:

Definition 2.7. A graph H is a *subgraph* of a graph G (written as $H \subseteq G$), if the vertex set of H is a subset of the vertex set of G and the edge set of H is a subset of the edge set of G. If H is a subgraph of G, it is said that G contains H.

[2]The term graph indicates the more general case of a multigraph, but often simple graphs are referred just as graphs [1].

[3]For some authors, the term multigraph refers to graphs with multiple edges or loops, while for others it refers only to graphs with multiple edges and without loops. See [4].

[4]Throughout this thesis the words vertex and nodes will be used indistinctly as synonyms.

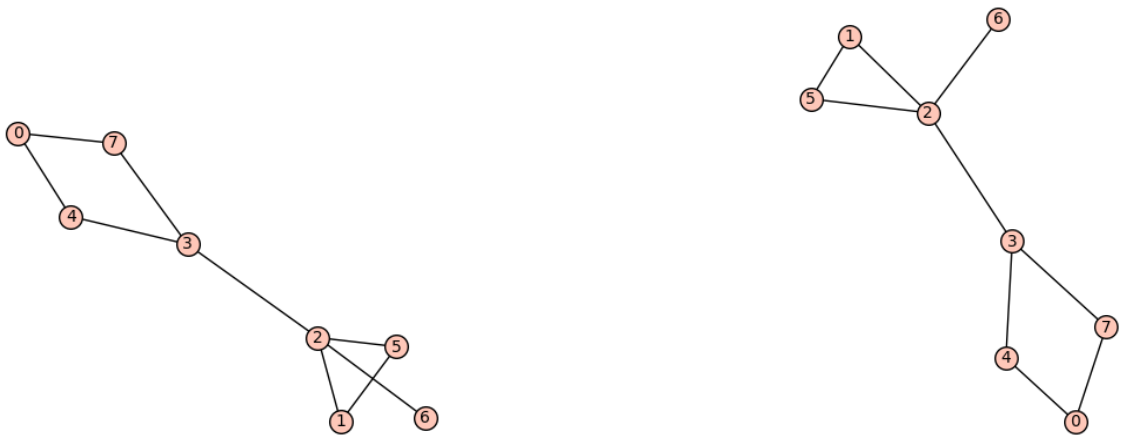
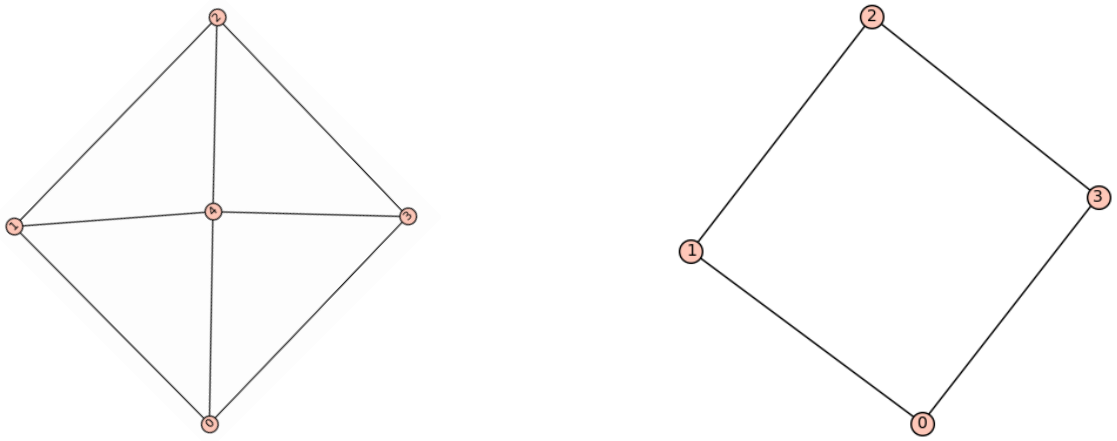


Figure 2.2: Two graph diagrams that look different but represent the same graph.

It must be noted that a subgraph is another graph and that every graph is a subgraph of itself [2] (see Fig. 2.3).



(a) $G = (V(G), E(G))$ where $V = \{0, 1, 2, 3, 4\}$ and $E = \{01, 03, 04, 12, 14, 23, 24, 34\}$

(b) $H = (V(H), E(H))$ where $V = \{0, 1, 2, 3\}$ and $E = \{01, 03, 12, 23\}$

Figure 2.3: (b) is a subgraph of (a).

2.2 The Vertex Degree

In this section, the basic definitions, used to characterize and describe the number of components of a graph, are given. Then, the handshaking lemma is presented.

2.2.1 Some Basic Definitions

Firstly, it is important to measure how big or small a graph is. The next definitions are helpful for this purpose [1]:

Definition 2.8. The *order* of a graph G, written as $n(G)$ or $|V(G)|$, is the number of vertices in G. An *n-vertex graph* is a graph of order n.

Definition 2.9. The *size* of a graph G, written as $e(G)$ or $|E(G)|$, is the number of edges in G.

When working with graphs it is important to know not only its order or size but also the "number of connections" of each node. This parameter is called the degree of the vertex, and it is defined as [1]:

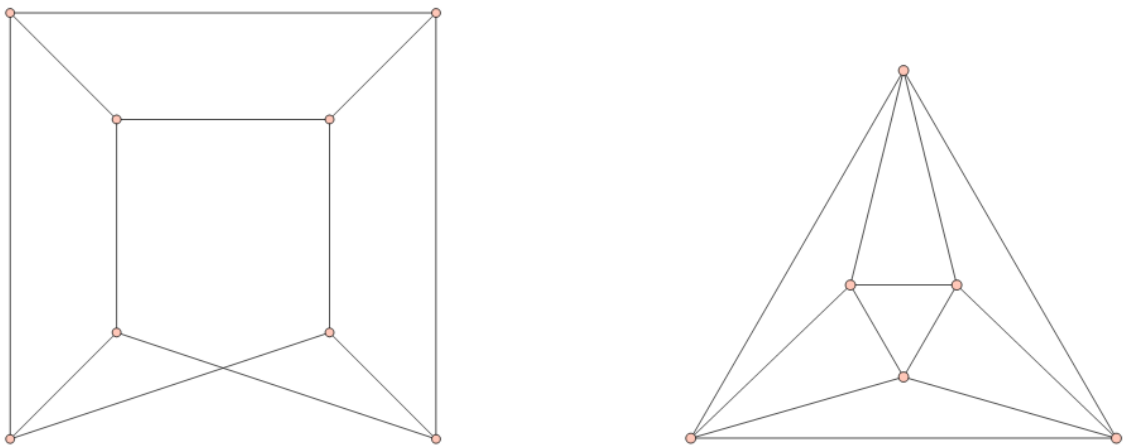
Definition 2.10. The *degree* of a vertex v in a Graph G, written $d_G(v)$ or $d(v)$, is the number of non-loop edges containing v plus twice the number of loops containing v.

In more simple words, the degree of a vertex is the number of lines connected to it and if the edge is a loop then we count the two connections to it even though they belong to the same edge.

A special name is given to the graphs which vertices have the same degree [3]:

Definition 2.11. If all the vertices of G have the same degree k, then G is *k-regular*, or simply *regular*.

Examples of regular graphs are shown in Fig. 2.4.



(a) A 3-regular graph. **(b)** A 4-regular graph.

Figure 2.4: Examples of regular graphs.

Unless we are working with regular graphs, the vertices do not need to have the same degree. In such cases, a minimum, and a maximum degree are defined as [3]:

Definition 2.12. The number $\delta(G) := min\{d(v)|v \in V\}$ is the *minimum degree* of G, the number $\Delta(G) := max\{d(v)|v \in V\}$ is the *maximum degree*.

If the graphs studied have many nodes, each one most likely having different vertex degree, then it is more convenient to describe the graph by specifying only the average degree of the nodes defined as [3].

Definition 2.13. The number $d(G) := \frac{1}{|V|}\sum_{v\in V} d(v)$ is the *average degree* of G. The average degree satisfies the relation $\delta(G) \leq d(G) \leq \Delta(G)$.

This number quantifies globally the number of edges of G per vertex, which is written as $\varepsilon(G) := \frac{|E|}{|V|}$, and satisfies $\varepsilon(G) = \frac{1}{2}d(G)$.

2.2.2 The Handshaking Lemma

Lemma 2.14. *[2]. In any graph G, the sum of all the vertex-degrees is equal to twice the number of edges. That is* $\sum d(v) = 2|E(G)|$.

Corollary 2.14.1. *[3]. The number of vertices of odd degree in a graph is always even.*

Corollary 2.14.2. *[2]. The sum of all the vertex-degrees is an even number.*

Corollary 2.14.3. *[2]. A graph which has n vertices and is r-regular has exactly* $\frac{1}{2}nr$ *edges.*

All the concepts presented in this section are summarized in an example in Fig. 2.5.

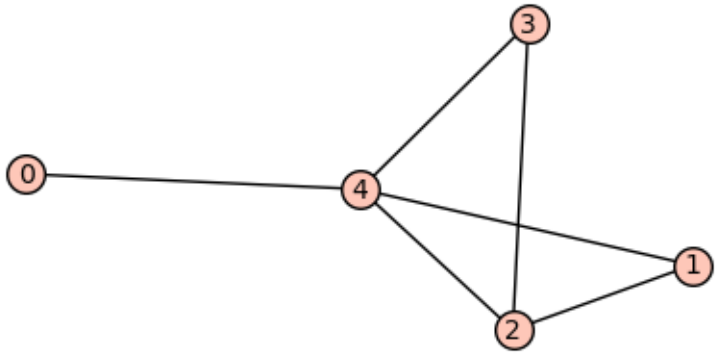


Figure 2.5: An example graph $G = (V, E)$ where $V = \{0, 1, 2, 3, 4\}$ and $E = \{04, 12, 14, 23, 24, 34\}$. The order of G is $n(G) = 5$. The size of G is $|E(G)| = 6$. The degrees of the vertices are: $d(0) = 1, d(1) = 2, d(2) = 3, d(3) = 2$ and $d(4) = 4$. The handshaking lemma is fulfilled because $1 + 2 + 3 + 2 + 4 = 2 * 6$; the number of vertices of odd degree is 2, so its corollary is also fulfilled. The minimum and maximum degrees of G are respectively: $\delta(G) = 1$ and $\Delta(G) = 4$. The average degree of G is $d(G) = 2.4$ while the number of edges per vertex is $\varepsilon(G) = 1.2$.

2.3 Adjacency and Incidence Matrices

There are many ways to specify or describe a graph. Until this section, for example, we have been specifying a graph by giving the list of vertices and the list of edges. Nevertheless, in some situations like when we need to generate random graphs or when we want to compute their properties, this can be cumbersome. In such situations, we can rely on more convenient representations using matrices. In this section, we will study the two most common representations of graphs using matrices, although, we must have in mind that they are not the only possible representations. We will continue this discussion in Section 2.10.

2.3.1 The Adjacency Matrix

The adjacency matrix of a graph is built by considering whether the pair of vertices are adjacent or not (see Definition 2.2). For a simple graph, the adjacency matrix is defined as follows [3]:

Definition 2.15. The *adjacency matrix* $A = (a_{ij})_{nxn}$ of a simple graph G is given by:

$$a_{ij} = \begin{cases} 1 & \text{if } ij \in E \\ 0 & otherwise \end{cases} \tag{2.1}$$

This definition is generalized for the case of a multigraph [2]:

Definition 2.16. The *adjacency matrix* $A = (a_{ij})_{nxn}$ of a multigraph G, is the matrix in which the entry a_{ij} is the number of edges joining the vertices i and j. The diagonal entries a_{ii} count the number of loops of the vertex i.

The adjacency matrix is symmetric. If it does not have any loops, then it has zeros on the diagonal and the sum of the elements of a column(row) equals the degree of the vertex that the column(row) represents. Furthermore, if the graph is simple, then all its entries are $0's$ or $1's$.

Finally, it must be remarked that a graph may have many adjacency matrices, depending on the way we label and order the vertices.

2.3.2 The Incidence Matrix

Instead of considering the adjacency of the vertices, we can specify a graph by saying if the pairs of vertices are incident or not (see Definition 2.2). The incidence matrix is defined as follows [3]:

Definition 2.17. The *incidence matrix* $B = (b_{ij})_{nxm}$ of a simple graph G with n vertices and m edges is given by:

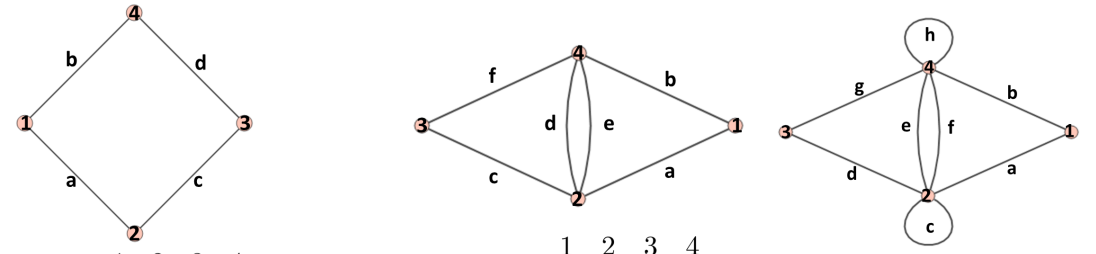
$$b_{ij} = \begin{cases} 1 & \text{if } v_i \in e_j \\ 0 & otherwise \end{cases} \tag{2.2}$$

If we wish to consider multigraphs with loops, then we will generalize this definition as:

Definition 2.18. The *incidence matrix* $B = (b_{ij})_{nxm}$ of a multigraph G with n vertices and m edges is given by:

$$b_{ij} = \begin{cases} 1 & \text{if } v_i \in e_j \\ 2 & \text{if } v_i \text{ is doubly incident with the loop } e_j \\ 0 & otherwise \end{cases} \tag{2.3}$$

As in the case of the adjacency matrix, the incidence matrix also depends on the way the edges and vertices are labeled. Some examples of the adjacency and incidence matrices of three different graphs are shown in Fig. 2.6.



$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \qquad B = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

(a)

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix} \end{array} \qquad B = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

(b)

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & 1 & 1 \end{pmatrix} \end{array} \qquad B = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccccc} a & b & c & d & e & f & g & h \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 2 \end{pmatrix} \end{array}$$

(c)

Figure 2.6: Three different graphs with their respective *adjacency matrix* A and *incidence matrix* B. (a) A simple Graph. (b) A multigraph without loops. (c) A multigraph with two loops.

2.4 Isomorphic Graphs

In Definition 2.6, we had a first approach to the equality of graphs. However, if we consider two graphs which only differ in the way their vertex sets have been labeled, then our definition is not able to capture that kind of "sameness" (see Fig. 2.7) [5]. For such situations, we ought to have a suitable definition that does not depend on the name of the vertices [3].

Definition 2.19. Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. We call G and G' *isomorphic*, and write $G \cong G'$, if there exists a bijection $f : V \to V'$ with $xy \in E \Leftrightarrow f(x)f(y) \in E'$ for all $x, y \in V$. Such a map f is called an *isomorphism*[5]; furthermore, if $G = G'$, it is called an *automorphism*. We do not normally distinguish between isomorphic graphs. Thus, we usually write $G = G'$ rather than $G \cong G'$.

Thus, we can say that an isomorphic graph G' can be obtained from G by relabeling its vertices in such a way that there is a one-to-one correspondence between the vertices of G and G' such that the number of edges joining each pair of vertices in G is preserved for the corresponding pair of vertices in G'. Therefore, an easy way to construct isomorphic graphs of a graph G, is just using permutation matrices to permute the rows and columns of its adjacency matrix $A(G)$ to get an adjacency matrix corresponding to a graph G' which only differs from G in the order we have labeled the vertices. Evidently, isomorphic graphs present transitivity, that is, if $G \cong G'$ and $G' \cong G''$ then $G \cong G''$.

The collection of isomorphisms f of a graph G which are also automorphisms (graph isomorphisms with themselves) form a group called the automorphism group

[5]The reader must not confuse the concepts of isomorphism and isomorphic. Isomorphism refers to a bijection f while isomorphic refers to an object G' which can be obtained by applying such bijection f to an object G.

and denoted as $Aut(G)$. This automorphism group preserves the incidence matrix of G, i.e., the incidence matrix is an invariant of this group [6].

2.5 Paths and Cycles

Firstly, we define what we understand when we talk about a *walk* [2]:

Definition 2.20. A *walk of length k* in a graph G is a succession of k edges of G of the form *ab, bc, cd, ..., ef*. We denote this walk by *abcd...ef* and refer to it as a *walk between a and f*. [6]

This definition allows the repetition of edges and vertices. A particular type of walk is a trail, and a particular case of a trail is a path. We define them as follows [2]:

Definition 2.21. If all the edges (but not necessarily all the vertices) of a walk are different, then the walk is called a *trial*. If, in addition, all the vertices are different, then the trail is called a *path*.

Now, we will define the case when the walk has the restriction that it starts and finishes at the same vertex [2]:

Definition 2.22. *A closed walk* in a graph G is a succession of edges of G of the form *ab, bc, cd, ..., ef,fa*. If all those edges are different, then the walk is called a *closed trail*. If, in addition, the vertices *a,b,c,d,...,e,f* are all different, then the trail is called a cycle.

In closed walks, the vertex of beginning can be anyone. An example of a walk is shown in Fig. 2.8.

Using the concept of path, we can define a new class of graph G, which have the particularity that any two of its vertices are *connected* by a path [2]:

Definition 2.23. A graph G is *connected* if there is a path in G between any given pair of vertices, and *disconnected* otherwise. Every disconnected graph can be split up into several connected subgraphs, called *components*.

In Fig. 2.9, examples of connected and disconnected graphs are shown.

2.6 Some Families of Graphs

There exist many families and ways to classify graphs based on different properties, like the number or degree of the vertices, symmetries, etc. Even there are graphs so interesting that they can have their own special name. Indeed, we have already found some families of graphs in definitions 2.11 and 2.23.

For the sake of completeness, in this section, a few of the most important families of graphs will be presented. Although we will have to leave out of discussion some

[6] Since in simple graphs the direction of the edges is not specified, we could refer to this walk as a *walk between f and a*.

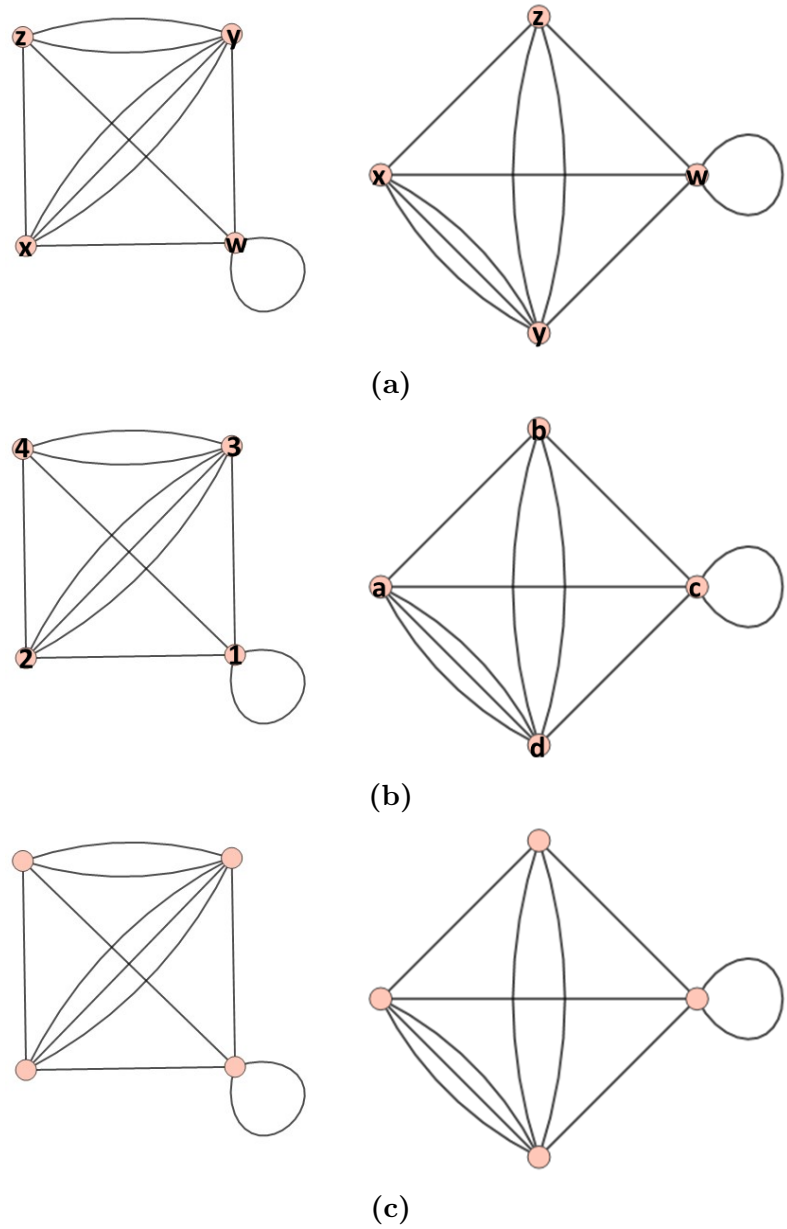


Figure 2.7: (a) Labeled graphs that are the same. (b) Labeled graphs that are not the same but are isomorphic. (c) Unlabeled graphs that are isomorphic (they are not labeled to highlight the fact that they are isomorphic independently of the labels chosen).

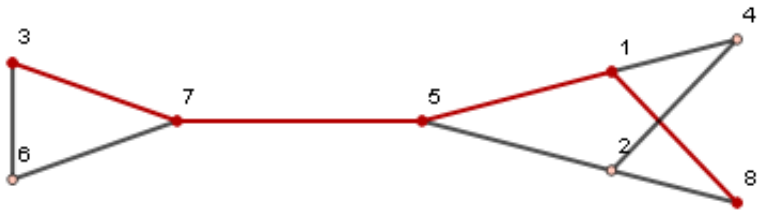


Figure 2.8: A walk of length 4 between vertices 3 and 8. All the edges are different, so we can consider this walk as a trail. Furthermore, all the vertices are different so we can consider it as a path.

(a) A connected graph. **(b)** A disconnected graph.

Figure 2.9: Examples of connected and disconnected graphs.

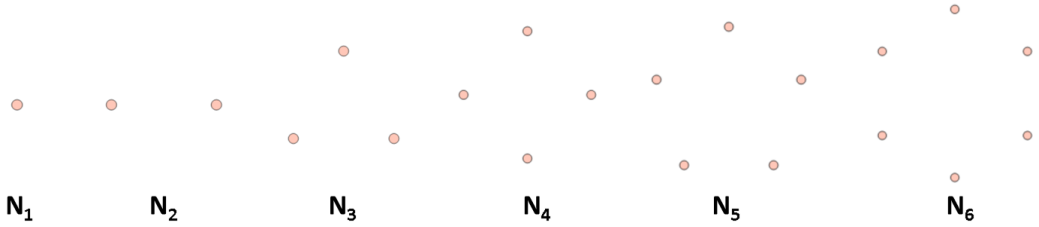


Figure 2.10: The first six null graphs.

families as the *Platonic graphs*, *the planar graphs* or the *Petersen graph*, a complete reference about the names and families of graphs can be found in the literature compiled by the Documentation Center of the Wolfram Language & System in [7] or in the references given at the end of this thesis.

2.6.1 Null Graphs

The *null graph* on n vertices, denoted as N_n, is the graph which has the vertex set $V = \{v_1, v_2, v_3, ..., v_n\}$ and no edges [5]. Obviously, N_n is zero regular. The first six null graphs are shown in Fig. 2.10.

2.6.2 Complete Graphs

A more interesting family is the family of complete graphs. A *complete graph* on n vertices, denoted as K_n, is the graph having vertex set $V = \{v_1, v_2, v_3, ..., v_n\}$ and all possible edges, i.e., every two distinct vertices are joined exactly by one edge [5]. The complete graph K_n is $(n-1)$-regular and has $\frac{1}{2}n(n-1)$ edges, following the handshaking lemma in 2.14. The first six complete graphs are shown in Fig. 2.11.

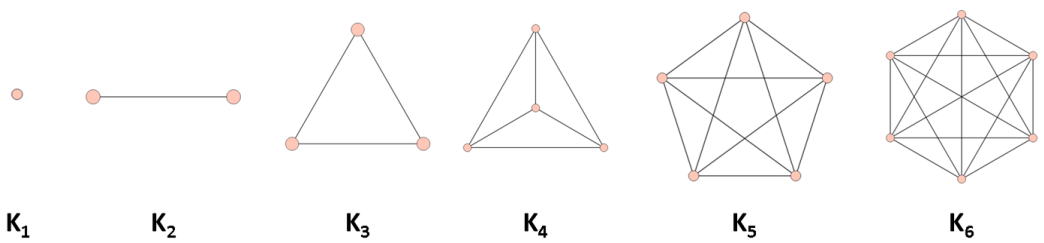


Figure 2.11: The first six complete graphs.

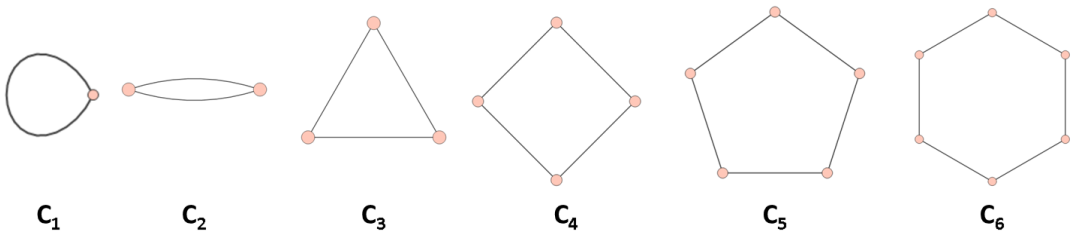


Figure 2.12: The first six cycle graphs.

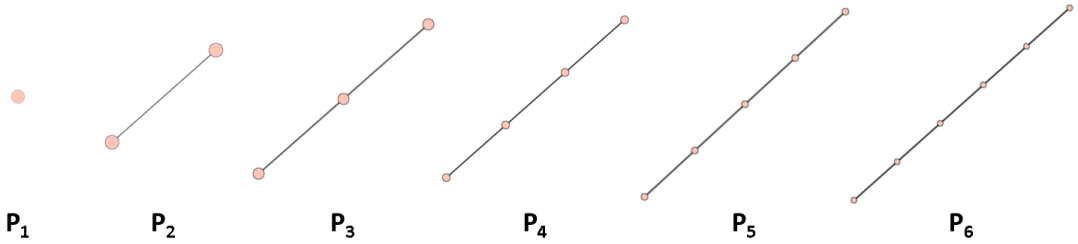


Figure 2.13: The first six path graphs.

2.6.3 Cycle Graphs

The *cycle graph* or *cyclic graph* on n vertices, denoted as C_n, is the graph consisting of a single cycle (see Definition 2.22). It has the vertex set $V = \{v_1, v_2, v_3, ..., v_n\}$ and the edge set $E = \{v_i v_{i+1} | i = 1, ...n-1\} \bigcup \{v_1 v_n\}$. [7] The cycle graph C_n has n edges and is 2-regular [2]. The first six cycle graphs are shown in Fig. 2.12.

2.6.4 Path Graphs

This family as the name indicates is the family which graphs consist of only a single path. A *path graph* on n vertices, denoted as P_n, is the graph which has the vertex set $V = \{v_1, v_2, v_3, ..., v_n\}$ arranged in a sequence such that the edge set is $E = \{v_i v_{i+1} | i = 1, ...n-1\}$. We can obtain the path graph P_n which has $n-1$ edges, from the cycle graph C_n by removing any edge [2]. The first six path graphs are shown in Fig. 2.13.

2.6.5 Bipartite Graphs

In this family, we found the graphs whose vertex set V admits a partition into two classes V_1 and V_2 such that every edge $uv \in E$ joins a vertex in V_1 to a vertex in V_2, i.e., $u \in V_1$ and $v \in V_2$. Two examples of bipartite graphs are shown in Fig. 2.14.

Furthermore, the *complete bipartite graph*, denoted as $K_{n,m}$, is the bipartite graph in which each vertex in V_1 is joined to each vertex in V_2 by exactly one edge, where $|V_1| = n$ and $|V_2| = m$. The special case $K_{1,m}$ is called a *star graph* [3].

The graph $K_{n,m}$ has $n+m$ vertices (where n are of degree m and m are of degree n) and nm edges. Finally, it must be noted that $K_{n,m} = K_{m,n}$, so we usually put the smaller number in the first index [2]. Some examples of complete bipartite graphs are shown in 2.15.

[7] For some authors, an additional requirement when defining the *cycle graph* on n vertices is that n must be equal or greater than *3*.

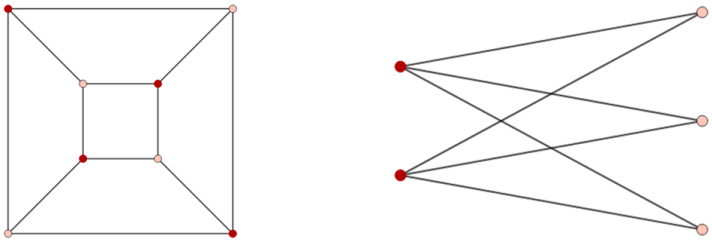


Figure 2.14: Two examples of bipartite graphs. As can be seen, the vertex sets of both graphs have been divided into two sets (of a different color) in such a way that they satisfy the definition of a bipartite graph.

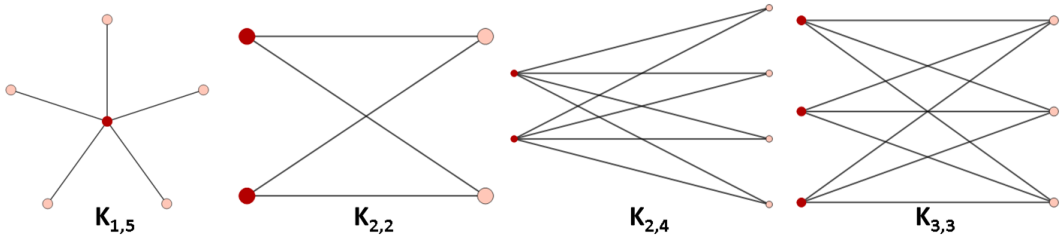


Figure 2.15: Examples of complete bipartite graphs. The first graph on the left is also a star graph.

2.6.6 Cube Graphs

The *n-cube graph*, also known as *n-hypercube graph* and denoted as Q_n, is the graph whose vertices are the 2^n possible binary words (sequences of $0's$ and $1's$) of length n of which two vertices are adjacent (are joined) if the two corresponding binary words are the same except for one element [8]. We note that Q_n has $n \cdot 2^{n-1}$ edges following the handshaking lemma 2.14. This type of graphs has applications in coding theory [2]. The first four cube graphs are shown in Fig. 2.16.

2.6.7 Trees

If a graph does not have any cycles or circuits (see Definition 2.22), it is called a *forest*. If a *forest* is connected (see Definition 2.23), then it is a *tree*, or in other words: A *tree* is a connected graph which is acyclic, and a forest is a graph whose components are trees, as shown in Fig. 2.17. This definition means that in a tree we found no multiple edges and that there is a unique arc or branch connecting any pair of vertices. Following the analogy, the vertices of degree 1 in a tree are called leaves [3]. A tree with n vertices has $n - 1$ edges [9].

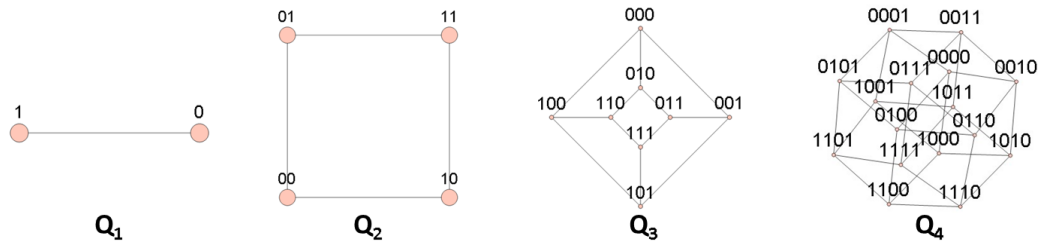


Figure 2.16: The first four cube graphs.

Figure 2.17: Two examples of tree graphs.

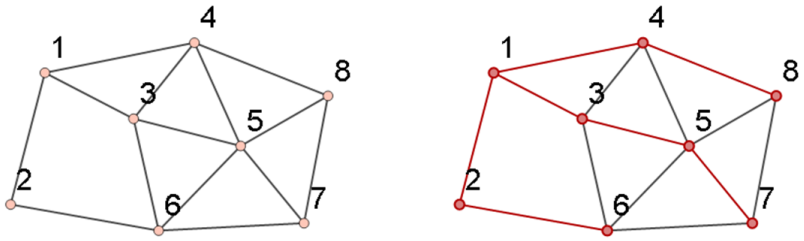


Figure 2.18: On the left, we can see a connected graph and on the right one of its subgraphs (highlighted) which is a spanning tree.

Let G be a connected graph, then we define a *spanning tree in* G as a subgraph of G which includes every vertex in G and which is also acyclic [2]. An example of a *spanning tree* is shown in Fig. 2.18.

Trees have many applications in computer science, especially in data storage and communication. For instance, these graphs are used in the Huffman coding, a type of coding that is used for lossless data compression and that is part of the algorithm used in the ZIP compression method [1]. Another application is the k-dimensional tree, a type of space-partitioning data structure for organizing information which then can be used to access data in a more efficient way in algorithms of search such as the algorithm of the k-nearest neighbors which is used in machine learning or the depth-first search (DFS) and the breadth-first search (BFS) which are used when a particular piece of information in the random-access memory (RAM) of a computer file is needed [2].

2.6.8 Circulant Graphs

These graphs are defined as follows [10]:

Definition 2.24. A *circulant graph* is a graph of n vertices in which the ith graph vertex is adjacent to the $(i+j)$th and $(i-j)$th graph vertices for each j in a list of jumps $\{j_1, j_2, ...\}$. The circulant graph $Ci_n(1, 2, ..., \lfloor n/2 \rfloor)$ is equal to the complete graph K_n and the graph $Ci_n(1)$ gives the cyclic graph C_n.

Some examples of these types of graphs are shown in Fig. 2.19.

2.7 Counting Graphs

A natural question arises when generating graphs with certain given properties: how many different graphs of a certain kind there exist? For instance, we could ask: How many different graphs of a given number of vertices there exist? We would find that

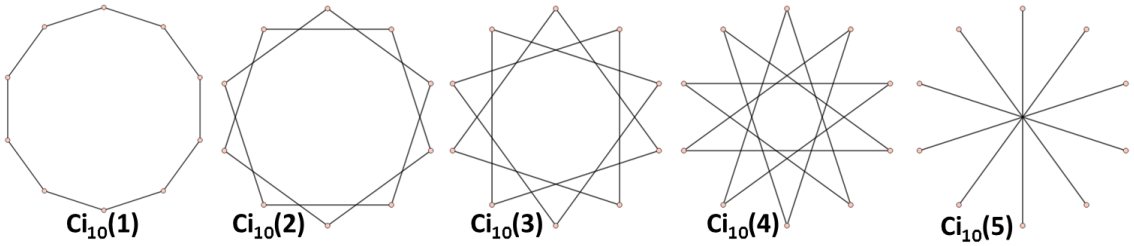


Figure 2.19: Circulant graphs of 10 vertices and only one jump. The jumps are $1, 2, 3, 4$, and 5 respectively.

in general, the number of possible graphs with some given characteristics increases fast with the number of vertices and that in certain situations the problem to find this number is yet open [2]. To illustrate it, let us see what happens with two of the simplest cases.

2.7.1 Labeled Graphs

We will consider labeled simple graphs which are not isomorphic, so we say that we are counting up to isomorphism. The problem is to find the number of this kind of graphs given the number n of vertices. If we first take into account the $(n-1)$-regular graph of n vertices which has all the possible edges for the type of graphs we are considering, thus by means of the corollary 2.14.3 it has $\frac{1}{2}n(n-1)$ edges. The rest of the graphs will have or not each of these edges (there are two possibilities), then the number of possible graphs is $2^{\frac{1}{2}n(n-1)}$. For instance, for $n = 4$ the number of labeled graphs is 64, while for $n = 7$ the number increases up to 2097152.

2.7.2 Unlabeled Graphs

Here, we treat with non-isomorphic unlabeled graphs. This problem is not as easy to solve as it was for labeled graphs. Usually, we will have to invoke formulas (if they exist) as the general formula found by George Pólya in 1935 to count graphs with any number of vertices and edges [2]. As a reference we can see the Table 2.1, which also contains the number of unlabeled connected graphs and unlabeled regular graphs up to 8 vertices, and presents the same behavior stated previously, the number of graphs increases fast with the number of vertices n.

As stated before, there exist formulas like the general formula of George Pólya to count other types of graphs. An example is the Caley's Theorem which states that the number of labeled trees with n vertices is n^{n-2}. However, this is not always the case and sometimes the problem remains unsolved for both labeled and unlabeled graphs or just for one of these cases. In general counting problems for labeled graphs are much easier to solve [2].

2.7.3 Counting Adjacency Matrices

We can also try to count the number of possible adjacency matrices of a given graph. First, we have to remember that we can permute the labels of a graph without changing its structure, that is, we could permute the corresponding rows

n	1	2	3	4	5	6	7	8
graphs	1	2	4	11	34	156	1044	12346
connected graphs	1	1	2	6	21	112	853	11117
regular graphs	1	2	2	4	3	8	6	20

Table 2.1: Number of unlabeled graphs for $n \leq 8$.

and columns of the adjacency matrix in order to achieve a permutation of the labels of the graph and get a matrix which still refers to the same graph.
Let G be a graph of order $n(G)$ with an automorphism group of order $|Aut(G)|$, then the number of distinct adjacency matrices N_A is given by $N_A = \frac{n(G)!}{|Aut(G)|}$ where $n(G)!$ is the number of permutations of vertex labels [11].

2.8 Directed Graphs

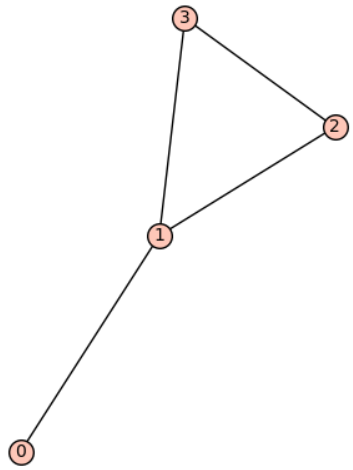
Up to this point, in our notation for edges, we have made no distinction between uv and vu, i.e., an edge is defined only as a pair of vertices but not as an ordered pair of vertices. Graphically our usual notation for uv or vu means that we are connecting the vertices u and v using a line with no preferential direction. However, in some applications, we would like to give a direction to each edge in a graph, or graphically, we would like to draw each edge as an arrow starting in a vertex and finishing in the other. In fact, this kind of graphs will be used extensively throughout this thesis because they will allow us to define a Boolean Network in the next chapter, thus we define a *directed graph* also called *digraph* [1]:

Definition 2.25. A *directed graph* or *digraph* D consists of a *vertex set* $V(D)$ and an *edge set* $E(D)$, where each edge is an ordered pair of vertices. We write uv for the edge (u, v), with u being the *tail* and v being the *head*. We write $u \to v$ when $uv \in E(D)$ meaning "there is an edge from u to v".

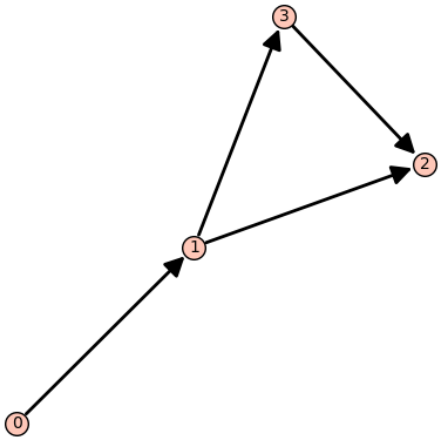
The difference between a graph and a digraph is shown in Fig. 2.20.
In the same way, following Definition 2.4, a directed graph can be simple or not. In the most general case, a directed graph which is not simple (called *directed multigraph* or *multidigraph* [8]) can have multiple edges between two vertices. These multiple edges can be *parallel* if they have the same direction or can be a *loop* if they start and finish in the same vertex [3].

We will continue giving some basic definitions for digraphs. They usually are only extensions of the corresponding definitions for graphs; therefore, we could define again all the concepts given for graphs, but we shall only present the most important for our needs.

[8] In the next chapters, we will refer to simple and not simple digraphs just as digraphs unless we need to make a distinction.



(a) Edges treated as unordered pairs of vertices



(b) Edges treated as ordered pairs of vertices

Figure 2.20: Difference between a graph and a digraph.

2.8.1 The Underlying Graph

As we can see in Fig. 2.20, we can obtain a graph from a digraph [2].

Definition 2.26. Let D be a digraph. The *underlying graph* of D is the graph G obtained by replacing each directed edge by the corresponding undirected edge. We say that D is an orientation of G called an *oriented graph*. In the same way we can obtain an *oriented graph* D from a graph G by choosing an orientation for each edge.

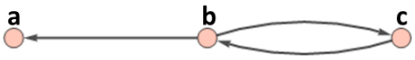
2.8.2 Subdigraphs

We can also construct the subdigraphs of a digraph D [2]:

Definition 2.27. Let D be a digraph with vertex-set $V(D)$ and edge set $E(D)$. A *subdigraph* of D is a digraph all whose vertices belong to $V(D)$ and all whose directed edges belong to $E(D)$.

An example of a digraph and a subdigraph is shown in Fig. 2.21.

(a) $D = (V(D), E(D))$ where $V = \{a, b, c, d\}$ and $E = \{ba, bc, cb, cd, dc\}$



(b) $C = (V(C), E(C))$ where $V = \{a, b, c\}$ and $E = \{ba, bc, cb\}$

Figure 2.21: (b) is a subdigraph of (a).

2.8.3 Local Degrees in a Digraph

In Definition 2.10, we saw that we can regard the degree of a vertex as the number of lines or edges connected to it, counting by two if the corresponding edge is a

loop. Due to the nature of digraphs, we can consider two distinct types of degrees depending in if the edge is going in or going out of the vertex, i.e., we have two local degrees [1]:

Definition 2.28. Let v be a vertex in a directed graph. The *vertex out-degree* $d^+(v)$ of v is the number of outgoing directed edges from it, that is, the number of edges with tail v. The *vertex in-degree* $d^-(v)$ of v is the number of incoming directed edges from it, that is, the number of edges with head v.

For instance, for the digraph in Fig. 2.20b, the in and out degrees of the vertices are: $\{d^-(0) = 0, d^-(1) = 1, d^-(2) = 2, d^-(3) = 1\}$ and $\{d^+(0) = 1, d^+(1) = 2, d^+(2) = 0, d^+(3) = 1\}$. It must be noted that if a vertex in a digraph has a loop, this will contribute by one to the count of both degrees, the *vertex out-degree* and the *vertex in-degree*.

2.8.4 Adjacency and Incidence Matrices for a Digraph

We will define the adjacency matrix for the most general type of digraph, that is a digraph that can have multiple edges and loops (a multidigraph) but we will continue referring to them just as digraphs. We define [2]:

Definition 2.29. Let D be a digraph with n vertices labeled $1, 2, 3, ..., n$. The *adjacency matrix A(D)* is the $n \times n$ matrix in which the entry in row i and column j is the number of directed edges going from vertex i to vertex j.

If we sum the row(column) elements of *A(D)* for a given row(column) we will obtain the out-degree(in-degree) of the corresponding vertex. This property is very useful, and in fact, we will make use of it in the next chapters to compute random digraphs with a given in-degree.

Let v and w be vertices of a digraph. If v and w are joined by a directed edge, then v and w are said to be *adjacent*. If the directed edge is directed from v to w, then the arc is said to be *incident from* v and *incident to* w. Then, the incidence matrix of a digraph is defined as follows [2]:

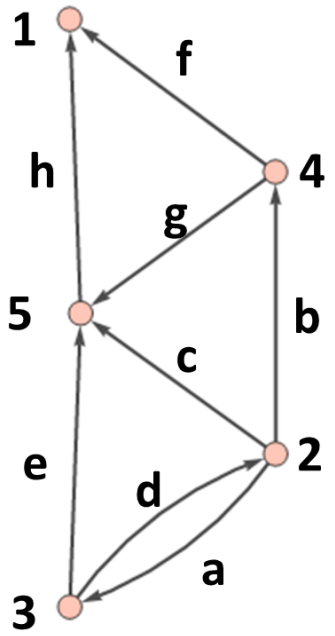
Definition 2.30. Let D be a digraph with n vertices and m directed edges. The *incidence matrix B(D)* is the nXm matrix in which the entry in the row b_{ij} is given by:

$$b_{ij} = \begin{cases} 1 & \text{if the directed edge j is incident from vertex i} \\ -1 & \text{if the directed edge j is incident to vertex i} \\ 0 & \textit{otherwise} \end{cases} \tag{2.4}$$

An example of the adjacency and incidence matrices for a digraph is shown in Fig. 2.22. It can be noted, that the adjacency and incidence matrices of a digraph depend in the way we have labeled the vertices. Just as we did for graphs, we can use this fact to find isomorphic digraphs which we define next.

2.8.5 Isomorphic Digraphs

Another important concept we must extend for digraphs is the concept of isomorphism:



$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$B = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccccc} a & b & c & d & e & f & g & h \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 \\ 1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 0 & -1 & 1 \end{pmatrix} \end{array}$$

Figure 2.22: An example digraph and its adjacency and incidence matrices labeled as A and B respectively.

Definition 2.31. Two digraphs C and D are *isomorphic* if D can be obtained from C by relabeling the vertices- that is if there is a one-to-one correspondence between the vertices of C and those of D, such that the number of edges joining any pair of vertices in C is equal to the number of edges joining the corresponding pair of vertices (in the same direction) in D.

Thus, we can ignore the labels of the vertices of digraphs in some applications, e.g., if we are checking if two digraphs are isomorphic or not, we can consider the corresponding unlabeled graphs, because we can relabel them as necessary (see Fig. 2.23).

As was said in Section 2.4, we can easily construct isomorphic digraphs in an analogous way to the one proposed for graphs by using permutations matrices to permute the rows and columns of the adjacency matrix in such a way that we are changing the order the vertices are labeled (see Fig. 2.24).

2.8.6 Counting Digraphs

As was said in Section 2.7.1, there are $2^{\frac{1}{2}n(n-1)}$ simple labeled graphs with vertices $v_1, ..., v_n$. In the case of digraphs, there exist 2^{n^2} with these same vertices such that each ordered pair of them appears at most once as an edge. If we consider digraphs without loops and admit only one of the two possibilities $u \to v$ or $v \to u$ for each of them, then there are $3^{\frac{1}{2}n(n-1)}$ different digraphs of this type [1].

2.9 Random Graphs

Sometimes, for some reasons, we will need to randomly create graphs that are different, but such that they all share some features in common. These features can be

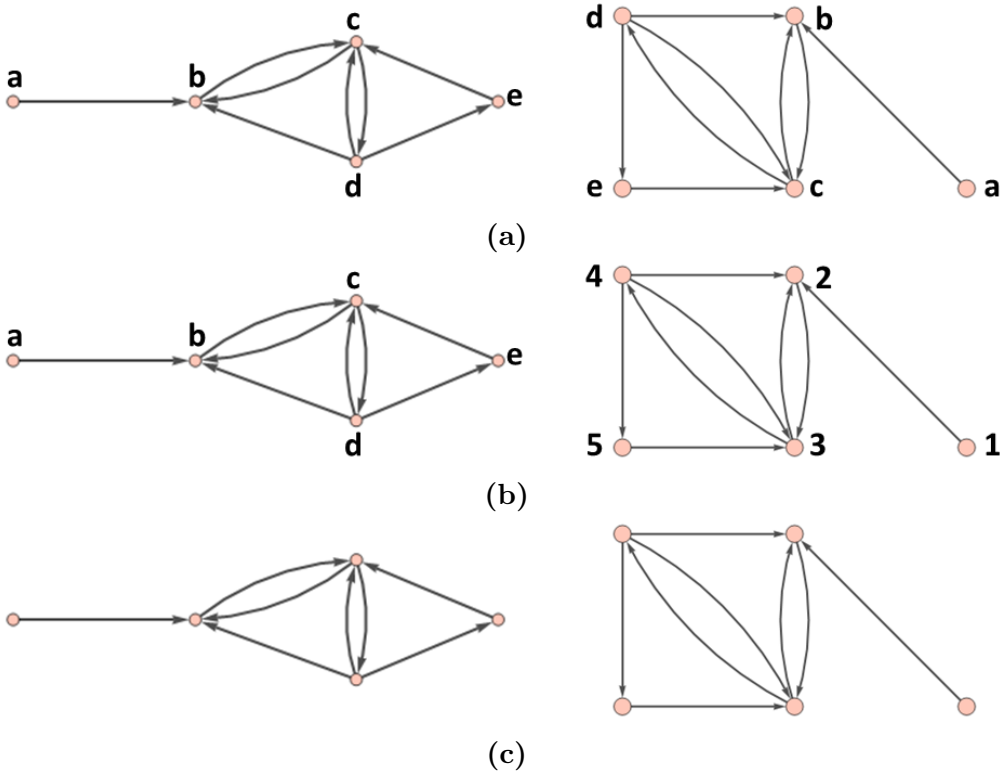


Figure 2.23: (a) Labeled digraphs that are the same. (b) Labeled digraphs that are not the same but are isomorphic. (c) Unlabeled digraphs that are isomorphic.

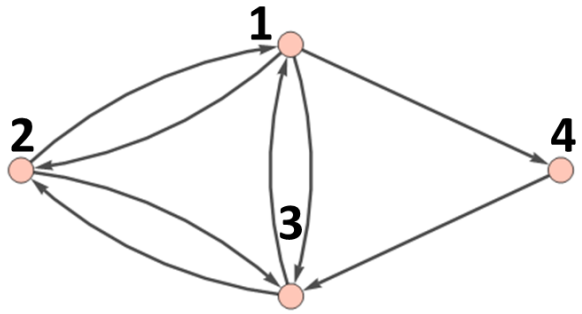
the number of vertices, edges, connections, etc. Thereby, we define a random graph as follows [12]:

Definition 2.32. A *Random Graph* $G(n,p)$ is a probability space of all labeled graphs on n vertices $\{1,...,n\}$, where for each pair $1 \leq i \leq j \leq n$, ij is an edge of $G(n,p)$ with probability $p = p(n)$, independently of any other edges. Equivalently, the probability of a graph $G = (V,E)$ with $V = \{,1,...,n\}$ in $G(n,p)$ is $Pr[G] = p^{|E(G)|}(1-p)^{\binom{n}{2}-|E(G)|}$.

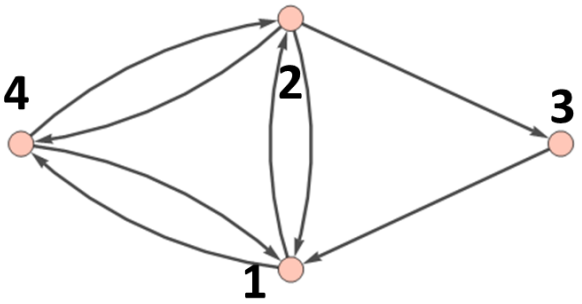
Once the edge distribution is chosen, a random graph is obtained by picking out one graph within this distribution. There have been proposed different edge distributions to attack different problems, following we will present some of them which we will use afterward, though strictly speaking they should be named pseudo-random graphs (see [12] for a broader discussion).

2.9.1 The Watts-Strogatz Small-World Graph Distribution

This graph distribution is composed of graphs which are built using the following random procedure. We start with a circulant graph of n vertices and jump list $\{1,2,...,k\}$. Then, we introduce disorder to this graph by rewiring each edge with a probability p, making sure that no loop or multiples edge is created. Evidently, p satisfies $0 \leq p \leq 1$. For $p = 0$ we have graphs which show regularity and order, meanwhile as we increase the value of p we increase the disorder until we reach $p = 1$ where the graph edges are totally disordered [13]. In Fig. 2.25 three



$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$



$$A' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 2.24: A digraph which vertices have been labeled in two different orders (they are isomorphic) and their corresponding adjacency matrices. The adjacency matrix A' is obtained from A by permuting the rows and columns of A following the rules: $1 \mapsto 2$, $2 \mapsto 4$, $3 \mapsto 1$ and $4 \mapsto 3$

.

random graphs generated using this distribution by means of the built-in function *WattsStrogatzGraphDistribution* of Mathematica are shown.

2.9.2 The Barabási-Albert Graph Distribution

The graphs generated by this distribution are useful since they can describe networks of complex and possibly unknown topology such as the topology of web pages where nodes are individual web pages and the edges are the hyperlinks or the peer-reviewed scientific literature where the nodes are the publications and the edges are citations [14]. These graphs are also known as scale-free networks and their construction is based on preferential attachment since they expand by adding vertices which attach preferentially to high degree vertices. They have the property that the probability of a given vertex v of having a vertex degree $d(v)$ decays as a power-law $P(d(v)) \sim d(v)^{-\gamma}$ where $\gamma > 0$. The preferential connectivity is achieved with the following procedure [15]. Starting from a graph of $(n - m)$ vertices we add at each step a vertex with k edges. These new k edges are randomly attached to vertices at random with the probability of linking a node v_i given by [14]:

$$P(linking\,to\,node\,v_i) \sim \frac{d(v_i)}{\sum_j d(v_j)} \tag{2.5}$$

This procedure is performed m times, i.e., until we have a graph with n vertices. In Fig. 2.26 three random graphs generated using this distribution by means of the built-in function *BarabasiAlbertGraphDistribution* of Mathematica are shown.

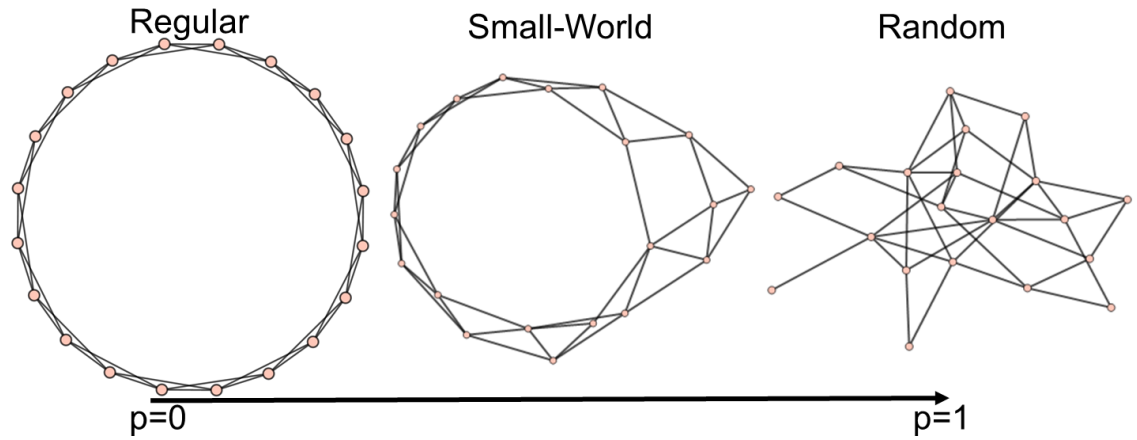


Figure 2.25: Three random graphs of 20 nodes generated by using the Watts-Strogatz graph distribution with increasing p value. The starting graph chosen was the circulant graph $Ci_n(1, 2, 3, 4,)$, i.e., the complete graph K_4 as can be seen for $p = 0$. The graphs like the middle one with regular lattices and small characteristic path lengths are known as "small-world" networks [13].

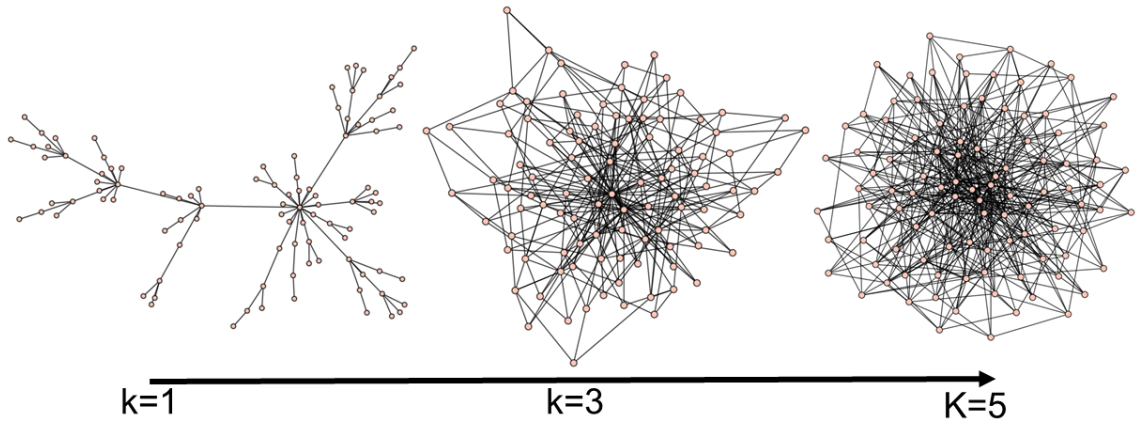


Figure 2.26: Three random graphs of 100 nodes generated by using the Barabási-Albert graph distribution. They were constructed starting from a cycle graph with 3 vertices. At each step was added a vertex with $1, 3$ and 5 edges respectively.

2.9.3 The Uniform Graph Distribution

This is perhaps the simplest graph distribution. The sample space is made up just of all the simple graphs of n vertices and k edges. This graph distribution is implemented in Mathematica by means of the built-in function *UniformGraphDistribution*. Alternatively, the sample space can consider all the directed graphs of n vertices and k directed edges, or even we can be more specific and consider all the directed graphs of n nodes with vertex out-degree d^+ and vertex in-degree d^-.

2.10 Other Ways of Representing Graphs

Throughout this chapter, we have used diverse ways to represent a graph. For instance, we began describing a graph using lists, then we moved to matrices, even we could consider the graphical representations as another type of representation.

To finish this chapter, it must be mentioned that in addition to the ones used here, there are many other ways to encode or represent a graph. For instance, we could continue using lists, but instead of giving the list of vertices or edges of the graph, we could give a nested list with the list of vertices adjacent to each vertex in the graph, this is known as the adjacency list. Another option could be to give a list with the vertex degree of each vertex in the graph. Or we could use other types of matrix representations, for example, we could use the *Laplacian Matrix* which we did not define here.

It must be remarked, that each representation, encodes or is focused on a feature or information of the graph, so when we work with graphs, we have to use the representation that best fits our needs for a particular application. The representation used will have to allow us to perform calculations without the loss of the information we care about. Often, the adjacency matrix will satisfy our requirements, but other times it will not.

Chapter 3

Boolean Networks

This chapter will be devoted to an introduction of Random Boolean Networks. First, we will begin by defining what a Boolean Network is, then we will present the Random Boolean Network Model. We will talk about the topology of these networks and the updating functions used. We will continue by discussing the dynamics of Random Boolean Networks and we will present an example. Afterward, we will briefly review the different dynamical phases and other possible updating schemes. We will finish by commenting on some applications and the software available to simulate Random Boolean Networks.

The model we are interested in was originally proposed by Stuart Kauffman in 1969 [16] [17]. He proposed a mathematical model to simulate genetic regulatory networks in which each gene was represented by a node which can only take the two values 0 or 1, i.e., every gene could only be in an "on" state or in an "off" state. There are N nodes in the network, and the state of each node is controlled by the information it receives from k randomly[1] chosen nodes (itself included) which are connected to it. Once the connections are established in a random way, they stay fixed during the dynamics of the network. Each node has associated with it one updating function which is also chosen in a random way from the set of all possible Boolean functions of k inputs and stays fixed during the dynamics as well. This logic function establishes the next state of the node in every time step according to the states of its input nodes.

3.1 Boolean Networks and The Random Boolean Network Model

3.1.1 Boolean Networks

Formally we can define a Boolean Network as follows [18]:

Definition 3.1. A *Boolean Network*, is composed by a set of N nodes, $\sigma_1, \sigma_2, ..., \sigma_N$. At each time t ($t = 0, 1, 2, ...$), a node has only one of two different values: 1 or 0. The state of each node is controlled by a set of k nodes (possibly including itself)

[1] It is supposed that we are using some probability distribution which can be for example a normal distribution.

in a synchronous manner. Thus, the network can be described by a set of equations as follows:

$$\begin{cases} \sigma_1(t+1) = f_1(\sigma_{n_{1,1}}(t), \sigma_{n_{1,2}}(t), ..., \sigma_{n_{1,k}}(t)) \\ \sigma_2(t+1) = f_2(\sigma_{n_{2,1}}(t), \sigma_{n_{2,2}}(t), ..., \sigma_{n_{2,k}}(t)) \\ \quad \vdots \\ \sigma_N(t+1) = f_N(\sigma_{n_{N,1}}(t), \sigma_{n_{N,2}}(t), ..., \sigma_{n_{N,k}}(t)) \end{cases} \tag{3.1}$$

where $f_i (i = 1, 2, 3, ..., N)$ is a k-bit function with $k \leq N$, and $n_{i,k}$ is the label for the k^{th} node which acts as input for the i^{th} node.

Hence, the set of equations 3.1 can be written in a short form as [19]:

$$\sigma_i(t+1) = f_i(\sigma_{n_{i,1}}(t), \sigma_{n_{i,2}}(t), ..., \sigma_{n_{i,k}}(t)) \tag{3.2}$$

We can consider this system of equations as a discrete dynamical system starting from the input state $(\sigma_1(t=0), \sigma_2(t=0), ..., \sigma_N(t=0))^T$ which is evaluated synchronously [18], though this is not the only possible updating scheme and certainly is not the more general or closest to the updating schemes of real-world genetic regulatory networks. In this thesis, we will only work with Boolean Networks with a synchronously updating scheme.

3.1.2 Random Boolean Networks

In the random Boolean Network model originally proposed by Kauffman, also known as $N-k$ model or Kauffman network, the functions f_i and the connections between nodes are chosen randomly following some probability distribution. We will refer to a Boolean Network which functions and links between the nodes have been chosen randomly as a *Random Boolean Network* or *RBN* for short.

A Random Boolean Network is entirely described by its topology, i.e., the connections between the nodes, and the dynamical rules, i.e., the functions f_i, together, they give the dynamics of the network [20]. We will continue discussing these Boolean Networks features with a little more detail.

3.1.3 Topology

When we say topology of the RBN, we mean the nodes and the links between them. We represent the topology of an RBN by means of a directed graph (see Section 2.8). Unless we mention something else, the number k of input links for each node is chosen to be the same for all of them[2]. Or in other words, the vertex in-degree $d^-(i)$ has the same value k for each node (i is the index of the node). Although the vertex in-degree is fixed for all the nodes by the parameter k, the vertex out-degree $d^+(i)$ is not fixed and every node can have a different value of output directed edges[3].

Once the number N of nodes is known, the links between these nodes are chosen randomly for each node. Again, unless we say something else, we will consider that we are using a uniform probability distribution which means that it is equally probable to link a node with all the other nodes including itself. If we consider the

[2]In the special case where $N = k$, the RBNs are also known as random maps [21].

[3]It has to be clear that indeed we are dealing with multidigraphs.

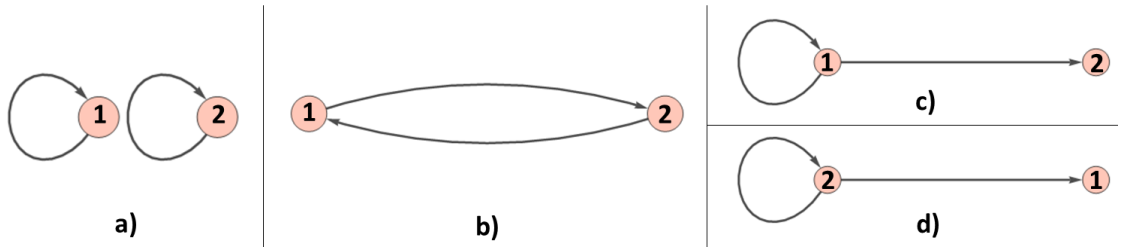


Figure 3.1: Ensemble of possible topologies for $N = 2$ and $k = 1$. The topologies a) and b) have the same statistical weight 1/4. The topologies c) and d) are the same (we consider unlabeled digraphs) so its probability is 1/2. If we would have assigned updating functions to the nodes, though the topologies of c) and d) are the same their updating functions would not have been necessarily the same.

possible topologies as unlabeled digraphs, then when generating an ensemble of digraphs with a sufficient amount of elements, eventually all of the possible topologies will appear, although their probabilities of apparition will be different, i.e., in general, the statistical weights are different for each possible topology [20]. Nonetheless, we must be aware that once we have assigned an updating function to a node, we have labeled it, see Fig. 3.1.

The topologies generated by this model are known as *homogeneous random topologies* because in the thermodynamic limit $N \to \infty$, the number of outgoing links, i.e., the vertex out-degree d^+ of each node follows a Poisson distribution [20]:

$$P_{out}(d^+) = \frac{k^{d^+}}{d^+!} e^{-k} \tag{3.3}$$

This means that every element of the network is statistically equivalent to any other, thus, we can characterize the topology of the network by the average connectivity k, as we can expect all the nodes to have a connectivity value close to this value [22]. Nonetheless, the reader must have in mind that the probability distribution of Eq. 3.3 is not the only possible type of distribution which can be used to generate the topology. In fact, we could use any other probability distribution to generate the topology of the network. For instance, one of the most famous topologies is the so-called *scale-free topology*[4] which considers the habitual feature of complex networks of having few elements with many links and many elements with few links [21]. This type of behavior is present in molecular networks, genetic networks, social networks, etc. and is generated by the following probability distribution [22]:

$$P_{out}(d^+) = [\zeta(\gamma)(d^+)^\gamma]^{-1} \tag{3.4}$$

Where $\gamma > 1$ and $\zeta(\gamma) = \sum_{j=1}^{\infty} j^{-\gamma}$ is the Riemann Zeta function. This time the parameter which will characterize the topology will be the scale-free exponent γ [22]. Throughout this work, we will not consider Boolean Networks with scale-free topologies or topologies with a probability distribution different to the probability distribution given by Eq. 3.3.

[4]Compare with the scale-free graphs described in Section 2.9.2.

σ_1 σ_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 3.1: Truth table with all the possible Boolean functions for $k = 2$.

3.1.4 Updating Functions

3.1.4.1 Boolean Functions

The two possible states of each node in an RBN are Boolean values, i.e., $\sigma_i \in \{0, 1\}$. These states are updated by means of an updating function, i.e., a logic function, Boolean function or Boolean expression which output is another Boolean value which depends in the k Boolean variables used as inputs:

$$f : \{0, 1\}^k \rightarrow \{0, 1\} \tag{3.5}$$

Thus, an alternative definition to Eq. 3.1 and 3.2 for a Boolean Network and which highlights it as a function is as follows [18]:

Definition 3.2 (Alternative Definition)**.** A Boolean network of N nodes can be seen as a Boolean function:

$$f : \{0, 1\}^N \rightarrow \{0, 1\}^N \tag{3.6}$$

where $f = \{f_1, f_2, ..., f_N\}^T$ and each f_i is of the form shown in Eq. 3.1.

Now, we must know how to represent the functions f_i. The easiest way to describe a Boolean function is through a truth table which is ([23], page 26):

Definition 3.3 (Truth Table)**.** A list of all possible input states to a Boolean function, listed in ascending binary order, and the output response for each input combination.

It can be easily seen that there are 2^k possible input states and therefore there are 2^{2^k} possible Boolean functions for a given k. For instance, for $k = 2$ there are $2^2 = 4$ possible input states and $2^{2^2} = 16$ possible logic functions. The truth table for $k = 2$ is shown in Table 3.1.

Some Boolean functions receive a special name and symbol which was inherited from their use in digital electronics where they are better known as gates. We will review some of these special logic functions since they are the Lego bricks to build more complex Boolean functions.

The first of these functions is the NOT function which is usually represented as $!\sigma$, $\neg\sigma$, $\bar{\sigma}$ or $\sim \sigma$ [24]. The truth table of this logic function is shown in Table 3.2.

σ	$!\sigma$
0	1
1	0

Table 3.2: Truth table for the NOT function.

σ_1 σ_2	$\sigma_1 \wedge \sigma_2$
0 0	0
0 1	0
1 0	0
1 1	1

Table 3.3: Truth table for the AND function.

The next function is the AND function, which is usually represented as $\sigma_1 \wedge \sigma_2$, $\sigma_1 \cdot \sigma_2$, $\sigma_1.\sigma_2$, $\sigma_1\sigma_2$, $\sigma_1\&\sigma_2$ or $\sigma_1\&\&\sigma_2$ [24]. The truth table of this logic function is shown in Table 3.3.

Finally, we have the OR function, which is usually represented as $\sigma_1 \vee \sigma_2$, $\sigma_1 + \sigma_2$, $\sigma_1|\sigma_2$ or $\sigma_1 \parallel \sigma_2$ [24]. The truth table of this logic function is shown in Table 3.4.

σ_1 σ_2	$\sigma_1 \vee \sigma_2$
0 0	0
0 1	1
1 0	1
1 1	1

Table 3.4: Truth table for the OR function.

The Boolean functions AND, OR and NOT are the Lego bricks from which any other Boolean function can be built ([23], page 34). Indeed, some other special gates are derived from these three gates: the NAND, NOR, XNOR, and XOR functions.

The NAND (contraction of NOT AND) function inverts the output of an AND function. It is usually represented as $\sigma_1 \bar{\wedge} \sigma_2$, or $\overline{\sigma_1 \cdot \sigma_2}$ [24]. The truth table of this logic function is shown in Table 3.5.

The NOR (contraction of NOT OR) function inverts the output of an OR function. It is usually represented as $\sigma_1 \bar{\vee} \sigma_2$, $\sigma_1 \downarrow \sigma_2$, or $\overline{\sigma_1 + \sigma_2}$ [24]. The truth table of this logic function is shown in Table 3.6.

The XOR (contraction of exclusive OR) is usually represented as $\sigma_1 \underline{\vee} \sigma_2$, or $\sigma_1 \oplus \sigma_2$ [24]. The truth table of this logic function is shown in Table 3.7.

The XNOR (contraction of exclusive NOR) is usually represented as $\overline{\sigma_1 \oplus \sigma_2}$. The truth table of this logic function is shown in Table 3.8 [25].

Using these derived functions, and the symbols we have introduced, we can rewrite the truth table with all the possible Boolean functions for $k = 2$ as in Table 3.9.

The reader must had noticed up to this point that logic functions can be written as different compositions of distinct functions, for instance, $((\sigma_1\&\&!\sigma_2)||(!\sigma_1\&\&\sigma_2))$

σ_1 σ_2	$\sigma_1 \bar{\wedge} \sigma_2$
0 0	1
0 1	1
1 0	1
1 1	0

Table 3.5: Truth table for the NAND function.

σ_1 σ_2	$\sigma_1 \bar{\wedge} \sigma_2$
0 0	1
0 1	0
1 0	0
1 1	0

Table 3.6: Truth table for the NOR function.

and $((!\sigma_1||!\sigma_2)\&\&(\sigma_1||\sigma_2))$ are equivalent ways of representing the function $\sigma_1 \oplus \sigma_2$, since all of them share the same truth table. One question arises: is there any preferred combination of functions when trying to represent a Boolean function besides its Boolean table? The answer will depend on the context, in general, the preferred representation will rely on our needs. However, we must know that between the derived functions we talked about above two of them highlight: the NAND and the NOR functions. These functions are known as universal logic gates since any other logic function can be implemented using only combinations of one of these functions [26]. We will use this property in chapter 5 to represent Boolean functions in a systematic way, so we define:

Definition 3.4 (Universal Boolean Functions)**.** The NAND and NOR functions are universal Boolean functions since any other Boolean function can be represented just by combinations of one of them.

Therefore, we only need to use repeatedly one of these functions to build any other logic function.

3.1.4.2 Choosing the Updating Functions

There are diverse ways to choose the set of functions for a Boolean Network, i.e., the function assigned to each node. Some techniques try to give biological meaning to this assignment (see [20]). For example, one usual technique which tries to resemble the behavior of neural networks is that in which only threshold functions are used. Thus, the state of each node is given by [20]:

$$\sigma_i(t+1) = \begin{cases} 1 & \text{if } \sum_{j=1}^{N} (c_{ij}(2\sigma_j + 1) + h) \geq 0 \\ 0 & \text{else} \end{cases} \tag{3.7}$$

Where $c_{ij} = \pm 1$ with equal probability if the node i receives an input from node j, otherwise $c_{ij} = 0$.

Nevertheless, in this work, we will use standard logic functions and we will choose the Boolean functions of each node in the following way: as was said before, there

σ_1 σ_2	$\sigma_1 \oplus \sigma_2$
0 0	0
0 1	1
1 0	1
1 1	0

Table 3.7: Truth table for the XOR function.

σ_1 σ_2	$\overline{\sigma_1 \oplus \sigma_2}$
0 0	1
0 1	0
1 0	0
1 1	1

Table 3.8: Truth table for the XNOR function.

are 2^{2^k} possible Boolean functions for a given k value, thus we will choose with a uniform probability distribution[5] a function from the set of possible Boolean functions for each node, this means that the probability of choosing each possible Boolean function is $1/(2^{2^k})$, e.g., for $k = 2$ all the 16 possible functions have the same probability of 1/16 of being chosen in each node.

In the case where we assign the same function to all the nodes, we have what is known as a cellular automaton or finite state machine with random wiring. Thereby, a Boolean Network is a more general model [20].

A generalization to the original Kauffman model introduces a bias parameter p when the Boolean functions are chosen [27]. This parameter gives the probability of the values in the truth table of being 1 given a uniformly distributed input, and thereby the probability of being 0 is $(1 - p)$. Therefore, the probability of choosing a function with n times the output value 1 and $(M - n)$ times[6] the output value 0 is given by $p^n(1 - p)^{M-n}$. Therefore, the method which we will use in this work to choose the logic functions is the special case with $p = 1/2$, which means that all the possible functions have the same probability of being chosen [20].

3.1.5 Counting RBNs

Since the RBNs are generated in a random way it is useful to give us an idea of how many different RBN is possible to create given the number of nodes N and the parameter k of each digraph. First, we must remember that there are 2^{2^k} possible logic functions for a given k. Thus, for a digraph with N nodes we can assign to each node one of these possible Boolean functions which gives [18]:

$$\underbrace{2^{2^k} \times 2^{2^k} \times \cdots \times 2^{2^k}}_{\text{N Times}} = 2^{N \times 2^k} \tag{3.8}$$

[5] More specifically it is a discrete uniform distribution.

[6] Where $M = 2^k$, i.e., the number of possible input states.

Function	Symbol
f_0	0
f_1	$\sigma_1 \wedge \sigma_2$
f_2	$\sigma_1 \wedge !\sigma_2$
f_3	σ_1
f_4	$!\sigma_1 \wedge !\sigma_2$
f_5	σ_2
f_6	$\sigma_1 \oplus \sigma_2$
f_7	$\sigma_1 \vee \sigma_2$
f_8	$\sigma_1 \bar{\wedge} \sigma_2$
f_9	$\overline{\sigma_1 \oplus \sigma_2}$
f_{10}	$!\sigma_2$
f_{11}	$\sigma_1 \vee !\sigma_2$
f_{12}	$!\sigma_1$
f_{13}	$!\sigma_1 \vee \sigma_2$
f_{14}	$\sigma_1 \bar{\wedge} \sigma_2$
f_{15}	1

Table 3.9: Truth table with all the possible Boolean functions for $k = 2$ rewritten by using symbols.

This is the number of possible RBNs once we have fixed the topology of the digraph. This number increases so fast, e.g., for $N = 4$ and $k = 2$ the number of possible RBNs for a given topology is $65,536$ and for $N = 6$ and $k = 2$ this number is $16,777,216$.

Nevertheless, we do not only assign the Boolean functions randomly, but we also wire the nodes of the digraph randomly, thus we must consider these combinations. If every node has the same vertex in-degree $d^-(i) = k$, then each node has $N!/(N-k)!$ possible ordered combinations for k different links [20]. Consequently, combining this result with Eq. 3.8 the number of possible RBNs generated randomly is:

$$(\frac{2^{2^k} N!}{(N-k)!})^N \tag{3.9}$$

This number is even bigger than our initial estimation, e.g., for $N = 4$ and $k = 2$ the number of possible RBNs for a given topology is $1,358,954,496$ and for $N = 6$ and $k = 2$ this number has an order of magnitude of 10^{16}. Hence, the universe of possible RBNs is immense and moreover there exists a very high variance in the statistical studies within this enormous set. However, it is possible to focus on representative features and extract some general properties [20].

3.1.6 Dynamics: The State Space and Some Basic Definitions

As was mentioned before, throughout this work, we will only consider the synchronous updating scheme, which means that the state of all the nodes is updated at the same time in a parallel way. Every node can have two possible states, thus the number of possible states for an RBN of N nodes is 2^N. Consequently, the

number of possible initial states is also 2^N. We will usually study all the possible initial states of every RBN since the dynamics of a Boolean Network is in general different depending on the initial state we have chosen. The configuration state of a Boolean Network at time t can be represented by the vector [19]:

$$\sigma(t) \equiv \Sigma_{t+1} = \{\sigma_1(t), \sigma_2(t), ..., \sigma_N(t)\} \tag{3.10}$$

With this notation the dynamics of the system can be further abbreviated as [22]:

$$\Sigma_{t+1} = \mathcal{F}[\Sigma_t] \tag{3.11}$$

Where $\mathcal{F}$ represents the action of the updating functions $\{f_1, f_2, ..., f_N\}$ on the configuration state Σ_t.

To visualize the dynamics of an RBN we represent its possible states in a 2^N-dimensional state space where we assign to each state a point. If a state flows to another state according to the updating functions, we use a directed edge to join both states with the head of the directed edge pointing to the newer state. Therefore, the state space can be represented by means of a directed graph[7] of 2^N nodes where the vertex out-degree $d^+(i)$ of each node is 1 (since every state has a unique successor state[8]). On the other hand, the vertex in-degree $d^-(i)$ of each node is not fixed which means many states can drive to the same state.

The trajectory in state space has some features. First, because the state space is finite and the dynamics is deterministic, when starting from any initial state, eventually a state or a sequence of states will be repeated, which means we have reached an attractor [21]:

Definition 3.5. A periodic sequence of states (which we also call cycle) is an *attractor* if there are states outside the attractor that lead to it [20].

Definition 3.6. If an attractor consists of only one state it is called a *point attractor* or *steady state*, but if it consists of two or more states, it is called a *cycle attractor* or *state cycle* [27].

Thus, a point attractor can be visualized as a loop in the digraph representation of the state space. Some other interesting definitions are the following:

Definition 3.7. The *size* or *length of an attractor* is the number of different states on the attractor. [20].

Definition 3.8. The *basin of attraction* or *attractor basin* of an attractor is the set of all states that eventually end up on this attractor, including the attractor states themselves. The *size of the basin of attraction* is the number of states belonging to it.

Those states which are not reachable beginning from any other state are called garden-of-Eden states:

[7]The reader must not confuse the digraph which represents the topology of the Boolean Network and the digraph which represents the state space.

[8]It is said that the RBNs are dissipative systems [21].

Definition 3.9. States without predecessors are called *garden-of-Eden states* [21].

The dynamics flows from these garden-of-Eden states and converges in an attractor:

Definition 3.10. The time it takes to reach an attractor is called *transient time* [21].

Now, it is time to see a Boolean Network in action. We will study a network of $N = 4$ nodes with parameter $k = 2$. The truth table for the functions in each node is shown in Table 3.10. The digraph used in our example is shown in Fig. 3.2. Now, once we have established the topology and the dynamical rules, our Boolean Network is entirely described, and its state space can be seen in 3.3. Note that, to label the nodes in the state space diagram we have replaced the states $\{0000, 0001, \cdots, 1111\}$ by their corresponding decimal representation $\{0, 1, \cdots, 15\}$. This convention allows us to abbreviate and make more understandable our diagram, especially when dealing with large state spaces.

σ_1 σ_2	f_{Node1}	f_{Node2}	f_{Node3}	f_{Node4}
0 0	1	1	0	1
0 1	1	1	1	1
1 0	0	0	1	0
1 1	0	1	0	0

Table 3.10: Truth table with the functions assigned to a Boolean Network of 4 nodes and parameter $k = 2$. These functions were chosen with a uniform probability from the set of all the possible Boolean functions of 2 variables.

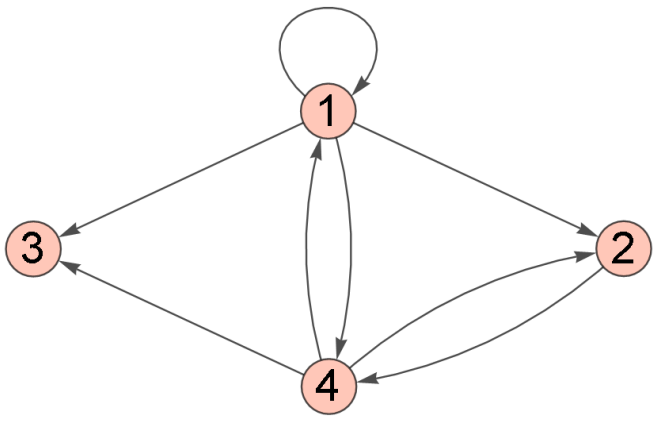


Figure 3.2: Topology of a Boolean Network of 4 nodes and vertex in-degree 2. This topology was randomly chosen with the help of a uniform distribution among all the possible topologies with the given parameters.

We finish this section by commenting that the study of the mean number of attractors of the RBNs is of interest, though that discussion is out of the purposes of this work.

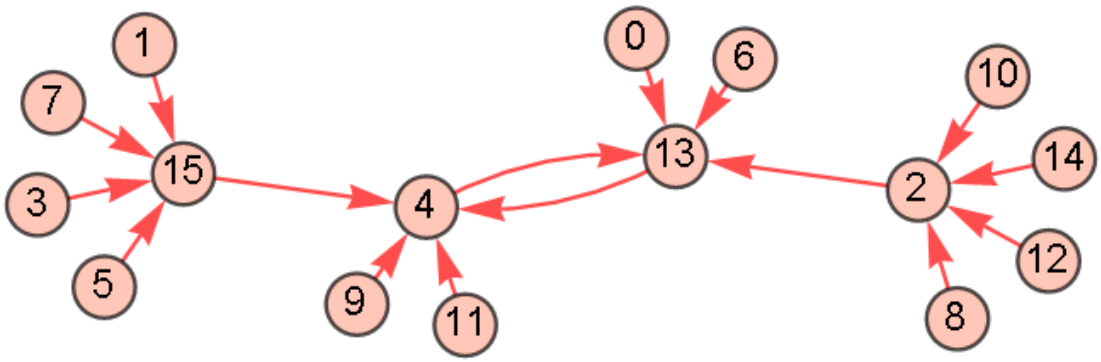


Figure 3.3: The state space of the Random Boolean Network whose topology is shown in Fig. 3.2 and whose dynamical rules are shown in Table 3.10. This Boolean Network has a cycle attractor of length 2 made up of the states 4 and 13. Moreover, the size of the basin of attraction is 8 and is made up of the states $\{0, 2, 4, 6, 9, 11, 13, 15\}$. Finally, the states $\{0, 1, 3, 5, 6, 7, 8, 9, 10, 11, 12, 14\}$ are the garden-of-Eden states.

3.2 The Dynamical Phases

Since in the end, the RBNs are dynamical systems, we can expect their dynamics will have distinct phases. To visualize them, we must plot the states of each node of the network in a grid such that every square represents a node and neighbor squares depend topologically as in the original network [21]. If the state of a node is "on" we can color it black and if it is "off" we can color it white. Then, we choose an initial state and let the dynamics flow. In this way, it is possible to distinguish three phases with unique features: the ordered, chaotic and critical phases.

The *ordered* or *frozen phase* is characterized because when we choose a random initial state, initially many states will be changing, however, it will be stabilized quickly and most of the nodes will be static, i.e., almost all nodes are frozen. Also, in this regime attractor cycles are short [27]. Another feature is that when we introduce small perturbations to the network, such as changing the connection between two nodes or flipping the states of a node, this damage usually will not be spread, i.e., the perturbation dies out and the behavior of the perturbed network is similar to the normal network. The final feature is that similar states tend to converge to the same state [21].

The *chaotic phase* is characterized because when we choose a random initial state, states are not stabilized and most of them stay changing, i.e., states have fluctuating values. Besides, in this regime attractor cycles can be long [27]. When we introduce small perturbations to the network this phase is highly sensitive and small changes tend to propagate through the network, i.e., perturbations have strong effects making these networks not robust[9]. This is the well-known butterfly effect which is exhibited by chaotic systems. Moreover, in this phase similar states tend to diverge [21].

[9]We say that a system is robust if it continues functioning after a perturbation [28]

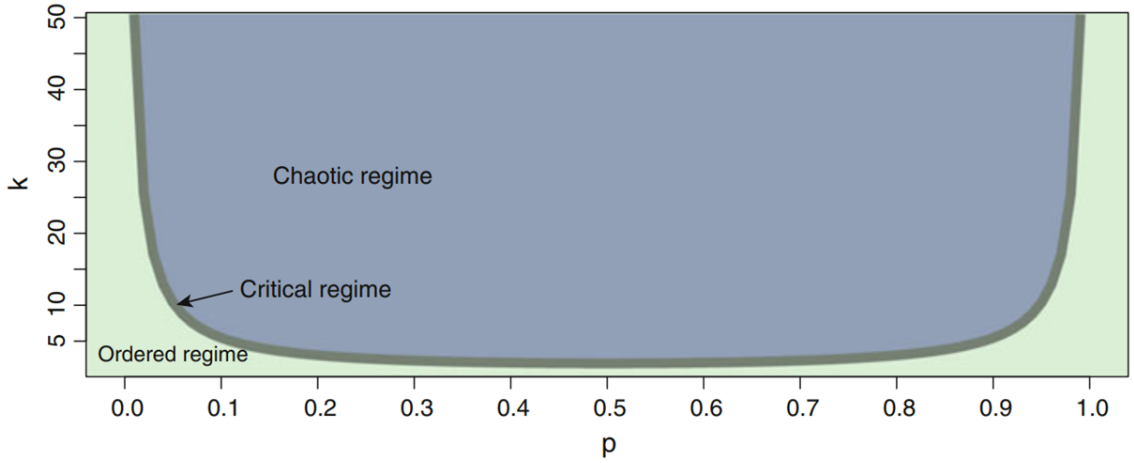


Figure 3.4: The critical curve which shows the border between the chaotic and ordered regimes as a function of the parameters k and p. Reprinted from [27] by permission from Springer Nature (adapted from [22] by permission from Elsevier).

The *critical phase* is at the border between the ordered and chaotic phases [20]. Between these two phases occurs a phase transition called the *edge of chaos*. At this transition phase, perturbations can propagate, though not necessarily through all the network. Besides, similar states perform a trajectory that neither diverges nor converges [21]. The networks within this regime are of biological interest because it is thought they are complex enough to allow biological processes but also flexible enough to deal with changes [27].

Apart from the visualization approach, there exist methods more standard to identify these phase transitions. The idea behind them is to measure the effect of perturbations, the sensitivity to initial conditions, or the damage spreading [21], i.e., the main idea is to distinguish the features of each phase by computing some statistical or analytical measurements.

It is known that in the limit $N \to \infty$ the phases are controlled by the parameter k and the bias parameter p we mentioned in Section 3.1.4.2 [27]. The system is chaotic for $k \cdot 2p \cdot (1-p) > 1$ and ordered for $k \cdot 2p \cdot (1-p) < 1$. In the uniform case $p = 0.5$, we have the ordered regime for $k < 2$, the chaotic regime for $k < 2$ and the edge of chaos at $k = 2$. In Fig. 3.4 can be seen the critical curve showing the boundary between these regimes for different values of k and p.

Though we will not delve into the subject and we will not further analyze the properties of these phases, we can mention, for instance, that for critical networks with parameter $k = 2$ and N nodes Kauffman originally found by computer simulations that the number of attractors goes as $\sqrt{N}$, although more recently it was shown that this number grows greater than any power law with N, i.e., it grows superpolynomially [27]. Meanwhile, in scale-free topologies, the transition from order to chaos occurs at the vale of the exponent γ_c which solves the transcendental equation [22]:

$$2p(1-p)\frac{\zeta(\gamma_{c-1})}{\zeta(\gamma_c)} = 1 \tag{3.12}$$

3.3 Other Updating Schemes

Coming back to the original paper of Kauffman, he focused his study on critical networks since as we argued before, they seem to be more robust to the life features. It is well known that different type cells from the same individual share the exact same genetic information, but they differ in their genetic activity, which differentiates them. In this way, Kauffman thought that having the same DNA equals to having the same network, i.e., each gene is a node of a network and thereby its number of attractors must be equal to the number of different cell types and the attractor's length to the cell cycle time [20]. As we mentioned, he found that the average number of attractors and their length was proportional to $\sqrt{N}$. He argued that the number of attractors found in the RBNs compared with the number of possible states is analogous to the number of cell types compared with the number of genes [21]. His idea was backed because in those years available data pointed out that the number of cell types seems to be proportional to the square root of the number of genes for distinct species [20]. At that time, the number of genes which forms the human genome was believed to be around 8000, thus agreeing with the number of human cell types [21]. Nevertheless, once the Human Genome Project sequenced human DNA, we knew it consists of around $25,000$ genes ([29], page 1239). This discrepancy is due mainly[10] to the assumption of synchronicity in the genes' activity. There is no reason to think genes change their state at the same time. Some genes change their state before others, i.e., their dynamics is asynchronous [21].

As was said at the beginning, the updating scheme which we will be considering is synchronous, which means states of nodes at time $t+1$ are determined by states of nodes at time t, so states in all the nodes must be updated at the same time. However, this model is not realistic, so for the sake of completeness, we will briefly define some variations which can be introduced to the original $N-k$ model by means of different updating schemes. We will name the original Kauffman network model as *Classical Random Boolean Networks (CRBN)*.

3.3.1 Asynchronous RBNs

In the *Asynchronous Random Boolean Networks (ARBNs)*, unlike CRBNs, at each step, only one randomly chosen node is updated. This asynchronous updating destroys the deterministic behavior of the network. There are no cycle attractors only point attractors can exist, though once the system falls into a point attractor it cannot escape from it. Also, there exist *loose attractors*, which are regions of the state space which capture the dynamics of the network, even though their order is not repeated deterministically due to the random updating order of the nodes [30].

3.3.2 Generalized Asynchronous RBNs

The *Generalized Asynchronous Random Boolean Networks (GARBNs)* are a generalization of the ARBNs. In this model, at each step, the nodes to be updated are

[10] Among other loopholes of this model, we can cite: the lack of consideration of scale-free topology and biased functions and the existence of non-coding DNA without any functionality [21].

chosen randomly, i.e., unlike ARBNs, in the GARBNs the number of randomly chosen nodes to be updated can be a number between one or all of them, even updating none is a possibility at each step. This model also is non-deterministic and lacks cycle attractors. Only point and loose attractors can exist [31].

3.3.3 Deterministic Asynchronous RBNs

In the model of *Deterministic Asynchronous Random Boolean Networks (DARBNs)*, we associate two parameters to each node: p and q such that p,$q \in \mathbb{N}$ and $q < p$. These parameters control the frequency of the update of each node. If the modulus of p over time equals q, then the node is updated. If two or more nodes are to be updated at the same step, then they are updated one after the other in an arbitrary order. This model does allow the existence of both point and cycle attractors, allowing the model of asynchronous but not random phenomena [30].

3.3.4 Deterministic Generalized Asynchronous RBNs

The model of *Deterministic Generalized Asynchronous Random Boolean Networks (DGARBNs)* is like the model of DARBNs, with the difference that nodes are updated if they fulfill the condition $(p \, mod \, t) == q$ and there is no arbitrary restriction to update them one after the other because all of them are updated at the same time, i.e., in a synchronous way [31].

3.3.5 Mixed-context RBNs

The *Mixed-context Random Boolean Networks (MxRBNs)* are like the DGARBNs, with the difference that each T time steps the values of p and q are randomly chosen from the sets $\bar{P}$ and $\bar{Q}$. These sets contain different values for p and q respectively which are called the contexts of the network [31].

All the random Boolean Networks previously discussed are classified as *Discrete Dynamical Networks (DDN's)* since they all have discrete-time, states, and values [30]. A diagram showing the classification of the RBNs discussed in this section can be seen in Fig. 3.5. The model of DDN's includes the *Multi-Valued Networks* in which nodes can take more than two discrete values (if they were continuous, they would be real-valued networks which study is up to dynamical systems theory)[30].

3.4 Applications of The RBNs

As we have mentioned, Random Boolean Networks were originally conceived by Kauffman as an abstract model of gene regulatory networks. Using these networks, we can model the causal link between genes by means of a directed edge, the expression or not expression of a gene by assigning to them a Boolean value, or even the stability of the cell (to damage, mutations, etc.) by means of the examination of the sensibility of attractors to perturbations, like a change in the network topology, a change in an updating function, a change in the state of a node, etc [32]. In this sense, Random Boolean Networks have been used to study the transcriptional network and the network regulating the cell cycle in the budding yeast

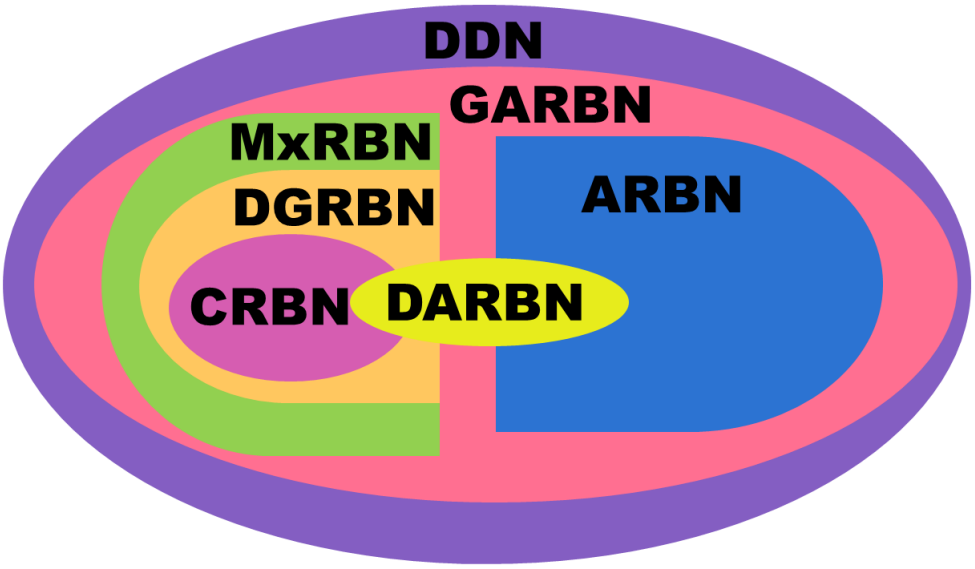


Figure 3.5: Classification of Random Boolean Networks according to their updating scheme [30].

Saccharomyces cerevisiae [33] [34]. The use of *Saccharomyces cerevisiae* had a great advantage because it is an organism well studied and thereby, we know or have an idea about the topologies of its regulation networks. If we know the topology of the network, half of our work is done, and we only must vary the random updating functions. Nevertheless, even if the topology is unknown, it is possible to infer it by means of experimental data (see [35]). When the genetic data is incomplete or insufficient, then *probabilistic Boolean Networks* are usually used [21]. The objective of modeling genetic networks with RBNs is to describe regulations at a system level, analyze and predict genomic interactions but also intervene in the network [36]. For instance, with the current knowledge about protein-protein interactions in cancer, it was possible to develop a topology to evaluate by means of the RBNs some possible molecularly targeted cancer therapies [37]. Among other applications of the RBNs to genetic modeling, we can cite the study of the cell cycle of the fission yeast *S. pombe* [38], the prediction of the expression pattern of the segment polarity genes in *Drosophila melanogaster* [39], the analysis of the mammalian cell cycle [40], etc.

Nonetheless, the field of application of the RBNs is not restricted to genetic networks. It has also been applied for example, to the study of Neural Networks [41], social segregation modeling [42], music generation [43], statistical physics [44], and due to the robustness and rich dynamics of the critical RBNs, their study represents an abstract study of the evolution of life [28]. Even an application to implement logic functions by Kauffman networks has been proposed [32]. Since the RBNs are a generalization of cellular automata, the number of possible applications is immense. Even though the RBNs could represent only a first step to the more realistic modeling of real-world networks, they already have proved to be of value in several different areas.

3.5 Software to Study RBNs

To study the properties, visualize the dynamics and simulate in an easy way the RBNs, we can resort to *The BoolNet R* package [27] or to *The PyBoolNet* package written for being used in python language [45]. Other software tools are *The DDLab*, *The RBNLab*, and the Matblab implementations *RBN Toolbox* and *BN/PBN Toolbox* [21].

Despite the existence of different software implementations to study the RBNs for various platforms, in the next chapters, we will use our own algorithms. This will allow us to better understand the functioning of the RBNs. Moreover, since we are only interested in the study of a few properties of the Random Boolean Networks, the creation of our own algorithms will enable us to focus on the features we need with the advantage that it is easier to debug the code that we have written ourselves. This flexibility does not exist when we use the software created by anyone else and we must adapt our needs to whatever the program offers us. What is more, these algorithms will be developed in the Wolfram Language, a powerful platform for which we have not found any implementation to analyze Boolean Networks. The algorithms used throughout this work to study the RBNs are presented at the end, in the Appendix section.

Chapter 4

Algorithmic Complexity

In this chapter, we will give a small revision of Kolmogorov complexity (K-complexity). The main objective is to explain how the library that we will use in the next chapter to measure the K-complexity works. Firstly, we will motivate the idea behind an algorithmic type complexity, suddenly the Turing machine device will be reviewed in order to be able to define this complexity, then we will discuss the former approaches to try to measure it. We will continue with some important theorems and definitions concerning the algorithmic complexity. Finally, we will describe with more detail the methodology behind the library which measures the K-complexity including some applications.

4.1 The Kolmogorov Complexity

Before we try to measure the complexity of something, we must ask: what is complexity? what makes something more complex than other things? is it possible to find a systematic way to measure complexity? but even more important, in the first place, we have to decide which kind of things we are interested to compute its complexity. A quick search in the literature will reveal that there exist different types of complexities such as the algebraic, the computational, the linear, etc. each one with its own features and not necessarily useful for the same purposes, moreover, there is no guarantee that their measurements will coincide or be proportional when applied to the same object. There is not a universal definition of complexity, and if there were any, it would be so ambiguous, that it would not be possible to state it mathematically, and it would look almost like the definition of the Oxford dictionary which defines something which is complex simply as: "connected of many different parts. Not easy to analyze or understand; complicated or intricate" [46]. Depending on the kind of object or process we are interested, we will have to choose (or build if it does not exist) a definition of complexity that suits our interests, that allows our object or process to be measured and that fits our intuition about how the complexity of that thing should increase or decrease. Therefore, although it should be possible, for instance, to measure if one car is more complex than other, we are not interested in that specific kind of objects and complexity because it does not make any sense for our purposes. Neither we are interested, for example, in knowing if the computation of the solution of an algebraic expression is more complex than another. In this chapter, the kind of objects we want to measure its complexity are the objects that can be stored in a computer, and when we say stored, we mean that

they can be represented in some way as information. So, in this kind of category, we could find an image, a text file, an audio file, etc. All these objects have in common that they are stored in a computer as sequences of bits, that is, sequences of $0's$ and $1's$. Therefore, we seek a complexity that can work with any object which can be represented as a sequence of bits.

Now, what should we expect about the complexity of a sequence of bits? Let's consider the following sequences:

$$x_1 = 01010101010101$$
$$x_2 = 00101110100101$$
$$x_3 = 00000000000000$$
$$x_4 = 11111111111111$$
$$x_5 = 0000000$$

The sequence x_1 is a repetition of 01 seven times, thus it could be written just as: $x_1 =$ "repeat 01 seven times", making it simpler than it looks. On the other hand, the sequence x_2 has no evident repetition pattern that allows us to compress it as we did with x_1, therefore it seems x_2 is more complex than x_1. The sequences x_3 and x_4 repeat the same digit fourteen times, therefore they are highly compressible, and we should expect they are less complex than x_1. Between them, these sequences should be equally complex because they can be described by the same sentence: "repeat α fourteen times", where α is one element of the alphabet $\{0, 1\}$. Finally, the sequence x_5 is shorter than the former sequences and is highly compressible. Then it must be less complex than x_3. In summary:

$$C(x_5) < C(x_3) = C(x_4) < C(x_1) < C(x_2)$$

where $C(x_i)$ denotes the complexity of sequence x_i. Thus, the type of complexity we are looking for must be sensitive to the next aspects:

1. Between two sequences of the same length, the sequence more complex will be the less compressible sequence, or in other words the sequence with fewer regularities.

2. This complexity must be symmetric, that is, it must be insensitive to a reflection of the alphabet elements, which means that 0101 and 1010 are equally complex.

3. Between two sequences of different length containing just repetitions of a shorter sequence, the less complex of the two sequences will be the sequence with the less length.

These observations were made for extreme cases, but our definition of complexity also must be sensitive to cases where there is no evident way to compress two sequences. For instance, for the sequences 0111011001 and 1101000101, there is no obvious way to compress them. Thus, we cannot say which one is more complex, however, we wish a definition of complexity which indeed can answer this question.

We also must remark two words we mentioned above: *regularities* and *compressible*. These two concepts are the foundation of two of the initial ways people tried

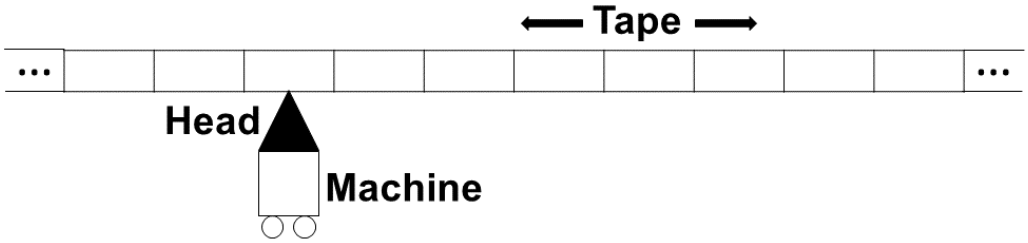


Figure 4.1: The Turing Machine.

to measure the complexity we are about to define. We will review these methods later in this same chapter.

The complexity that fulfills the previously stated needs is the *algorithmic complexity* also known as *Kolmogorov complexity*, *K-complexity*, *Kolmogorov-Chaitin complexity* or *program complexity* [1]. The idea behind the algorithmic complexity is to measure the complexity of an object not by the object itself but by the description that generates it. This area of study is known as algorithmic information theory. However, which kind of description should we use? and which kind of object generates the object we are talking about? A language could make a description very simple, but others could make it particularly complex. We need a universal language and a universal device which can construct objects using this universal language. To answer this question mathematicians had to wait until the advent of computer science and electronic computers, especially the ideas of Alan Turing. In fact, the idea of algorithmic complexity was not formally introduced until 1965 by the Russian mathematician Andrei N. Kolmogorov and the American mathematician Gregory J. Chaitin. An interesting historical review of the development of algorithmic complexity can be found in ([47], pages 86-92). Hence, before we define K-complexity, we will review the Turing machine.

4.1.1 The Turing Machine

In 1936, Alan Turing proposed the machine which now is named after him to provide a simple computational model which could be used to prove formal mathematical statements [48].

It consists of an unlimited linear tape divided into cells which are coupled to a finite state machine (the machine) through a moving head which can be situated at each moment in one of the cells of the tape. This tape acts as external memory or storage where each cell or square can have a symbol from a finite alphabet set. The head performs three functions in each cycle of the finite state machine: it reads the symbol in the cell of the tape, erases the cell and writes on it a symbol (it can be the same previous symbol) according to what it has read and the rules of the program (in the machine), the machine is moved together with the head to an adjacent square (to the left or to the right) which then becomes the next cell to be scanned to continue with the next cycle (see Fig. 4.2) [49].

[1] We will use indistinctly these names, whenever we use them, we are referring to the same definition of complexity.

Hence, we can think of a Turing machine basically as a finite state machine or finite automaton with the difference that it has unlimited auxiliary memory and the ability to reread its input and write and erase over it. This gives the Turing machine more computational power over a finite automaton but also makes it a hypothetical device[50]. We can think in the Turing machine as the simplest computer that could exist. Nevertheless, it does not mean this theoretical machine is less powerful than any modern computer, in fact, any problem that can be solved in a Turing Machine is also solvable in a modern computer and vice versa. This is known as the Church-Turing thesis [51].

More formally, we define a Turing machine by means of a septuple [48]:

Definition 4.1. A *Turing machine* is define by means of a septuple of the form:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where:

- Q is the finite set of states of the state machine.
- Σ is the finite set of input symbols.
- Γ is the complete set of tape alphabet symbols (Σ is a subset of Γ).
- δ is the transition function.
- q_0 is the initial state.
- B is the blank space symbol.
- F is the set of final states or accepting states which is a subset of Q.

An alternative way to define a Turing machine is by describing its actions, that is by describing the behavior of the transition function δ which can be thought as the program the machine executes. This can be accomplished by means of a set of quintuples [50]:

Definition 4.2. Let Q be a finite set of states and Γ a finite set of tape symbols (the tape alphabet) including a special symbol B. A *Turing machine* is a set of quintuples of the form (s, i, i', s', d) where $s, s' \in Q$; $i, i' \in \Gamma$; and $d \in \{R, L\}$ and no quintuples begin with the same s and i symbols. The symbol s is the present state, i is the tape symbol being scanned, i' is the symbol written, s' is the new state, and d is the direction in which the head will be moved (R is to the right and L is to the left).

The operation of a Turing Machine can be better understood through an example. Consider the Turing Machine defined by the quintuples in Table 4.1. This transition table defines a parity counter, that is, a program which output will be 0 or 1 depending on whether the number of 1's in an input string of bits (the input sequence) is odd or even [49]. The behavior of this machine for a given string is shown in Fig. 4.2. The end of the input string is indicated to the machine with the symbol B. The machine can be in two states q_0 and q_1 which we have denoted 0 and

1 for simplicity. Likewise, we have simplified the notation for the movement of the head denoting with a 1 the movement to the right and with a 0 the movement to the left. The machine starts in the state q_0 and halts when it reaches the symbol B after it has erased the input sequence and printed the answer.

s_j	i_j	$i'_{j,k}$	$s'_{j,k}$	$d_{j,k}$
0	0	0	0	1
0	1	0	1	1
0	B	0	H	-
1	0	0	1	1
1	1	0	0	1
1	B	1	H	-

Table 4.1: The transition table of a parity counter for a Turing machine. s_j is the initial state of the machine (0 or 1), $s'_{j,k}$ is the final state of the machine (0, 1 or halt), i_j is the initial symbol in the cell of the tape read by the machine (0, 1 or blank), $i'_{j,k}$ is the symbol printed by the head on the cell of the tape (0 or 1) and $d_{j,k}$ is the direction to which the head and the machine will be moved (1 means to the right, 0 means to the left and "$-$" means there is no movement).

4.1.2 The Universal Turing Machine

We have said that a Turing Machine is the simplest theoretical example of a modern computer, although, it has a problem. Modern computers can execute multiple different programs, we do not have to buy a new computer each time we need to perform a different computation with a different set of instructions. This is not the case with a Turing machine, for instance, consider the Turing machine with the transition table given in Table 4.1. This Turing machine runs a parity counter program, but what happens if now we want to run a program which computes a sum? In this case, we would have to build a brand-new Turing machine to execute this new task and we would have to do this step every time we needed to run a different program. Obviously, our current model of the Turing machine fails to mimic the ability of a modern computer to run different instructions. To overcome this problem, Turing was farther and conceived the idea of a Universal Turing Machine, a Turing machine which can simulate any other Turing machine. Or in words of Turing itself [52]:

> It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D[2] of some computing machine t, then U will compute the same sequence as T.

A universal Turing machine can be defined in simple words as follows [53]:

[2]S.D stands for standard description and is the set of quintuples which define the transition table placed in a line one after the other and separated by semi-colons. Although, the convention originally proposed by Turing to order the elements in a quintuple was (s, i, i', d, s') and not (s, i, i', s', d) as we have agreed.

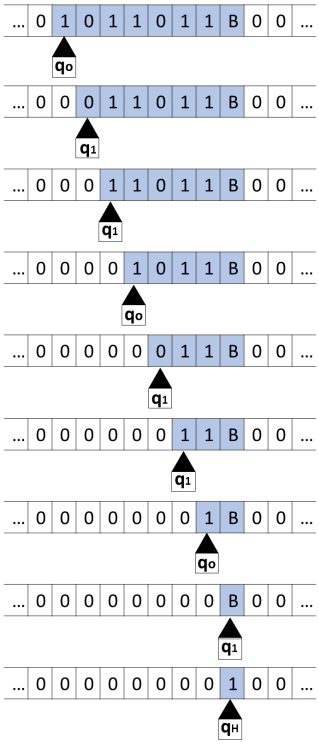


Figure 4.2: Configurations of the parity counter program shown in Table 4.1 in a Turing machine. The input sequence is 1011011 which has five 1's, thus the result of the computation is 1.

Definition 4.3. A *Universal Turing Machine (UTM)* is a Turing machine which, by appropriate programming using a finite length of input tape, can act as any Turing machine whatsoever.

If we wish a more formal definition, we first ought to define a Turing-computable function as follows [49]:

Definition 4.4. A function $f(x)$ will be said to be the Turing-computable if its values can be computed by some Turing machine T_f whose tape is initially blank except for some standard representation of the argument x. The value of $f(x)$ is what remains on the tape when the machine stops.

The standard representation used for the argument x does not matter so long as we are consistent with it [49], so we will use a unary notation were, e.g., $x = 1111$ means 4 and $x = 111$ means 3. Thus, we can define a UTM as follows [49]:

Definition 4.5. A single, fixed UTM is a machine with the property that for each and every Turing machine T, there is a string of symbols d_t (which Turing called S.D) such that: if the number x is written in unary notation on a blank tape, following the string d_t, and U is started in q_0 on the leftmost symbol of d_t, then when the machine stops the number $f(x)$ will appear on the tape, where $f(x)$ is the number that would have been computed if the machine T had been started with only x on its tape.

The former statements help to give a more formal definition of a universal Turing machine; however, it is possible to give an even more general definition (see [54],

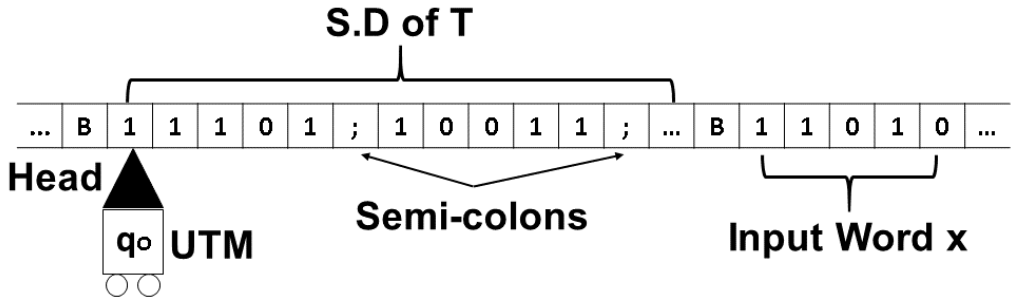


Figure 4.3: A universal Turing Machine U. The head of U starts in the state q_0 and begins scanning the leftmost symbol of the standard description (S.D) of the Turing machine T which U simulates. The S.D of T is created by putting the quintuples of T one after each other and separating them by semi-colons. After the S.D of T, the input word x is placed in unary notation. The computation of U will give $f(x)$, the same result that the computation of T would have given if it had started only with x on its tape.

page 140-144). An example of a Universal Turing Machine can be seen in Fig 4.3. An implementation of a UTM can be found in ([49], page 143-144).

4.1.3 The Halting Problem

When we use a Turing machine and a determined tape to perform a computation, in some situations the process to get a result may take just a few steps but in others, it could require much more time. In fact, the execution could take a lot of time that it becomes impractical, or even it could never halt. Thus, for practical reasons, we would like to have a way to know the time it will take the machine to halt or just confirm before we start the computation if the execution will ever halt. This is known as the halting problem and can be formulated as [50]:

Definition 4.6. The *halting problem* asks, does an algorithm exist to decide, given any Turing machine T and string x, whether T begun on a tape containing x will eventually halt?

Unfortunately, the following theorem which can be proved gives an answer to the halting problem [50] [49]:

Theorem 4.1. The halting problem is unsolvable.

Although the halting problem for a general Turing machine is unsolvable, this problem is solvable for machines with less than four states, furthermore, the four-state problem is open, and the five-state problem is almost certainly unsolvable [55].

A special case of the halting problem is the blank-tape halting problem [49]:

Definition 4.7. Is it possible to build a machine which will decide, for each machine T, does T halt if started on a blank tape?

Evidently, this special case is also unsolvable for a general Turing machine, however, this also means that the special cases for machines with just a few states do have a solution as stated for the general halting problem. Indeed, the Busy Beaver functions solve the problem, but we will return to this point later when we discuss the methodology which has been used to measure the Kolmogorov complexity.

4.1.4 The Formal Definition of K-Complexity

Prior to being able to formally define K-complexity, we ought to build a theoretical framework to make immediately easy to understand its definition. This is because this definition was the result of a process which took several years to be completed and many concepts had to be introduced and clarify before we could reach a satisfactory universal definition of complexity, see ([47] , pages 86-92 and pages 47-49) and footnote 3 in ([56], pages 122-123).

We will work with sequences or strings [3] of elements which belong to the nonempty set $B = \{0, 1\}$. The set of all finite strings over B is denoted B^*, where $B^* = \{\emptyset, 0, 1, 00, 01, 10, 11, 000, 001, ...\}$ and $\emptyset$ is the empty sequence [47]. There is a widely used, though not one-to-one[4] correspondence of B^* onto the natural numbers such that "100" maps to 4 and 101 maps to 5. It is known as counting in base 2 or binary representation. These sequences are represented such that [47]:

Definition 4.8. If x is a string of n $0's$ and $1's$, then x_i denotes the ith bit (binary digit) of x for all i, $1 \leq i \leq n$, and $x_{i:j}$ denotes the $(j-i+1)$-bit segment $x_i x_{i+1} ... x_j$.

For example, for $x = 1101$, we have $x_1 = x_2 = x_4 = 1$ and $x_3 = 0$. We will consider binary strings of different lengths [47]:

Definition 4.9. The *length* of a binary string s is the number of bits it contains and is denoted by $l(s)$ or $|s|$.

We can consider decoding functions which are defined next [47]:

Definition 4.10. If D is any function $D : \{0, 1\}^* \to N$, then if we consider the domain of D as the set of *code words*, and the range of D as the set of *source words*, $D(y) = x$ is interpreted as "y is a code word for the source word x, and D is the *decoding function*. The set of all code words for source word x is the set $D^{-1}(x) = \{y : D(y) = x\}$ and $E = D^{-1}$ is the encoding relation (or encoding function if D^{-1} happens to be a function).

Following it is necessary to introduce the concept of prefix-free codes [47]:

Definition 4.11. Let $x, y \in \{0, 1\}^*$. Then we call x a *prefix* of y if there is a z such that $y = xz$.

For instance, the string 10 is a prefix of 1011010 where $x = 10$, $y = 1011010$ and $z = 11010$. In addition, we define a prefix-free set as [47]:

Definition 4.12. A set $a \subseteq \{0, 1\}^*$ is *prefix-free*, if no element in A is the prefix of another element in A.

Furthermore, we have that [47]:

Definition 4.13. A function $D : \{0, 1\}^* \to N$ defines a *prefix-code* if its domain is prefix-free.

[3] We will use indistinctly these names as synonyms.

[4] It is not one-to-one because for example both "100" and "0101" map to the same number 5.

Definition 4.14. A code is a *prefix-code* or *instantaneous* if the set of code words is prefix-free (no code word is a prefix of another code word).

With the former definitions in mind we can define a prefix-free Turing machine [57]:

Definition 4.15. A Turing Machine is said to be *prefix-free* if the group of its valid programs forms a prefix-free set (no element is a prefix of any other).

Or in other words:

> A *prefix-free function* is one whose domain is prefix-free. Similarly, a *prefix-free Turing machine* is one whose domain is prefix-free. It is usual to consider such a machine as being *self-delimiting*, which means that it has a one-way read head that halts when the machine accepts the string described by the bits read so far. The point is that such a machine is forced to accept strings without knowing whether there are any more bits written on the input tape. This is a purely technical device that forces the machine to have a prefix-free domain, but it also highlights how the use of prefix-free machines circumvents the use of length to gain more information ([56], page 122).

Now we have the framework necessary to present the formal definition of the K-complexity [57]:

Definition 4.16. The ***algorithmic complexity*** $K(S)$ of a string S is the length[5] of the shortest program p that outputs the string S, when running on a universal (prefix-free) Turing machine U. That is:

$$K(S) = min\{|p| : U(p) = S\} \tag{4.1}$$

where $|p|$ is the length of the program p and $K(S) = \infty$ if there are no such p.

Unfortunately, the cost of being a universal definition of complexity [6] is that it is incomputable, i.e., there is no such function K which accepts a sequence S and returns the length of the shortest program which produces S. Nonetheless, the upper bound of Kolmogorov complexity is computable, so it is called upper semi-computable or lower semi-computable [7] [57]. In the next section, the former approaches to try to approximate the K-complexity will be reviewed.

4.2 Lossless Compression and Entropy as Approximations to K-Complexity

The concepts of algorithmic information theory were born as problems in the frontiers of probability theory and information theory. In fact, the original purpose of

[5] See Definition 4.9.

[6] This measure of complexity is said to be universal because it has been proved to be robust since several independent definitions converge to it, thus making it reliable [58].

[7] Upper in the sense that the upper bound can be found and lower in the sense that it is an approximation.

the definition of algorithmic complexity was to define randomness: "If the shortest program p producing s is larger than $|s|$, the length of s, then s is considered random" [58]. Therefore, its origins give a clue in how we can approximate K-complexity by using compression methods and entropy measurements which can detect regularities in data sequences. We will begin discussing lossless compression as an approximation to K-complexity to continue with entropy as another approximation.

4.2.1 The Lossless Compression

Theoretically, a compression function is defined as [59]:

Definition 4.17. A *compression function* is a computable function q such that $U(q(x)) = x$, for all strings x for some fixed universal Turing machine. The trivial compression function $q(x)$ simply outputs the program *"print x"*.

To be called lossless compression it has to exist an inverse function which fully recovers the original data x from $q(x)$, i.e., the function U, in a process called decompression [8]. Thus, we can see a compression algorithm as a function which maps an input string x onto other string y which is written using the same alphabet units of x. [9]. The goal of a compression method is that the length of the resulting string y plus the length of the instructions needed to reconstruct x (in our definition the instructions of the Turing machine) must be less than the original length of x. Among the most famous lossless compression algorithms, we can cite the Huffman Coding which creates a compressed code by means of an extended binary tree which takes into account the relative apparition frequency of sequence elements of a string (we briefly mentioned it in Section 2.6.7) (see [60], pages 131-133). Other famous methods are the Lempel-Ziv algorithms (LZ77 and LZ78) which are dictionary-based methods in which a dictionary of phrases is used to replace elements indicated by pointers in the string. Nowadays, variations of Lempel-Ziv algorithms are widely used to get compressed files in well-known extensions such as LHarc, PKZIP, GNU ZIP (GZIP), Info-ZIP, Portable Network Graphics (PNG), PDF, etc (see [60], page 229). One of the most important derivations is the DEFLATE algorithm which uses at the same time a combination of the LZ77 and Huffman methods [61].

Nonetheless, it does not matter which compression algorithm we choose, there will always be at least one file which will not be compressed to a smaller size if we try to do it [57]:

Proof. Strings of data of length N or shorter are clearly a strict superset of the sequences of length $N-1$ or shorter. It follows therefore that there are more data strings of length N or shorter than there are data strings of length $N-1$ or shorter. And it follows from the *pigeonhole principle* that it is not possible to map every sequence of length N or shorter to a unique sequence of length $N-1$ or shorter. Therefore, there is no single algorithm that reduces the size of all data. □

[8] If the information cannot be fully recovered it is called lossy compression (see [60], page 119). We are only interested in lossless compression so whenever we say compression, we are referring to it.

[9] If the resulting string y is written using the same alphabet units, the process is called data compression otherwise it is called data encoding [57]

Then it is necessary to have a way to quantify the goodness of compression. One way is with the compression ratio ([60], page 119):

Definition 4.18.

$$compressionratio = \frac{|x|}{|q(x)|} \tag{4.2}$$

If the compression ratio is greater than one it means that the compression algorithm worked otherwise it was not helpful. However, we must have in mind that this ratio does not consider how fast the algorithm works or how many information was preserved, and furthermore, it says nothing about the length of the instructions necessary to recover x. Nevertheless, in lossless compression methods, all the information is always preserved, and the length of the instructions are usually negligible ([60], page 119).

Initially, the lossless compression methods were mainly used as a measure of Kolmogorov complexity because compression is enough to test for non-algorithmic randomness (though the converse is not true) and provides an upper bound to K-complexity [57]. Therefore, the size of a compressed file is an approximate measurement of K-complexity and the most we can compress a file the less complex it is [62]. This compressed file can be thought as a program whose size is approximately the size of the shortest program p that outputs the string s when is decompressed [63]. With this approximation, it was possible to give practical applications to K-complexity in text classification, music [63], bioinformatics (genetic sequences) [64] [65] and clustering [66]. An elegant experiment demonstrating compression capabilities carry out by ants in nature is briefly reviewed in ([47], page 583).

The main problem with compression methods is that as was said, there is no single algorithm which can compress every string to a shorter size. This problem is more serious with short sequences since it is common that the result of applying a compression method to a short sequence will be a larger sequence due to certain data structures, headers, etc [62]. Besides, there is also a theoretical problem with this approach to short sequences: there is no way to further compress a single bit, which would mean that a single bit has maximal complexity. This result does not make any sense according to the intuition aspects we talk about at the beginning of the chapter and restricts the use of compression algorithms as an approximation of Algorithmic Complexity only for large sequences (larger than hundreds of bits)[63]. The main problem is that compression methods, like the LZ algorithms, work looking for repeated sequences [10], i.e., statistical regularities (redundancy) and not for algorithmic ones which make this approach no better than entropy-based estimations which we continue discussing in the following section.

4.2.2 The Shannon Entropy

Now we devote to briefly review the concept of entropy in information science. The branch of mathematics called information theory had its birth with a paper published

[10]There also exist probabilistic based compression algorithms, though the user usually does not have a way to update or infer a good statistical model of the source thus making necessary to use methods like the LZ (see [57] and [60] pages 193-195).

by the American mathematician Claude Shannon in 1948: "A mathematical theory of communication"[67]. Shannon was concerned with the problem of transmitting a message under the assumption that the ensemble of possible messages is known for both the sender and receiver no matter which meaning it could have ([47], page 65). Therefore, its measure of information does not take into account the content of the message but the action itself of choosing a message to communicate from an ensemble is what it cares about [11]. If the reader has taken a course in statistical physics he should have started to relate this approach with the notion of an ensemble of particles and the probability of finding a given state which in one way or another finishes hooked to the definition of entropy also known as Gibbs or Boltzmann entropy in this context. In fact, in a particular case, the Gibbs entropy is proportional to Shannon's entropy ([47], page 565) [12]. Besides the thermodynamics and statistical versions of entropy, we have a related definition of entropy in information science. Although the most common definition in computer science is that given by Shannon, there exist alternative definitions of entropy as the Kolmogorov-Sinai and the Rényi entropies which measure the amount of information needed to transmit the description of an object and rely on the assumption that the uncertainty measured by entropy is a non-decreasing function of the amount of information [69]. Thus, all of them exhibit the same behavior: Systems with low entropy are those with a little amount of information available and we can consider them simple [69]. Several other types of entropy definitions have been developed for specialized and punctual applications [69]. Despite this zoo of possible definitions, we will only work with the original statement of Shannon entropy which is defined as [60].

Definition 4.19. Let s be an event or phenomenon with a priory probability $P(s)$ of occurrence. We define the *amount of information disclosed or given off by the occurrence of and event s* as:

$$I(s) = log(\frac{1}{P(s)}) = -\log_b(P(s)) \tag{4.3}$$

If $P(s) = 0$ then $I(s) = \infty$.

Definition 4.20. Let $s_1, ...s_n$ an ensemble of possible events which are pairwise mutually exclusive with $\sum_{i=1}^{n} P(s_i) = 1$, then the average information of an event in the list ensemble or the *Shannon entropy* is:

$$H(s_1, ...s_n) = \sum_{i=1}^{n} P(s_i)I(s_i) = -\sum_{i=1}^{n} P(s_i)\log_b P(s_i) \tag{4.4}$$

Although the base of the logarithm usually is taken to be e by convention, sometimes this election depends on the number of alphabet elements we are working with, e.g., when working with binary sequences we can take the base of the logarithm to be 2. This election is not a big problem as the only effect it has is to multiply by a constant the result. If we consider the ensemble of possible events $s_1, ...s_n$ as a discrete random variable S with given probabilities $P(S = s_i) = p_i$ we write the **Shannon entropy** as:

[11] Opposite to Kolmogorov approach where we do measure the information contained in an object ([47], page 65)

[12] The study of entropy has its roots in the sunrise of thermodynamics and its second law in the nineteenth century with the work of Sadi Carnot, Clausius, et al. (See [68], chapters: 6,7, and 8)

$$H(S) = -\sum_{i=1}^{n} P(s_i) \log_2 P(s_i) \tag{4.5}$$

If $P(s_i) = 0$ we take $\log_2$ to be 0. This definition is generalized in the case of a continuous random variable [69]:

Definition 4.21. Given a variable S with n possible discrete outcomes such that in the limit $n \to \infty$ the density of S approaches the invariant measure $m(s)$, the *continuous entropy* is given by:

$$\lim_{n\to\infty} H(S) = -\int P(s)\frac{P(s)}{m(s)}dx \tag{4.6}$$

Shannon conceived its definition of entropy as a measure of the information transmitted over a stochastic channel when the alphabet elements are known and established hard limits to maximum lossless compression rates [57].

Now, we will see how we can use Eq. 4.5 through an example. Consider tossing a coin with equal probabilities of coming up heads or tails, i.e., $P(h) = P(t) = 0.5$. In this case, we have only two possible values or results when tossing the coin hence, when applying the entropy definition, we get:

$$\begin{aligned} H(toss) &= -P(h)\log_2 P(h) - P(t)\log_2 P(t) \\ &= -0.5\log_2(0.5) - 0.5\log_2(0.5) \\ &= -\log_2(0.5) = 1 \end{aligned}$$

Therefore, to send a message with the result of tossing a coin we would need to send one bit of information. In case we need to send the result of tossing the coin 10 times we would need to send 10 bits of information. This is the event with maximal entropy as the equal probabilities imply maximal uncertainty about the result of tossing a coin. However, if we now consider a loaded coin with unequal probabilities of getting heads or tails such that $P(h) = 0.2$ and $P(t) = 0.8$ we would get:

$$\begin{aligned} &H(toss) = -P(h)\log_2 P(h) - P(t)\log_2 P(t) \\ &= -0.2\log_2(0.2) - 0.8\log_2(0.8) \approx 0.721 < 1 \end{aligned}$$

This means to transmit the result of tossing a loaded coin we need less than one full bit. In case we need to send the result of tossing this loaded coin 10 times we would need to send in the optimal case with the optimal encoder around only 7 bits of information[13]. The unequal probabilities have reduced the uncertainty of tossing the coin which means the event of getting a result contains less information than the original coin with equal probabilities, i.e., the event of tossing a loaded coin has less entropy.

[13]Of course we can send 10 bits of information with the result of every event, however, less entropy means that it is possible to encode the result of tossing the loaded coin 10 times in a way we need less than 10 bits. In this case, with the most optimal encode we would need around 7 bits.

In the experiment of tossing a coin we only have two possible results, but we can have events with more results. For instance, consider an event X where the result can take five different values $\{a, b, c, d, e\}$ with probabilities: $\{P(a) = 0.2, P(b) = 0.1, P(c) = 0.5, P(d) = 0.1, P(e) = 0.1\}$. In this case the entropy is:

$$\begin{aligned} H(X) = &-P(a)\log_5 P(a) - P(b)\log_5 P(b) - P(c)\log_5 P(c) \\ &-P(d)\log_5 P(d) - P(e)\log_5 P(e) \\ = &-0.2\log_5(0.2) - 0.1\log_5(0.1) - 0.5\log_5(0.5) \\ &-0.1\log_5(0.1) - 0.1\log_5(0.1) \\ = &-0.2\log_5(0.2) - 0.5\log_5(0.5) - 0.3\log_5(0.1) \\ &\approx 1.9 > 1 \end{aligned}$$

This means we need almost 2 bits of information to transmit the result of an event, and in case we need to send the result of 10 events we would need around 20 bits in the most optimal case where we use the optimal encoder.

Nevertheless, it must be remarked, that in the last examples we are not talking about sending a set of results, i.e., we are not assigning these entropies to a sequence but rather we are calculating the entropy of a stochastic source.

In the case of a sequence, e.g., the sequence 1110101. First, we count the number of occurrences of the two possible outcomes. With this information can build the random variable which generated the sequence, in our case we get $S = \{s_0 = 0.285, s_1 = 0.715\}$. Therefore, applying Eq. 4.5 we get $H(S) = 0.863$.

However, what happens if we wish to know the amount of information needed to send the result of tossing two coins at the same time? i.e., the experiment of getting a result given that we got another result formerly. We could perform a tossing and send the result and then perform another tossing and again send the result. However, in some cases is a better and more efficient idea to first perform the experiments and then transmit the results. For such cases we define the joint entropy ([47], pages 68-69):

Definition 4.22. Let the *joint probability* $P(X, Y)$ of[14] the random variables X and Y be defined by: "$P(a, b)$ is the probability of the joint occurrence of event $X = a$ and event $Y = b$, i.e., the probability of event $Y = b$ occurring at the same time that event $X = a$ occurs. Hence, the *joint entropy* is given by:

$$H(X, Y) = -\sum_X \sum_Y P(X, Y)\log_b P(X, Y) \tag{4.7}$$

Now we can know the entropy of tossing two coins at the same time. Nonetheless, what if now we wish to toss 1000 coins at the same time? or if we want to know the entropy associated with a long sequence of events as 101010...10. This sequence is generated by repetition of the bits 10 and has the same number of $0's$ and $1's$, thus

[14]Some authors use the notation $P(X \cap Y)$ for the joint probability of events X and Y and $H(X \cap Y)$ for the joint entropy, this is in conformity with the notation for the intersection of two event sets $X \cap Y$ however, we will use the simpler notation $P(X, Y)$ and $H(X, Y)$

given these number of occurrences we have a random variable such that $S = \{s_0 = 0.5, s_1 = 0.5\}$. If we use Eq. 4.5, we will get that this sequence (or more precisely the source that generated it) has maximal entropy (this is just the example of a fair coin which discussed before), even though this sequence was generated in a very simple way. This result represents a problem if we want to use Shannon entropy to measure Kolmogorov complexity. According to Kolmogorov's ideas this sequence has low complexity. This problem arises because with the Eq. 4.5 we can only study the smallest granularity (1 bit) or 1-symbol block [15] of the sequence. On the other hand, if we use Eq. 4.7, we could study the entropy associated with blocks of length two. This could give a result that better captures the complexity of this sequence, but moreover, there should exist an optimal block length in which the entropy we get is minimal which would be in accordance with Kolmogorov complexity. Hence, for our purposes, it is necessary to generalize Eq. 4.7 to consider blocks of length n [70]:

Definition 4.23. The *joint entropy* of variables $X_1, ..., X_n$ is given by:

$$H(X_1, ..., X_n) = -\sum_{X_1} \cdots \sum_{X_n} P(x_1, ..., x_n) \log_b[P(X_1, ..., X_n)] \tag{4.8}$$

where $P(x_1, ..., x_n)$ is the probability of obtaining the combination of n symbols corresponding to a given block.

Finally, for infinite or very long sequences originated from a stationary source, we define the entropy rate of a sequence S as [57]:

Definition 4.24.

$$\lim_{n\to\infty} \frac{1}{n} \sum_{|s'|=n} H_n(s') \tag{4.9}$$

where $|s'| = n$ indicates we are considering all the generated strings of length n.

As was said before, entropy gives a clue in how much a finite sequence can be compressed, giving it a useful application for data compression. In fact, we can consider entropy as a measure of the amount of information that a finite sequence has [57]. Besides, it has been used as an approximation to Kolmogorov complexity, even though, it is an imperfect approximation for the simple reason that it is a computable function [57]. The best performance of Shannon entropy as an approximation of Kolmogorov complexity is gotten when a given sequence is partitioned in blocks of increasing size up to half the total length of the string. Then the entropy of the sequence is calculated for each one of these block lengths by means of Eq. 4.8. The best measure will be that with the block length that minimizes the entropy of the sequence, meaning that that partitioning better captures the periodic statistical regularities of the sequence [57].

The entropy is a bad approximation of Kolmogorov complexity due to the fact it only considers the frequency of occurrence of the events 0 and 1, i.e., it does

[15] We are assuming a uniform probability distribution for the $0's$ and $1's$ and for sequences of the same finite length.

not capture algorithmic features of the generative process [63]. Moreover, the value of Shannon entropy can be different for different invariant representations of the same object. This is the case of graphs since it has been shown empirically that the value of the entropy is strongly influenced by the representation used to describe the graph. Furthermore, entropy cannot consider more than one feature at the same time. All this gives as result high complexities to some graphs which should be classified as not complex (for a complete discussion of why entropy is a bad measure of graph complexity see [69] section 2.2).

4.3 Towards a Better Measurement of K-Complexity

The former discussion of lossless compression and Shannon entropy methods gives us the conclusion that we must search a better way to approximate Kolmogorov complexity. Due to the enormous amount of issues they have, both methods fail to provide a robust estimation to K-complexity.

An implementation which approximates Algorithmic Complexity has already been published, but before we try to understand how it works, we will present some definitions and theorems which will pave the road.

4.3.1 The Invariance Theorem

In Definition 4.16, we saw how Kolmogorov complexity is defined by means of a UTM, however, to which UTM is this definition referring? If we expect this definition to be robust enough to be considered a universal measure of complexity, this definition must be independent of the UTM chosen to measure K-complexity. This behavior is guaranteed by the invariance theorem which states that we can guarantee the convergence of the K-Complexity no matter what universal Turing machine we are using, this is because, in the end, every lossless description can be translated to another just by using a program of fixed length, which means the K-complexity will be equal up to a constant which we hope will be negligible for long sequences, that is ([57], also see pages 103-104 of [47]):

Theorem 4.2 (Invariance Rate)**.** If U_1 and U_2 are two *UTMs* and K_{U_1} and K_{U_2} the algorithmic complexity of S for U_1 and U_2, there exists a constant C_{U_1,U_2} such that:

$$|K_{U_1}(S) - K_{U_2}(S)| < C_{U_1,U_2} \tag{4.10}$$

where C_{U_1,U_2} is independent of S and can be considered to be the length (in bits) of a translating function between universal Turing machines U_1 and U_2, or as a compiler between computer programming languages U_1 and U_2.

The size of the constant C_{U_1,U_2} is unknown an can be arbitrarily large, although we can expect that for long sequences, the approximation with a machine U_a will converge to the true value [57], i.e., $K_{U_a}(S) \sim K_U(S)$. Unfortunately, this theorem says nothing about the rate of convergence and does not guarantee such convergence or what conditions a universal Turing machine must satisfy to give a monotonic convergence, even though, it implies that there always can exist a "natural" universal

Turing machine U_N such that K_{U_N} converges faster than any other UTM [57]. Lossless compression methods also are affected by a constant related with which lossless compression method we choose to use since there is no preferable method. This is the reason why lossless compression fails to describe the algorithmic complexity of short sequences since the accuracy we can get is directly proportional to the length of the string. We can expect that the longer the sequence the less impact of the constant in the approximation of complexity, i.e., the less the impact of the programming language or UTM we choose [58].

The importance of this theorem is that it allows to approximate Kolmogorov complexity with the experimental procedure we will discuss later, without having to worry about which UTM should be used, with the expectation that for long sequences the approximation with every UTM will converge to the same value, or at least will only diverge by a constant which is the length of the complier necessary to translate one program in a UTM to another program in another UTM. Moreover, it does not forbid an upper bound approximation to K-complexity [57].

4.3.2 The Algorithmic Probability and The Coding Theorem

The next definition is the cornerstone to approximate Kolmogorov complexity. Instead of calculating directly K-complexity itself, we will calculate the so-called Solomonoff-Levin algorithmic probability or just algorithmic probability, Solomonoff's semi-measure or Levin's semi-measure for short, but before we present it, let's do a gedankenexperiment. Suppose we sit a monkey in front of a typewriter, and we make him hit the keys. The monkey, who knows nothing about poetry or English grammar, the most he can do is hitting the keys in a random way. If we let him do this for a while, probably its writing will make no sense, however with enough time, he could be able to type a word. We can expect that short words will appear earlier as they are more probable. If we give him an enormous amount of time he could be able to type even an entire book like Moby-Dick. Apparently one of the pioneers of this idea was the French Mathematician Émile Borel [71]. The main point is what would happen if we change the monkey by a device which could generate random codes[16] which we then could try to compile in some programming language. Among all these codes, we surely will find many that will not work, and others which will be compiled but will do nothing. However, among the codes that can be compiled there will be programs which will return a string of bits[17]. As was pointed out in the original monkey experiment there will be output sequences which will appear more frequently, i.e., they will have a higher probability of appearance[18]. For instance, we can expect to get a sequence such as 00000000 more frequently than a sequence as 01010111. Hence, it seems to exist a connection between the complexity and the probability of appearance of a sequence. This probability of appearance is the algorithmic probability we mentioned before and which we define next [47][58]:

Definition 4.25. The algorithmic probability $m(S)$ of a string S describes the

[16]For simplicity the random codes could be written as sequences of bits.

[17]We will not use any ASCII type representation so the output of our programs when it exists, will always be a sequence of bits.

[18]We refer to a higher probability of appearing in the sense that there are more programs whose output will be that sequence.

expected probability of a random (prefix-free) program p running on a universal prefix-free Turing machine U producing S and halting:

$$m(S) = \sum_{p:U(p)=S} \frac{1}{2^{|p|}} \tag{4.11}$$

The algorithmic probability $m(S)$ satisfies:

$$\left[\sum_{S \in \mathbb{N} \cup \{0\}} m(S) \right] \leq 1 \tag{4.12}$$

Where S is the binary representation of the natural numbers including the zero. Thereby, the algorithmic probability $m(S)$ is said to be a semi-measure[19] since the equality in Eq. 4.12 holds exactly only for those UTMs for which every input is a halting program.

It must be emphasized that a UTM which is prefix-free, as pointed out in Definition 4.15, is one which group of valid programs forms a prefix-free set. This condition is necessary to guarantee that this probability can be well-bounded [58], such that $0 < m(S) < 1$. The summation runs over programs of all possible lengths, though, the major contribution is given by the program of shortest length [57], which makes possible to approximate $m(S)$ and at the same time approximate K-complexity through the following theorem which connects the Solomonoff's semi-measure with algorithmic complexity [58]:

Theorem 4.3 (The Coding Theorem)**.** There exists a fixed constant c, independent of S such that:

$$| - log_2 m(S) - K(S)| < c \tag{4.13}$$

Thus, according to this theorem, the Kolmogorov Complexity of a string can be calculated from its frequency of occurrence. The simpler the sequence the larger number of programs which outputs it and vice-versa. We just must rewrite Eq. 4.13 as (see [56],section 6.9):

$$K(S) = -log_2 m(S) + O(1) \tag{4.14}$$

Where the asymptotic notation means we can write [69]:

$$K(S) \approx -log_2 m(S) \tag{4.15}$$

The Algorithmic Probability $m(S)$ is said to be a universal semi-measure since it can be used with any string and can handle missing and multidimensional data [57]. This method of approximating the Algorithmic Complexity is more stable than other methods, solves the problem of short strings, and besides, it is the best method for long sequences [63]. Perhaps, the most important reason to use this method is that the results obtained with it agree with our intuition of what should be considered complex and what not [63].

[19]For a definition of semi-measure see [56] section 6.9 and [47] section 4.3.

4.4 A Library to Measure The K-Complexity

The implementation which we will describe in this section has already proved to be useful, robust, less dependent to different representations of the same object and moreover it is in accordance with our intuition on how it should behave. For instance, when used with evolving graphs it has been able to capture small topological changes, in opposite to entropy measures, furthermore, the variance in the results obtained when using different graph representations has been less than that obtained when using entropy methods (see [69]). For these reasons, between all the different methods to get an approximation to algorithmic complexity, we have chosen to use mainly this implementation in the next chapter. In this section, we will briefly review the experimental procedure by which libraries in different languages with this implementation were created by the *Algorithmic Nature Lab group*[20] and are available to be downloaded at *www.algorithmicdynamics.net/software.html*. An online implementation is available at *www.complexitycalculator.com*. The complete details about these implementations can be found in [57] and [58].

4.4.1 Methodology

4.4.1.1 The Coding Theorem Method (CTM)

This method is based in Eq. 4.11 and 4.15. According to them, it is necessary to find all the possible programs which output is the sequence S. This is impossible to do, at least we restrict ourselves to study the behavior of a subset of Turing machines, i.e., the next distribution must be computed [58]:

$$D(n,m,S) = \frac{|\{U \in (n,m) : U(p) = S\}|}{|\{U \in (n,m) : U(p) \quad halts\}|} \tag{4.16}$$

where (n,m) represents the space of all universal Turing machines with n states and m symbols with empty input. This function is an estimation of the algorithmic probability[21]. Therefore, an estimation of Kolmogorov complexity will be obtained with the next analogous equation to equation 4.15 [57]:

$$CTM(S) = -log_b D(n,m,S) \tag{4.17}$$

Where, as before, the base of the logarithm is the number of symbol elements in the alphabet. Now, according to Eq. 4.16, we only must find a way to know if a given Turing machine will ever halt to be able to compute $D(n,m,S)$ and $CTM(S)$. As was mentioned in Section 4.1.3, this problem is unsolvable for the general case, nonetheless, this problem has solutions for specific Turing machines with small n and m numbers, through the use of the Busy Beaver functions which are defined as [58]:

Definition 4.26. If σ_T is the number of $1's$ on the tape of a Turing machine T with n states and m symbols upon halting starting from a blank tape (no input), then we define:

[20] https://algorithmicnature.org

[21] As mentioned previously, a better approximation to Eq. 4.11 would be $D(n,m,S) = \frac{|\{U \in (n,m) : U(p) = S\}|}{|\{U \in (n,m)\}|}$, however for practical reasons it is a better idea to use Eq. 4.16 (see the discussion in [58]).

$$\Sigma(n,m) = max\{\sigma_T :\in (n,m) \quad and \quad T \quad halts\} \tag{4.18}$$

Alternatively, if t_T is the number of steps that a machine T takes before halting from a blank tape, then we can define:

$$S_\Sigma(n,m) = max\{t_T :\in (n,m) \quad and \quad T \quad halts\} \tag{4.19}$$

For instance, for Turing machines of $m = 2$ symbols, the first $S_\Sigma(n, 2)$ functions are $1, 6, 21$ and 107 [72]. This means, for example, that if a Turing machine of 2 symbols and $n = 4$ states has not halted after 107 steps, then it will never halt. This allows to immediately discard such Turing machines, thus saving computational time. Although, the Busy Beaver value for $n = 5$ is unknown, experimentally a value around 500 was used [58]. This method was performed on 5 million strings which maximum length was 12, and the Turing machines had an alphabet ranging from 2 to 9 symbols [57]. To give us an idea of the computational power needed to obtain good results, the number of Turing machines with n states is given by $(4n + 2)^{2n}$, which means that for $m = 4$ there are $11,019,960,576$ machines to test [58], though, for $n = 5$ it was not necessary to run all of them because of symmetry considerations (see [58]).

4.4.1.2 The Block Decomposition Method (BDM)

As was said in the previous section, the Coding Theorem Method approximates Kolmogorov complexity from the output distribution of Turing machines. Nonetheless, due to the enormous computational power needed to test the gigantic number of UTMs of 2,3,4 and 5 states, statistics have only been gathered for sequences of length less than 12. To overcome this problem, it was developed a block decomposition method (BDM) which recycles this information to extend it to sequences of any length.

When dealing with a sequence S of length larger than 12, this method first decomposes it into sub-sequences $\{s_1, s_2, ..., s_k\}$, of length $|s_i| = l \leq 12$. If the number l is not a multiple of the length of the original sequence $|S|$, then there will be a remainder, i.e., a subsequence of length $|s_i| < l$. We can either choose to ignore it in our computation or we can use overlapping sub-sequences s_i to cover the entire sequence. Following we use the next definition to approximate the complexity of S [57]:

Definition 4.27.

$$BDM(S,l,m) = \sum_{i}^{k} [CTM(s_i,m,k) + log(n_i)] \tag{4.20}$$

where n_i is the multiplicity of the sub-sequence s_i of length l. The possible remainder fulfils the condition $r < |l|$. The overlapping parameter m is such that $1 \leq m \leq l$. When $m = l$ there is no overlapping while for $m < l$ there will exist overlapping between the sub-sequences s_i.

The overlapping parameter defines a sliding window of size $l - m$. An example of decomposition of a sequence is shown in Fig. 4.4. Nonetheless, the smaller m the greater the overestimation of BDM we will get [57].

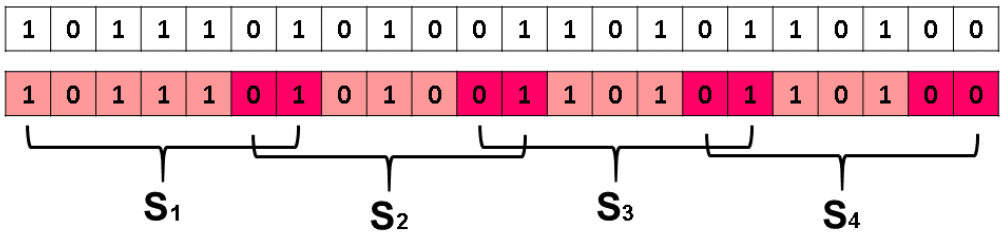


Figure 4.4: Decomposition of a sequence of length $|S| = 22$ into sub-sequences of length $l = 7$ through a sliding window of size 2, i.e., $m = 5$.

Thereby, this method combines a local estimation of Kolmogorov complexity with a classical entropy like measure in the long-range [57]. Finally, it must be mentioned, that in the best case this method approaches better algorithmic complexity than other methods and in the worst case, it behaves just like Shannon-Entropy [57].

4.4.1.3 Generalizations to More Dimensions and Symbols

If we wish to measure the complexity of sequences of bits in two dimensions, i.e., matrices of $0's$ and $1's$, we can take two approaches. The easiest one is just to flatten the matrix into a unique sequence, this can be achieved by putting each row(or column) of the matrix one in front of the other to form a unique string [69]. Another approach is just to generalize our coding theorem and block decomposition methods. Instead of using Turing machines which work out in a 1D tape, we use Turing machines which run over a 2D plane divided into cells [62]. In this way, the matrices less complex will be those with a greater probability of apparition. For instance, the matrices which are full of zeros or ones must be the less complex. As we can no gather statistics for matrices of all lengths, we can rely again on the block decomposition method to decompose matrices of large dimensions into blocks of matrices of shorter dimensions. An interpretation of this generalized method is that we are seeking the complexity of reconstructing the adjacency matrix associated with a graph from scratch [57]. Thereby, there is an immediate application to graph theory, but also to other objects, such as images, which in the end are matrices.

Finally, this method can be generalized to any dimensions in the exact same way we discussed for 2-dimensions [57]. Furthermore, the former methods can be generalized to consider Turing machines with more alphabet elements than just 0 and 1. For example, with four output symbols, the application could be immediately used to measure the complexity of DNA sequences. Although for the most purposes what we have discussed with two output symbols suffices.

4.5 Some Applications of Kolmogorov Complexity

We end this section commenting on some applications of the K-complexity. We find the main applications in areas where is important to measure the amount of information that an object has because, in the end, this is one of the features that the

Kolmogorov complexity allows us to know. Therefore, it has been used in physics, economy, linguistics, psychology, image treatment, image classification, etc [63] [57]. In the area of biology, it could be particularly important in the area of genetics, where researchers are interested in using sequences of DNA data to establish the evolutionary relationship of species [73]. Among others specific applications we can cite psychometrics [74], cellular automata [75] [76], graph theory [69], machine learning [77] [78], economic time series [79] [80], dynamical systems [81], computer network management [82] and general computation theory [83]. A compilation of applications to theoretical physics, information and computation can be found in ([47], chapter 8). Finally, in the next chapter, we shall present a novel application of K-complexity to Boolean Networks.

Chapter 5

The Complexity of Random Boolean Networks

In Section 3.4, some of the many different applications of the Boolean Networks were mentioned, especially, those for the Random Boolean Networks. These networks are useful to model and simulate networks that can be found in nature, but also in areas such as sociology, economics, computer science, etc. Hence, its study is important since they can be used to elucidate the hidden properties of these systems. Evidently, the capacity of a network to simulate a system depends on the topology and in the updating functions used. There are Boolean Networks which are more powerful than others. One indicator of the power of a Boolean Network must be its complexity. A network which is simple should not be able to imitate the behavior of complex networks such as the genetic regulatory ones. Nonetheless, the question of how to measure the complexity of a Boolean Network is not trivial. The main problem is that a Boolean Network is defined by two independent components, its topology, and its updating functions. These components then are brought together to create a dynamical system. This dynamical system is what we properly know as a Boolean Network and the object to which it is desirable to measure its complexity. Thus, measuring the complexity of its components separately may not be the best approach. We must devise a way to measure the complexity of both components acting together.

In this chapter, we will propose a way to measure the complexity of Boolean Networks. This proposal will be tested by using Random Boolean Networks. We will begin by measuring the complexity of random sequences of bits since they will be the key for measuring the complexity of Boolean Networks. Later, a procedure to measure the complexities of the individual components of a Boolean Network will be proposed. First, a method to measure the complexity of the topology will be studied. The topology is a directed graph, so this method will be tested firstly with graphs and then with digraphs. Afterward, the effect of the isomorphic representation used to represent the network will be studied. Next, a method to measure the complexity of the updating functions will be proposed. Finally, a method to measure the complexity of Boolean Networks will be presented and tested.

5.1 Software and Hardware Features

To perform the experiments of this chapter, the implementation described in Section 4.4 to measure Kolmogorov complexity will be used. Specifically, the implementation written in the Wolfram Language (Mathematica). This library contains the function *StringBDM* which measures the complexity of sequences of bits. It accepts a string of $0's$ and $1's$ as input and returns the complexity computed by means of the Block Decomposition Method (BDM) with the help of short sequences for which K-complexity was computed by means of the Coding Theorem Method (CTM). It is possible to control the overlapping parameter of the method, though by default it is set to be 1. We always will use the default parameter.

Additionally, this library contains the function *BDM* which allows measuring the complexity of adjacency matrices. These matrices must be composed of only $0's$ and $1's$, and they are introduced to the function as a list of lists. This is the 2-D generalization of the original Block Decomposition Method. It uses the complexity computed by means of the CTM for square matrices with sizes ranging from 1×1 to 4×4. In this work, we will only use the 4×4 matrices to perform the 2-dimensional BDM.

In this chapter, every time we mention Kolmogorov complexity, K-complexity or algorithmic complexity, we are referring to the approximation computed by means of the Block Decomposition Method. The codes of this chapter were implemented in *Mathematica 12* (version 12.0.0.0) in a 64-bits system with *Windows 10*.

5.2 The Complexity of Random Sequences of Bits

We performed this first experiment to be sure that the implementation to measure Kolmogorov complexity works as expected and to gain some experience in its use. The idea is simple, we generate a given number of 1-dimensional vectors of the same size. These vectors are initialized with zeros and we introduce them in a cycle. In the first iteration, we take the first of these vectors and we introduce it to another cycle (we have a nested cycle). In this nested cycle, we run over the indices of the vector, and with a probability p we change the zero in each element of the vector by a one (the probability that we do not change the zero in each element is $1-p$). Once we have iterated through all the elements of the vector, we measure the Kolmogorov complexity of the resulting sequence. We also measure the Shannon entropy of the resulting sequence by means of the *Entropy* built-in function of Mathematica and we compress it in $GZIP$ format to measure its file size. To better capture the complexity as a function of the parameter p, we repeat the process of replacing zeros by ones for each sequence T times, measuring the Kolmogorov complexity, the entropy, and the file size each time. Finally, we compute the quartiles for each set of measurements and take the mean by only considering the second and third quartiles. This trimmed mean procedure eliminates possible fluctuations in the results which can be seen as noise [1]. We save the results.

[1] For instance, it is possible that although the parameter p would have a high value like 0.9, the sequence obtained can be almost full of zeros, even though it so unlikely. This method allows us to avoid this problem and better capture the complexity of the sequences as a function of p.

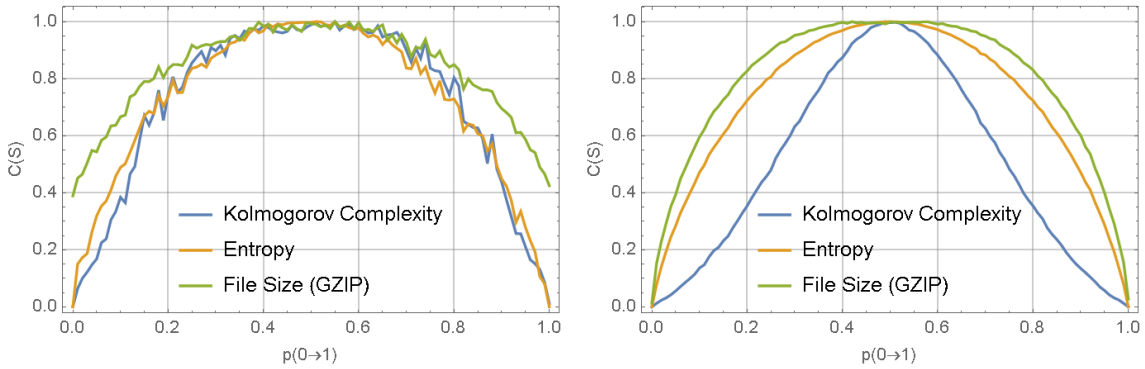


Figure 5.1: Complexity measurements for random binary sequences with three different methods versus the probability of changing a 0 by a 1 in each element of the sequence. For purposes of comparison, the results have been normalized such that the maximum value in each method equals 1. (a) Sequences of length 100, (b) Sequences of length 10000. See the text for more details.

Once we have finished, we take the second vector and repeat the same algorithm, however this time the probability of changing a zero by a one is $p + dp$. We do so until we have used all the vectors, summing dp to the probability parameter in each iteration. Evidently, the parameter p is such that $0 \leq p \leq 1$. Therefore, we initialize $p = 0$ and make sure that the number of vectors is such that to the last one will correspond the probability $p = 1$, i.e., the first vector will be full of zeros, and the last one will be full of ones.

The result of two experiments is shown in Fig. 5.1. We used sequences of length 100 and 1000. In both experiments, the step size in the probability was $dp = 0.01$ and the number of random sequences generated with the same p value was 10, from which the trimmed mean described before was done. As expected, for short sequences the lossless compression method failed to describe the complexity of the sequences since we expected the sequences with $p = 0$ and $p = 1$, i.e., sequences full of zeros and ones, to have a complexity close to zero. Meanwhile, the entropy and Kolmogorov complexity had a quite similar performance. As can be seen, for longer sequences the behavior of the entropy was more like the lossless compression method, which means that compression algorithms describe better the complexity of long sequences. However, these two methods were far from the result given by Kolmogorov complexity which as we argued before must be a better measurement of complexity.

Therefore, we can say that in general, lossless compression techniques are the worst, particularly for short sequences (around hundreds of bits). For short sequences, Shannon entropy has a similar performance to Kolmogorov complexity. Nevertheless, for long sequences (around thousands of bits) lossless compression techniques have more acceptable performance and their result is similar to that given by the entropy, although, both methods have a poor performance when compared with Kolmogorov complexity which we expect to be the nearest value to the

true complexity value of the sequence, so we always should choose this method, especially for long sequences, leaving the use of entropy only for short sequences and avoiding if possible the lossless compression techniques. If we cannot measure Kolmogorov complexity, we always should try to use entropy over compression methods, especially for short sequences.

The code used to carry out this experiment can be found in the Appendix Section A.1 (Fig. A.1). A similar experiment with similar results was performed by [57] (section 6).

5.3 The Complexity of Random Graphs

In this section, experiments regarding the complexity of random graphs will be presented. The main idea is to generate random graphs following a graph distribution. The complexity of the generated graphs is controlled by a parameter which can introduce order or disorder depending on its value. The experiments presented in this section are based on the work presented in [69].

5.3.1 Complexity from The Watts-Strogatz Graph Distribution

The first graph distribution which will be used to generate random graphs will be the Watts-Strogatz graph distribution. This graph distribution can be easily implemented in Mathematica by means of the built-in function *WattsStrogatzGraphDistribution*. We described how this function works in Section 2.9.1 and Fig. 2.25. A random graph is randomly chosen from this distribution using the built-in *RandomGraph* function.

The algorithm is like that used for measuring the complexity of random sequences of bits. We use a cycle to generate random graphs. The complexity of each graph obtained is controlled by the parameter p, i.e., the probability of rewiring. The initial value of this parameter is $p = 0$ and at each iteration, we increase its value using a probability step of size dp. The generation of graphs is stopped until we have reached the value $p = 1$.

At each step, the complexity of the generated graphs is measured by using the adjacency matrix as a representation of each graph. This time, only the Kolmogorov complexity and the Shannon entropy methods are used, since as was seen in the former section, lossless compression techniques are no better than the entropy to describe complexity. To do so, the adjacency matrix is flattened, i.e., we create a unique binary vector which contains the rows of the adjacency matrix one after the other. In this way, the complexity of each graph can be measured as usual, just by measuring the complexity of this binary sequence. Additionally, for purposes of comparison of this approach, we also used the 2D generalization of the BDM method which allows measuring directly the Kolmogorov complexity of adjacency matrices.

To eliminate fluctuations in the results, we also performed a trimmed mean procedure as we did for binary sequences, which means that we generate a given number

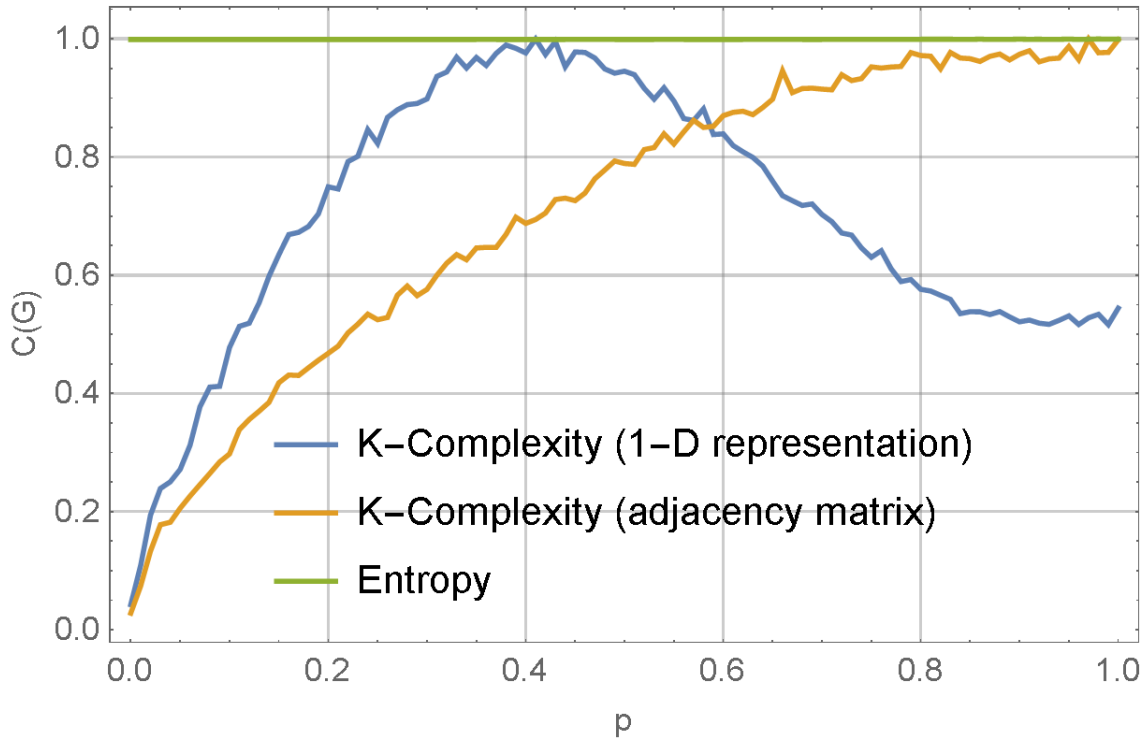


Figure 5.2: Complexity measurements using three different methods of random graphs obtained from the Watts-Strogatz graph distribution versus the probability parameter p. The graphs generated had 100 nodes and were created starting from a 10-regular graph following the Watts-Strogatz procedure. The number of random graphs generated with the same p value was 10 from which the trimmed mean described before was done. For purposes of comparison, the results obtained with each method were normalized with respect to the maximum value.

of graphs using the same probability parameter p. Then, we measure their complexity and compute the quartiles of this set of results. Finally, the mean is computed by considering only the second and third quartiles.

As we saw in Fig. 2.25, graphs generated using the parameter value $p = 0$ seem to be highly regular and thereby not complex. On the other hand, graphs with the parameter value $p = 1$ seem to be highly random and thereby very complex. Intermediate p values seem to have a complexity in between. Hence, we expect this experiment will show this complexity behavior as a function of the parameter p.

The result of this experiment is shown in Fig. 5.2. We can see that the results obtained by measuring the entropy of the 1-D representation of each graph remained practically constant, which means that this method is not capable of detecting changes in complexity by varying the parameter p. Meanwhile, the results obtained by measuring the Kolmogorov complexity with the 1-D representation had a better performance at the beginning, however, they reached its maximum approximately in $p = 0.4$ and then the complexity value started to fall. On the other hand, the measurement of complexity using directly the adjacency matrix representation captured exactly the expected behavior of the complexity versus the parameter p, i.e., the complexity at $p = 0$ is low and it increases until it reaches its maximum at

$p = 1$. This increasing seemed to follow a power-law $c(G) \sim p^{\alpha}$ with $0 < \alpha \leq 1$.

The code used to carry out this experiment can be found in the Appendix Section A.1 (Fig. A.2).

5.3.2 Complexity from The Barabási-Albert Graph Distribution

This experiment is almost the same as the former, but now we will use the Barabási-Albert graph distribution to generate random graphs. This distribution is implemented in Mathematica by means of the built-in function *BarabasiAlbertGraphDistribution*. We have already described how this function works in Section 2.9.2 and Fig. 2.26. As we mentioned before, this distribution is controlled by the parameter k, which is the number of edges of the vertex which is added at each step of the procedure.

According to Fig. 2.26, we expect that the complexity of these graphs will increase with the parameter k. However, this complexity must reach a maximum and start to fall from it as we continue increasing k, because for a sufficiently large value of k, the number of available nodes to link these k edges will not be sufficient and we will end with the case where each node will be linked to every other node, i.e., we would get a complete graph which would be highly regular and symmetric, thus with a low complexity.

The result of this experiment is shown in Fig. 5.3. As can be seen, the results showed the behavior expected by intuition. The complexity increased with the parameter k and reached a maximum approximately around $k = 30$ from which it began to fall. What was surprisingly in these experiments is that the results obtained using the three different methods to measure the complexity were very similar.

From the results obtained in this experiment and the former, we can see that entropy is not a robust measure of complexity since it had a good performance when using the Barabási-Albert graph distribution but when using the Watts-Strogatz graph distribution the results were awful. On the other hand, the results obtained by using K-complexity agree with our intuition about the complexity of graphs. Nonetheless, the performance obtained when using the traditional 1-dimensional K-complexity method seems still depend on the representation chosen for the graphs since when we tried to measure the complexity of a graph generated from the Watts-Strogatz graph distribution, we found that its performance can be a little different from expected, even though the 1-D representation used was lossless. Even so, with the Barabási-Albert graph distribution, our 1-D representation had no problems, so even though its performance is not as good as using directly the adjacency matrix representation, there exist objects which do not have an adjacency matrix representation, so for those objects we can rely on a 1-D lossless representation to measure its complexity.

The code used to carry out this experiment can be found in the Appendix Section A.1 (Fig. A.3).

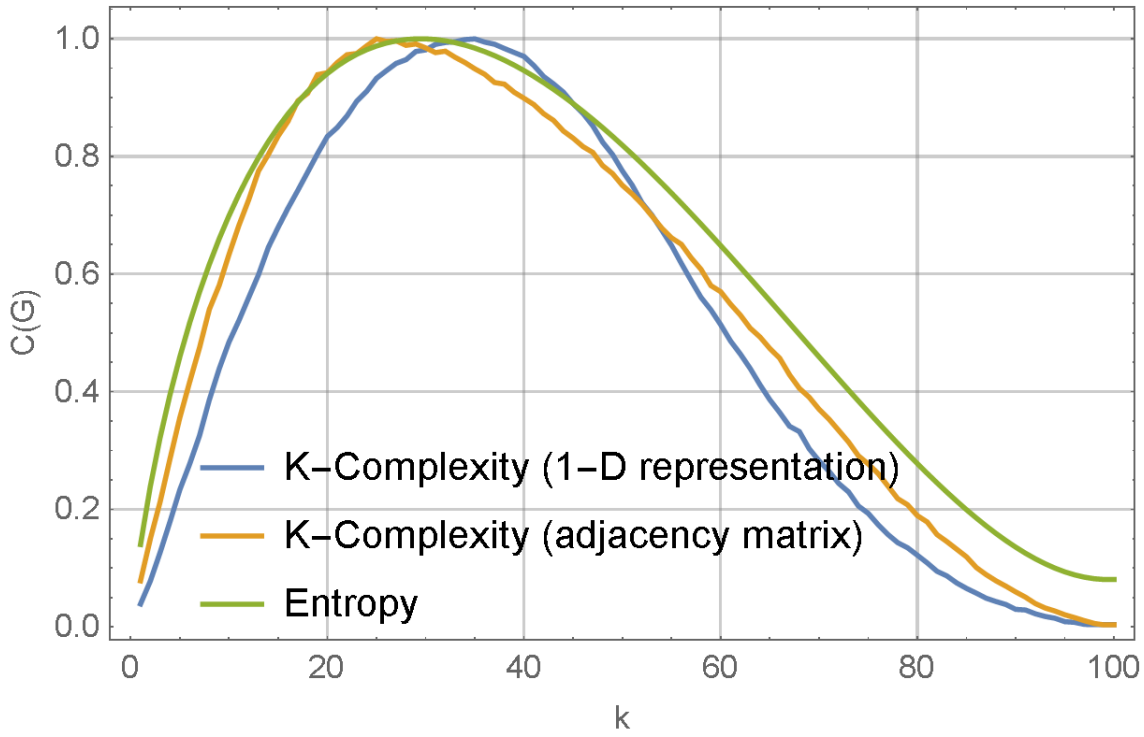


Figure 5.3: Complexity measurements using three different methods of random graphs obtained from the Barabási-Albert graph distribution versus the parameter k. The graphs generated had 100 nodes and each one was created starting from a cycle graph of 3 nodes following the Barabási-Albert procedure. The number of random graphs generated with the same k value was 10 from which the trimmed mean described before was done. For purposes of comparison, the results obtained with each method were normalized with respect to the maximum value.

5.4 The Complexity of Random Digraphs

Now, in this section, we will perform experiments like those shown before but now with directed graphs. We are interested in measuring the complexity of random digraphs because they serve as the topology for Random Boolean Networks whose complexity will be studied later. This time we will use a uniform digraph distribution. Nevertheless, we need to find a parameter or a way to verify that the complexity we are measuring makes sense.

If we simply use a uniform digraph distribution of n nodes and k edges, picking out a random digraph and measuring its complexity, then the digraph chosen can be complex or not since we do not know what to expect. Hence, to control the complexity of the digraphs chosen, our first approach is to gradually increase the vertex in-degree d^- of the nodes. Another option could have been increasing the number of nodes n, however, we consider that this approach lacks interest since it is obvious that for instance, any digraph of 50 nodes will be more complex than any digraph of 20 nodes, but it is not obvious if any digraph of n with in-degree $d^- = 50$ is more complex than any other digraph of n nodes with in-degree $d^- = 20$. On the other hand, the vertex out-degree d^+ can take any value. This choice is also motivated because in Classical Random Boolean Networks the topologies are chosen with a fixed number of nodes n with a fixed vertex in-degree d^- and a vertex out-degree d^+ which can vary. In the same sense, the digraphs generated will be allowed to have loops, though parallel edges (pointing in the same direction) will not be allowed. Thus, strictly speaking, we should call them multidigraphs, but we will continue calling them just as digraphs for simplicity.

5.4.1 The Complexity from Increasing the Vertex In-Degree

In this experiment, we will generate a random digraph of n nodes and d^- incoming directed edges, and as usual, at each iteration its complexity will be measured. To measure its complexity we will use the three same methods we use in the last section, i.e., we will measure the Kolmogorov complexity directly from the adjacency matrix representation of the digraph, but also a 1-dimensional lossless representation will be created by putting the rows of the adjacency matrix one after the other. Then, the entropy and K-complexity of this 1-D representation will be measured. The vertex in-degree d^- of the random digraphs will be increased at each iteration while vertex out-degree d^+ can vary. A procedure of trimmed mean to reduce fluctuations in the results will be performed as before as well.

It is expected that as the vertex in-degree d^- is increased, the complexity of the digraphs also will increase. However, as happened with the experiment in Section 5.3.2, this complexity must reach a maximum and start to fall. This should happen since the case where the vertex in-degree d^- equals the number of nodes and therefore also the vertex out-degree ($d^- = d^+ = n$) is highly regular and symmetric, and thus with low complexity.

The result of this experiment is shown in Fig. 5.4. As can be seen, these results were very similar to the results of the experiment with random binary sequences

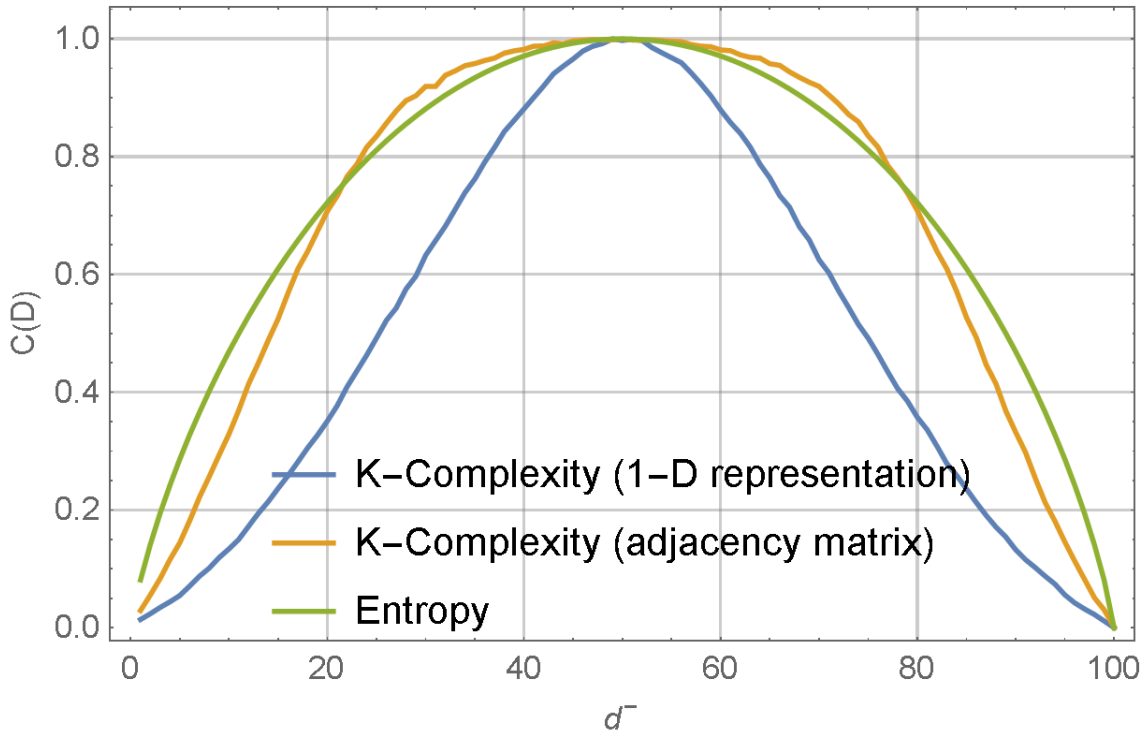


Figure 5.4: Complexity measurements using three different methods of random digraphs obtained from a uniform digraph distribution versus the number of incoming directed edges d^-. The digraphs generated had 100 nodes and the parameter d^- was vary from 1 to 100. The number of random digraphs generated with the same d^- value was 10 from which the trimmed mean described before was done. For purposes of comparison, the results obtained with each method were normalized with respect to the maximum value. Compare these results with those shown in Fig. 5.1b.

(Fig. 5.1b). In that experiment, we saw how Kolmogorov complexity had a better performance, showing a characteristic shape on its curve which could not be imitated by the entropy or compression techniques. If we now try to find this characteristic curve shape in the results in Fig. 5.4, we find that it is achieved by the Kolmogorov complexity measurement of our 1-D lossless representation. This is astonishing, considering that in the former experiments with graphs the adjacency matrix representation had a slightly better performance. This means that for directed graphs the lossless 1-dimensional representation we have proposed works better capturing the changes in complexity than the adjacency matrix representation, which this time gives results very similar to the entropy. Nevertheless, the three methods were able to show the behavior of the complexity that we expected, i.e., there is a maximum value in the complexity which in this case is found just in the middle ($d^- = 50$) from which the complexity starts to fall, moreover, the curve is symmetric around this point.

From this experiment, we can conclude that the 1-D lossless representation we have proposed works better to describe the features which make a digraph to be complex or not through the Kolmogorov complexity.

This time the random digraphs were not generated by using a built-in function because Mathematica does not have any function which can be used to generate random digraphs with a given vertex in-degree d^-, so we had to create our own algorithm to do so. The code used to carry out this experiment can be found in the Appendix Section A.1 (Fig. A.4).

5.4.2 The Complexity from The Uniform Digraph Distribution

In spite of the results obtained in the last experiment, they still not being so interesting if we wish to characterize not only the complexity of digraphs but also the complexity of the topology of Random Boolean Networks. In this model, both parameter n and d^- remain fixed, so we would like to also maintain fixed these parameters and see the behavior in the complexity of random directed graphs in this way. However, once we maintain fix these parameters, we will not be able to control the complexity of the networks, so the results just will fluctuate, and it will be impossible to infer anything about its behavior. To solve this problem, we designed the following approach.

We will generate random digraphs in the same exact way we did before, but now with both parameters, n (number of nodes) and d^- (vertex in-degree) maintained fixed. The complexity of them will be measured using the exact same procedure we did in the former experiment. Then, once we have the results, they will be ordered in increasing order of complexity and some of the digraphs will be drawn. In this way, it should be possible to visualize characteristics which we expect make the digraphs to be complex or not, like symmetries or regularities.

Hence, the purpose of this experiment is to verify that the measurements of complexity of digraphs that we perform agree with our visual intuition of what should

be considered as complex or not, especially in the extreme cases, just as we easily can say that the sequence 100101011 is more complex than the sequence 111110000.

The result of this experiment is shown in Fig. 5.5 and the code used can be found in the Appendix Section A.1 (Fig. A.5). As can be seen in Fig. 5.5a, the complexity for random digraphs with fixed parameters n and d^- was bounded and did not vary much with the methods of entropy and K-complexity with an adjacency matrix representation. The method which again seemed to have better performance is the K-complexity with our 1-dimensional representation since it captured a much rich amount of complexity values, which means it can detect slight changes in complexity which the other methods cannot. This confirmed our previous observation that this method is better to describe the complexity of digraphs.

Since the measure of complexity with the K-complexity method and a 1-D representation increases gradually, it should be possible to reaffirm the difference between the complexity of extreme cases by using the visual representation, although due to the low variation in the values this could be no possible to do for intermediate cases, where the slope of the curve is not so high. This is exactly what is shown in Fig. 5.5b. For digraphs with intermediate complexity, it is hard to try to guess which one is more complex, however, for extreme cases, the situation is not as expected. For example, between the most complex and the less complex digraphs shown in Fig. 5.5b, the difference in complexity is yet not clear, even though their computed complexities say there is a big difference.

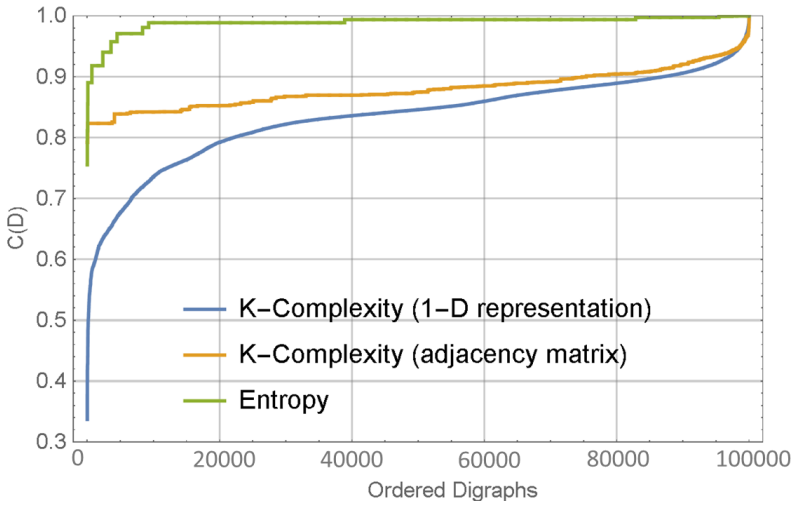
A possible explanation for the above results is that we are not considering isomorphisms when measuring the complexity of the digraphs, i.e., the complexity of the digraph could be affected by the order the nodes are labeled since labeling the nodes in a different order gives a different representation of the object. Hence, it is possible that measuring the complexity of random digraphs is not enough and it is also needed to consider the complexity of its isomorphisms. In the next section, we will try to answer this question.

5.5 The Complexity of Isomorphic Networks

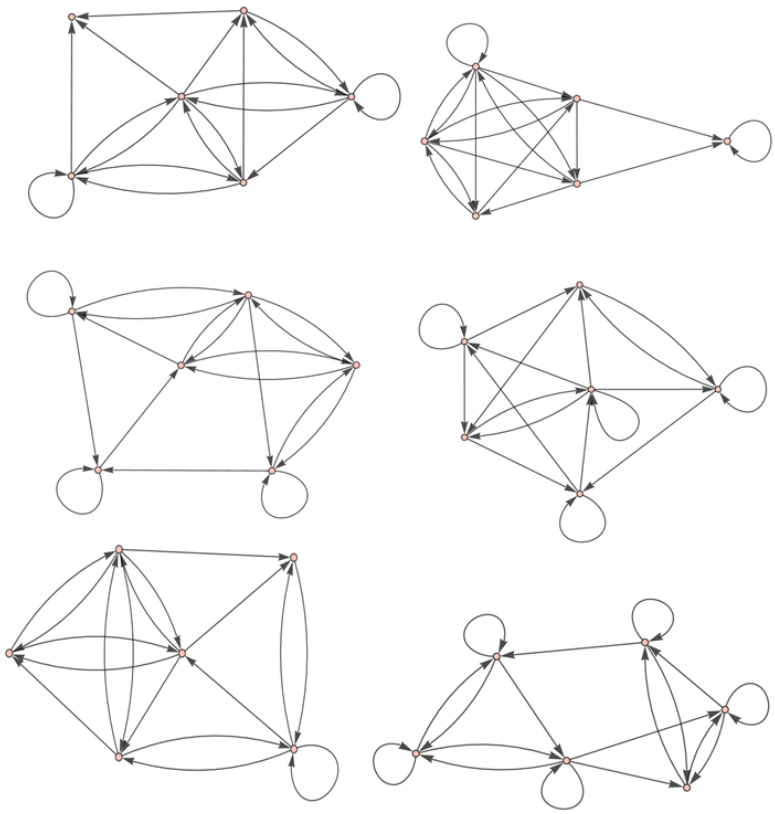
In this section, we will try to answer whether for the same graph or digraph the complexity measurement depends on the isomorphic representation used. First, an experiment using graphs will be performed followed by an experiment using digraphs.

5.5.1 The Complexity of Isomorphic Graphs

The experiment is quite simple. A random graph with a given number of nodes n and a given number of edges k will be generated by means of a uniform graph distribution which is easily implemented in Mathematica by means of the integrated function *UniformGraphDistribution*. Then, the rows and columns of the adjacency matrix of the randomly generated graph will be permuted such that this process corresponds to a reorder in the labeling of the nodes, e.g., if the nodes of a graph are labeled as $\{1, 2, 3, 4\}$ this process will permute the labels to a different order as can be $\{3, 4, 1, 2\}$ or $\{4, 3, 2, 1\}$. Clearly, the number of possible permutations in the

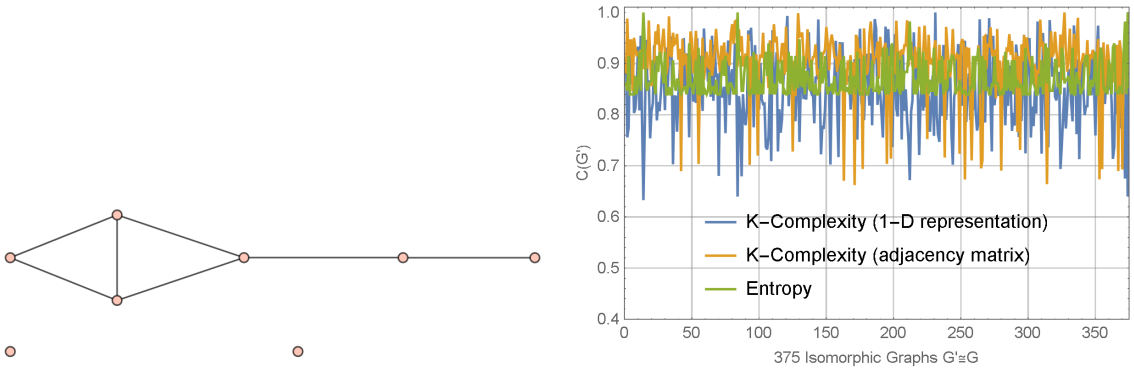


(a)



(b)

Figure 5.5: (a) Complexity measurements for random digraphs. The results were ordered by increasing complexity value. The number of random digraphs generated was $100,000$ and all they had 6 nodes and vertex in-degree 3. For purposes of comparison, the results obtained with each method were normalized with respect to the maximum value. (b) Some directed graphs plotted by increasing complexity value. The complexity increases from left to right and from up to down. The order was obtained from the complexity measurements from the K-complexity method using a 1-D representation. The digraphs plotted were chosen from regular intervals of complexity, such that the digraph at the upper left corner is the less complex and the digraph at the lower right corner is the most complex digraph generated in the experiment.



(a) **(b)**

Figure 5.6: Complexity measurements for isomorphic graphs. (a) The random graph of 8 nodes and 7 edges for which the isomorphisms were generated. (b)Results obtained from measuring the complexity of the same graph with different isomorphic representations. The number of random permutations in the labels of the nodes used to generate the isomorphisms was 400, but only the 375 which were not repeated were considered. For purposes of comparison, the results obtained with each method were normalized with respect to the maximum value.

labeling order of the n nodes is $n!$. Thus, as the number of permutations increases fast for large networks, in general, it will not possible to use all the possible permutations. Therefore, we will work with a fixed number of permutations randomly chosen from a uniform distribution containing all the possible permutations. If two or more permutations are repeated, we will simply ignore these repetitions. To measure the complexity of the isomorphisms we will use the same three methods which we have been using lately.

The result of this experiment is shown in Fig. 5.6 and the code used can be found in the Appendix Section A.1 (Fig. A.6). As can be seen, the complexity of the same graph fluctuated depending on the order we labeled the nodes, i.e., which isomorphism was used to measure the complexity. This result is astonishing and at first, it was not expected since the Kolmogorov complexity had shown to be robust to different representations (see [69]). However, this experiment proved that it is not robust for isomorphic graphs. Therefore, when trying to measure the complexity of a graph it is imperative to also consider its isomorphisms. According to the ideas of algorithmic complexity theory, which we reviewed in Chapter 4, the true Kolmogorov complexity must be the complexity obtained with the representation which gives the minimum value. Evidently, the Shannon entropy method also is subjected to this problem.

As was mentioned, for large graphs with many nodes, the number of isomorphisms increases fast and becomes impractical to try to find the representation with the minimum value of complexity. Therefore, we would like to know if given a graph, is there a complexity value which is more probable to measure? Or in other words, we are interested in knowing the distribution of complexities for the isomorphisms of a given graph. Thus, we used the results of the experiment in Fig. 5.6b and with the help of the built-in function *FindDistribution* of Mathematica we found the

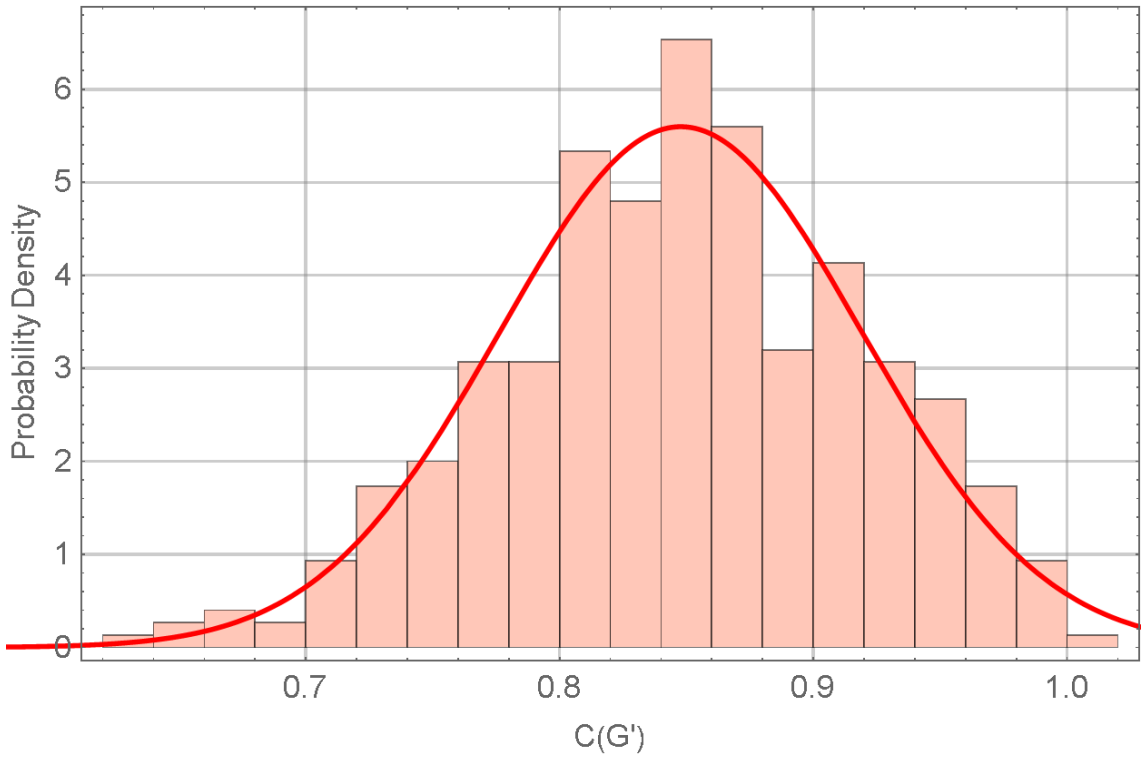


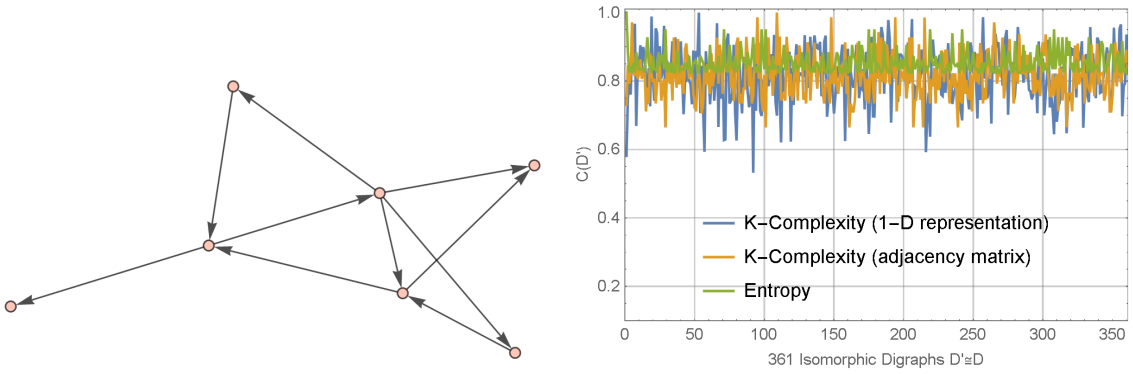
Figure 5.7: The probability distribution which best fits the data shown in Fig. 5.6b (K-complexity of the 1-D representation). The distribution found is a normal distribution with parameters $\mu = 0.847$ and $\sigma = 0.071$.

probability distribution which best fitted the data. The distribution fitted is shown in Fig. 5.7. This distribution is a normal distribution with parameters $\mu = 0.847$ and $\sigma = 0.071$. Therefore, the most probable value for the complexity of the graph shown in Fig. 5.6a when choosing a random isomorphism was $\mu = 0.847$ and the probability of measuring a complexity value in the interval $[-\sigma+\mu, \sigma+\mu]$ was 68.2%.

5.5.2 The Complexity of Isomorphic Digraphs

Now, the former experiment will be repeated, but this time instead of a random graph, a random digraph of n nodes and k edges will be generated. The process to generate it will be again through the built-in function *Random Graph* of Mathematica, but this time the option *DirectedEdges* $\rightarrow$ *True* will be used to indicate that the k edges must be directed. Then, the same procedure used before will be used to generate the isomorphisms, i.e., a fixed number of random permutations will be used to permute the order of the labels of the nodes. Finally, the complexity of the isomorphisms will be measured by using the three methods we have been implementing.

The result of this experiment is shown in Fig. 5.8 and the code used can be found in the Appendix Section A.1 (Fig. A.6). The results showed, as they did for graphs, that the complexity of the digraphs also depends on the isomorphic representation used to measure its complexity. The fluctuations indicated that there are isomorphisms which representation gives a low complexity value while others give a high complexity value no matter what method was used to measure the complexity.



(a) **(b)**

Figure 5.8: Complexity measurements for isomorphic digraphs. (a) The random digraph of 7 nodes and 10 directed edges for which the isomorphisms were generated. (b)Results obtained from measuring the complexity of the same digraph with different isomorphic representations. The number of random permutations in the labels of the nodes used to generate the isomorphisms was 500, but only the 361 which were not repeated were considered. For purposes of comparison, the results obtained with each method were normalized with respect to the maximum value.

Therefore, following the ideas of algorithmic complexity, the true complexity value must be that obtained with the isomorphic representation which gave the lower value.

Besides, as was did for isomorphic graphs, the distribution of complexities obtained using different isomorphic representations of the same digraph was calculated. The distribution which best fits the data in Fig. 5.8b was found using the function *FindDistribution* of Mathematica. This distribution is shown in Fig. 5.9. Once more the distribution which best fitted the distribution of the complexities obtained by means of isomorphic representations was a normal distribution. This time the parameters of the distribution were $\mu = 0.816$ and $\sigma = 0.085$. Thus, the most probable value for the complexity of the digraph shown in Fig. 5.8a when choosing a random isomorphic representation was $\mu = 0.816$ and evidently the probability of measuring a complexity value in the interval $[-\sigma + \mu, \sigma + \mu]$ again was 68.2% since this is a property of the normal distribution.

From this experiment and the former experiment with isomorphic graphs, we can conclude that the complexity measurement of a network depends on the specific isomorphic representation chosen to describe it. According to the ideas of algorithmic complexity, the true complexity of the network is the value obtained with representation which gives the lower complexity. Nonetheless, the number of isomorphisms increases fast with the number of nodes, so sometimes it will be impossible to tests all the isomorphism to find this true distribution. Fortunately, the distributions found in Figs. 5.7 and 5.9 are unimodal and show that there is a value of the complexity which is most probable to get when using any isomorphism and the probability of getting another value is not so high. Hence, even if not all the isomorphisms can be tested, we can use the most probable value of the distribution as an approximation of the complexity when comparing the complexities of many networks.

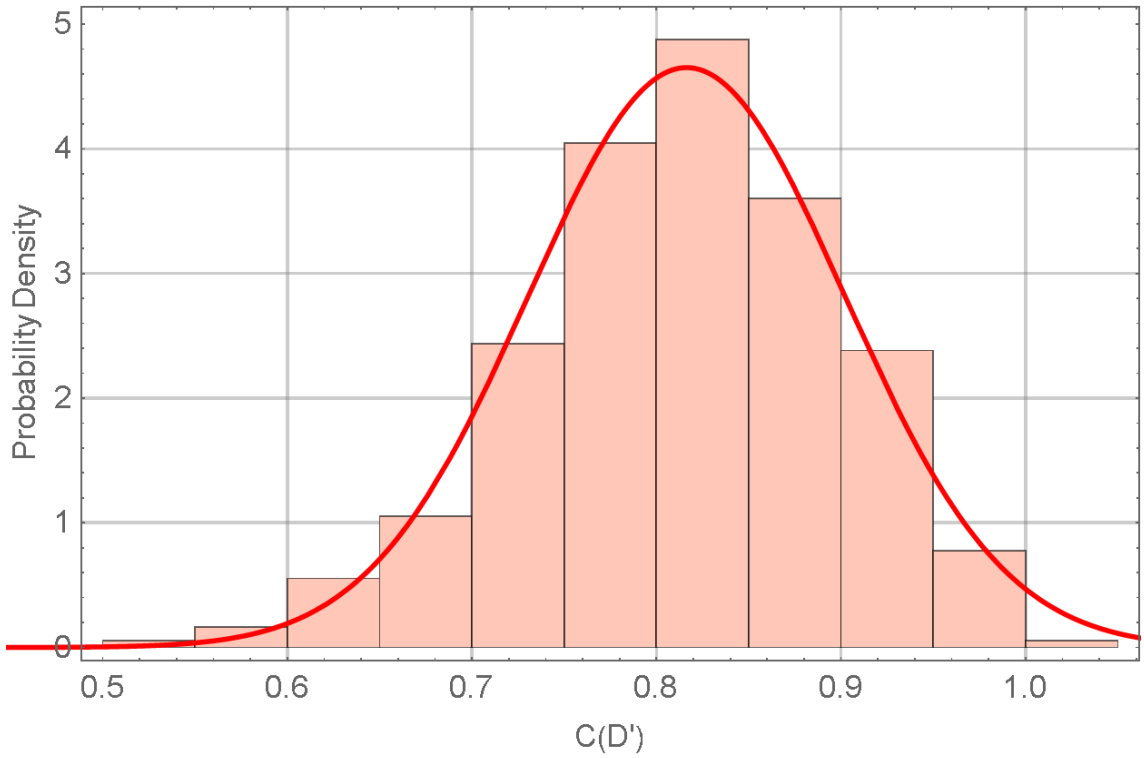
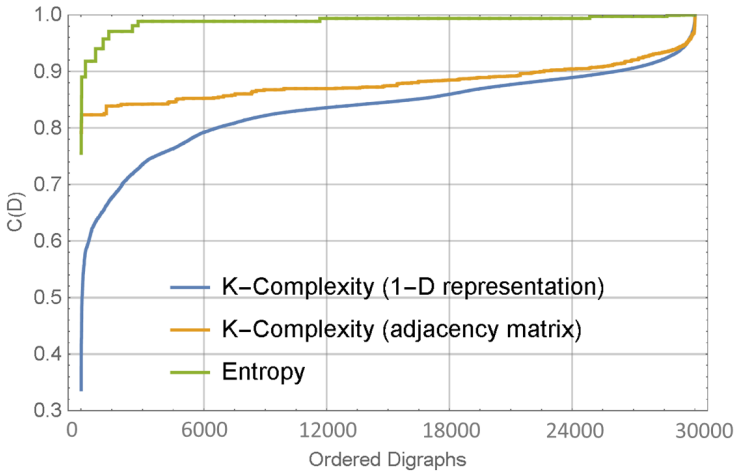


Figure 5.9: The probability distribution which best fits the data shown in Fig. 5.8b (K-complexity of the 1-D representation). The distribution found is a normal distribution with parameters $\mu = 0.816$ and $\sigma = 0.085$.

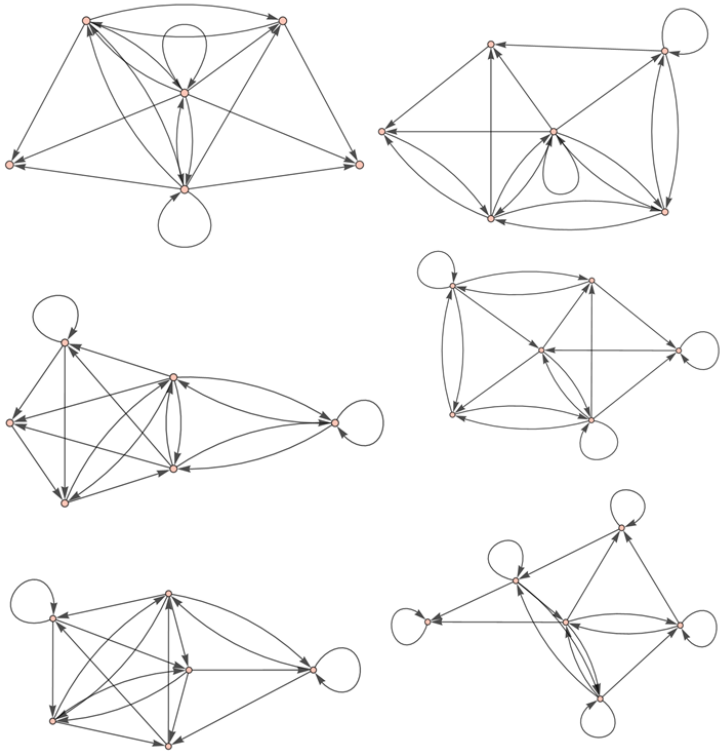
5.5.3 The Complexity from The Uniform Digraph Distribution Revisited by Considering Isomorphisms

Coming back to the experiment about the complexity of random digraphs from the uniform digraph distribution (section 5.4.2), we found in Fig. 5.5 that the random digraphs ordered by increasing value of complexity do not seem to show the same complexity increase when they are drawn. We expected a clear visual difference between the most complex digraph and the less complex digraph; however, this did not happen. That result took us to think we should consider the isomorphisms when measuring the complexity of digraphs. Now we have shown that this idea was correct, thus, the experiment of Section 5.4.2 will be repeated, but now every time a random digraph is generated, also some of its isomorphisms will be generated and its complexity will be computed. As was argued before, the true complexity of the digraph must be the lower value obtained among the complexity values of the isomorphisms, so this criterion will be applied with the three methods to measure the complexity we have been using.

The result of this experiment is shown in Fig. 5.10 and the code used can be seen in the Appendix Section A.1 (Fig. A.7). As can be seen in Fig. 5.10a the complexity measurements showed the same behavior as in Fig. 5.5a. This is normal since the digraphs generated had the same number of nodes and the same vertex in-degree. Nevertheless, in Fig. 5.10b it is possible to note a slightly different result. This time it is easier and clearer to visualize from the drawn digraphs, that the less complex digraph is indeed less complex than the more complex digraph. The less



(a)



(b)

Figure 5.10: (a) Complexity measurements for random digraphs. The results were ordered by increasing complexity value. The number of random digraphs generated was 30,000 and all they had 6 nodes and vertex in-degree 3. The number of random permutations in the labels of the nodes used at each iteration to generate the isomorphisms was 1000, but only which resulted not repeated were considered. For purposes of comparison, the results obtained with each method were normalized with respect to the maximum value. (b) Some directed graphs plotted by increasing complexity value. The complexity increases from left to right and from up to down. The order was obtained from the complexity measurements from the K-complexity method using a 1-D representation. The digraphs plotted were chosen from regular intervals of complexity, such that the digraph at the upper left corner is the less complex and the digraph at the lower right corner is the most complex digraph generated in the experiment.

complex digraph seems to be more symmetrical while the more complex digraph is totally asymmetrical. Thus, in this experiment, we have reconciled our intuition of how a complex digraph should look like and how a digraph which is not so complex should look like.

The results of this experiment are important since any complexity measurement must agree with our intuition about what should be considered complex otherwise the method would be useless. We saw in Section 5.2 how the measurements of complexity obtained with several methods agree with our intuition about the complexity of binary sequences, however, now we have seen that they also agree with our intuition about the complexity of digraphs. This result is what we needed to be sure the methods we have been using give results which are correct, i.e., they correctly measure the complexity of digraphs. This approach to verify our measurements was not needed previously (sections 5.3 and 5.4.1) since there we had a parameter which controlled the complexity of the networks, so we only had to ensure the agreement between the complexity measurements and this parameter. Nevertheless, our findings about the complexity of isomorphisms would not have been possible without this experiment since, until the experiment for the complexity from the uniform digraph distribution (section 5.4.2), the results seemed to be consistent.

5.6 The Complexity of a Set Boolean Functions

As indicated before, a Random Boolean Network is defined by its topology and its updating functions. In the last section, we devoted to learned how the complexity of the topology can be measured. Now, in this section, we will try to establish a systematic way to measure the complexity of the updating functions.

As it can be remembered, in a Boolean Network of N nodes, we assign to each node a logic function of k inputs. We denote this set of Boolean functions belonging to a Boolean Network as $f = \{f_1, f_2, ..., f_N\}$.

Here, we propose the following method to measure the complexity of a set of Boolean functions. First, it must be remembered that the behavior of each Boolean function f_i can be described by its truth table. Hence, the behavior of a set of Boolean functions can be described by a set of truth tables. From this set of Boolean functions, we can build a unique truth table as a representation of all of them. As this unique truth table contains all the information about the set of Boolean functions, a measurement of its complexity equals to measure the complexity of the set of Boolean functions. Finally, this unique truth table is a matrix which can be used as a lossless representation of the set of Boolean functions.

For instance, in Section 3.1.6, we saw an example of Random Boolean Network with the assigned logic functions we reproduce again in Table 5.1. From this truth table, we can build the representation shown in Fig. 5.11. This representation is a matrix with dimensions $2^k \times N$, thus we can apply to it the same methods we have used to measure the complexity of adjacency matrices in sections 5.3 and 5.4.

In the next sections, two experiments will be performed to check out if this

σ_1 σ_2	f_{Node1}	f_{Node2}	f_{Node3}	f_{Node4}
0 0	1	1	0	1
0 1	1	1	1	1
1 0	0	0	1	0
1 1	0	1	0	0

Table 5.1: Truth table with the set of Boolean functions assigned to a Boolean Network of 4 nodes and parameter $k = 2$ used to write the matrix representation of the set of updating functions.

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 5.11: Matrix representation of the truth table shown in 5.1.

representation of the set of Boolean functions of a Boolean Network delivers results which makes sense when measuring its complexity.

5.6.1 The Complexity from Sets of Random Boolean Functions with Increasing Number of Inputs

This first experiment is inspired by the experiments of sections 5.3.1, 5.3.2 and 5.4.1. In those experiments, we had a parameter which controlled the complexity of the random networks generated at each iteration. In this way, we were able to confirm that our measurements of complexity had the behavior expected according to the value of this parameter.

Similarly, in this experiment, at each iteration, a set of N random Boolean functions with k inputs will be generated. The logic functions will be randomly chosen from the uniform probability distribution of the 2^{2^k} possible Boolean functions with k inputs. Thus, the number of possible sets of random functions is given by $2^{N\times 2^k}$ as was shown in Eq. 3.8.

Since in any Random Boolean Network model, the number of nodes must remain fixed throughout the dynamics. In this experiment, the parameter N also will remain fixed and the complexity of the randomly generated sets of boolean functions will be controlled by the parameter k.

At each iteration, the matrix representation for each set of logic functions randomly generated will be built and used to measure its complexity. As was usual in the previous sections, the K-complexity will be directly measured from the matrix representation[2], but also a 1-D lossless representation will be created from this

[2]The method we have been using was originally created to measure the complexity of adjacency matrices, i.e., square matrices, nevertheless thanks to the BDM Method, it can be also used to measure the complexity of non-square matrices.

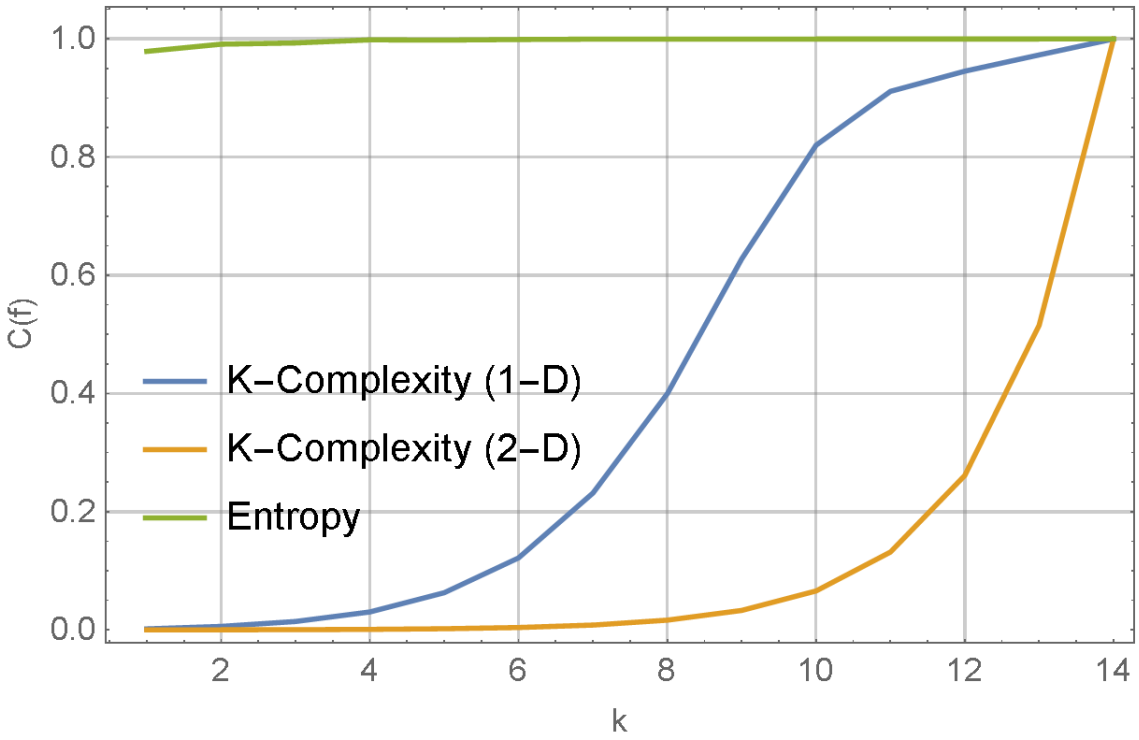


Figure 5.12: Complexity measurements using three different methods of random sets of Boolean functions obtained from a uniform distribution versus the parameter k (the number of inputs). The sets generated had 10 Boolean functions each one ($N = 10$) and the parameter k was vary from 1 to 14. The number of sets of random logic functions generated with the same k value was 10 from which the trimmed mean described before was done. For purposes of comparison, the results obtained with each method were normalized with respect to the maximum value.

matrix to measure the entropy and the 1-dimensional K-complexity. Moreover, as was did in sections 5.3.1, 5.3.2 and 5.4.1, the fluctuations in the results with a given value of k will be reduced by means of a trimmed mean which will be made by generating a given number of sets with the same parameter value, for later compute the quartiles of the results and finally compute the mean value by considering only the second and third quartiles. The result of this procedure will be considered the complexity value we are looking for. At the end of the iteration, the parameter k will be increased and the algorithm will be repeated.

As the value of the parameter k increases, we expect that the complexity of the set of Boolean functions will also increase since the information needed to reproduce the behavior of a Boolean function with larger k value is greater than for a shorter k.

The result of this experiment is shown in Fig. 5.12. The performance of the entropy method was awful, showing almost a constant behavior. On the other hand, the results with the K-complexity showed that the complexity increased when the parameter k increased just as was expected. Nevertheless, the behavior of the Kolmogorov complexity differed depending on which representation was used. The results from the 2-D representation (the matrix representation) seemed to be almost constant from $k = 1$ to $k = 8$. Meanwhile, the results from the 1-D representation

showed a gradual increase in all the range experimented. From these observations, we could be tempted to conclude that the 1-D representation method is better to measure the complexity of a set of Boolean functions. However, the range used is limited and does not allow to infer the behavior for a larger k value, especially the behavior of the 1-D representation. Though, since the complexity should continue increasing it is almost sure that the 1-D representation will continue to be the best method for larger k values.

The code used to generate sets of random Boolean functions and measure its complexity is shown in the Appendix Section A.1 (Fig. A.8). Maybe the most interesting part of the code is the generation of random Boolean functions. One possible approach could have been to generate a matrix with dimensions $2^k \times N$ and initialize it with zeros. Then, given a parameter p, we could have changed the $0's$ by $1's$ on each column of the matrix (a column corresponds to a logic function). This procedure would be ideal if we were interested in the generalization of the original Kauffman model for a Random Boolean Network discussed in Section 3.1.4.2. However, we are interested only in the original model proposed by Kauffman where for each node a Boolean function is randomly chosen within all the universe of possible functions with k inputs, and where all the functions have the same probability to be chosen, i.e., the probability distribution is uniform. This random choice is easily implemented in Mathematica by means of the Built-in function *BooleanFunction* which immediately allows accessing to all the 2^{2^k} possible Boolean functions with k inputs. Then, the Boolean function chosen is represented as a combination of one of the universal Boolean functions defined in 3.4 by means of the built-in function *BooleanConvert* and finally, the truth table is obtained by means of the built-in function *Boole*. The truth tables are joined as described before, and in this way, the matrix representation is built.

5.6.2 The Complexity from Sets of Random Boolean Functions with Fixed Number of Inputs

As happened before, the experiment with increasing parameter k is not so interesting since in the CRBN model the parameter k also remains fixed throughout the dynamics. Therefore, it desirable to perform an experiment where both parameter N and k could remain fixed. However, if these parameters are fixed, there will be no way to control the complexity of the sets of Boolean functions generated. This is the same problem we faced in sections 5.4.2 and 5.5.3. Thereby, the same approach used in those sections will be used to measure the complexity of sets of random logic functions.

The sets of random Boolean functions will be generated as in the last section, but this time with both parameters k and N fixed. For each set, the matrix representation will be built and from it, the complexity will be measured by means of the three methods we used in the former experiments. From the sections 5.4.2 and 5.5.3 we learned that the complexity of an object depends on the isomorphic representation used. Thus, it will be necessary to measure again the complexity using different isomorphic representations and use the minimum value obtained as the true value of the complexity. An isomorphic representation of our matrix representation is easily

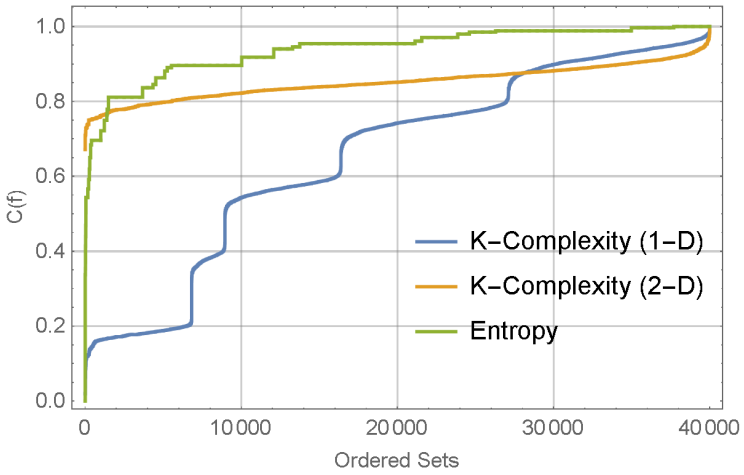
obtained by permuting its rows. This permutation corresponds to a reorder on the input states [3]. For instance, the truth table in Table 5.1 presented the input states in a *canonical order*: $\{00, 01, 10, 11\}$. Nevertheless, instead, we could have used the order $\{11, 10, 01, 00\}$ or any other, i.e., any other isomorphic representation. The number of possible permutations of the rows in our matrix representation with dimensions $2^k \times N$ is $2^k!$. Therefore, it could be computationally expensive to try all the possible permutations every time a set of random Boolean functions is generated. Thus, in order to measure its complexity, just a given number of permutations chosen randomly (with a uniform probability distribution) will be considered each time a set of random Boolean functions is generated.

Once the complexities of the isomorphic representations are measured, the minimum value obtained will be considered to be the complexity of the original matrix representation, i.e., the complexity of the set of Boolean functions. This procedure will be repeated a given number of times and the results will be ordered from the less to the greater complexity value. As happened in sections 5.4.2 and 5.5.3, we need a method to check out if the measurements of complexity we are getting make sense. In those sections, we relied upon a visual approach by drawing the digraphs to confirm that the extreme cases show a clear difference in complexity. In this experiment, as visual confirmation of the results, the matrix representation of every random set of Boolean functions will be plotted. To do so, we will take advantage of the fact that the matrix representation is built only by $0's$ and $1's$. In this way, the matrix elements will be represented in a grid where a square will be colored black if the corresponding matrix element is 1, otherwise, if it is 0, it will be colored white. This plotting is a visual representation of the set of Boolean functions since all the rules are contained within it.

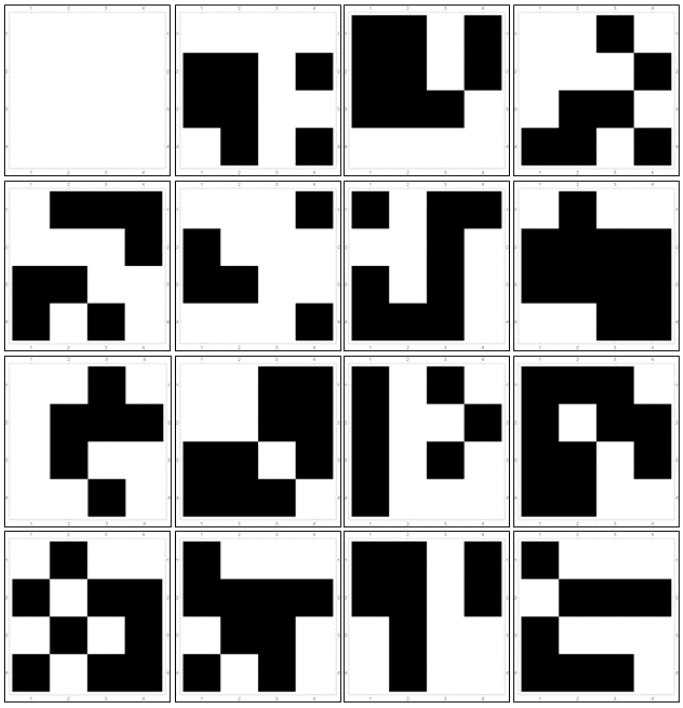
It is expected that the less complex set of Boolean functions will be the set made up only by functions which deliver always a zero(one) for all the possible input states. Thus, its matrix representation is full of zeros(ones). On the other hand, the most complex set of Boolean functions must be a set where there is no apparent order in the distribution of $0's$ and $1's$.

The result of this experiment is shown in Fig. 5.13. As can be seen in Fig. 5.13a, it seems that the three methods are able to measure a gradual increase in the complexity of the sets of Boolean functions. The results of the K-complexity with a 1-D representation showed a stairway like behavior meanwhile the 2-D representation showed a smoother curve. Surprisingly, the entropy method this time did not show a slope almost equal to zero as it did in the former experiments. Nonetheless, the range of complexity values shown by the entropy and the K-complexity with the 2-D representation was very limited when compared with the range of complexity values computed with the method of K-complexity and the 1-D representation, which means that this last method was able to capture with more detail the changes in

[3] An alternative would be to consider the permutations on the columns, which corresponds to a reorder on the labels of the functions, i.e., a reassignment of the functions to the nodes. Nonetheless, if we were studying the dynamics of an RBN, this type of permutations would give a Boolean Network with a completely brand-new dynamics and thus the complexity measurements would correspond to different objects. Hence, even though we are not considering the dynamics of a Boolean function, this type of permutations will not be considered.



(a)



(b)

Figure 5.13: (a) Complexity measurements for sets of random Boolean functions. The results were sorted by increasing complexity value. The number of random sets generated was $40,000$ and all they had 4 functions ($N = 4$) each one with $k = 2$ inputs. At each iteration, the number of randomly generated permutations of the rows in the matrix representation was 16, but only the permutations which resulted not repeated were considered. For purposes of comparison, the results obtained with each method were normalized with respect to the maximum value. (b) Some sets of Boolean functions plotted by increasing complexity value. The complexity increases from left to right and from up to down. The order was obtained from the complexity measurements from the K-complexity method using the 1-D representation. The sets of logic functions plotted were chosen from regular intervals of complexity, such that the set at the upper left corner is the less complex and the set at the lower right corner is the most complex set of Boolean functions generated in the experiment. Each set is represented by a square whose columns represent the logic functions and its rows represent the possible input states.

complexity on the sets of Boolean functions. Thus, this method proved again to be the best, this time by measuring the complexity of sets of Boolean functions.

Meanwhile, the results in Fig. 5.13b showed as expected, that the less complex set of Boolean functions is the set whose matrix representation is full of zeros. In general, among two sets of Boolean functions, it is hard to say which one is more complex just from the visual representation. However, the correct prediction of the less complex set of Boolean functions gives confidence in that the method to compute the complexity is working as it should so we do not have to depend on a visual representation anymore.

The code used to generate sets of random Boolean functions and measure its complexity is shown in the Appendix Section A.1 (Fig. A.9). It is like the code shown in Fig. A.8, but this time the parameter k remains fixed and the complexity is measured with the help of the isomorphic representations as was described before.

5.7 The Complexity of Boolean Networks

In the last sections, we already have studied the complexity of the components of Boolean Networks, i.e., the topology and the updating functions, though we have studied them separately. Nonetheless, measuring the complexity of the components of a Boolean Network separately and then simply adding them up should not be equal to simply measuring the complexity of a Boolean Network when it acts as an ensemble. The interaction between the topology and the updating functions is so intricate that a measurement of these components separately must be a poor approximation to the real complexity of the system. Of course, it is expected that a topology which is simple will produce a Boolean Network which also is simple and the same should happen for the Boolean functions, indeed we are going to investigate further this hypothesis lately, but the question of how to measure the complexity of a Boolean Network still exists.

We will try to answer this question by appealing to Definition 3.2 of a Boolean Network. In this definition a Boolean Network is seen as a logic function which maps $f : \{0,1\}^N \to \{0,1\}^N$ and where N is the number of nodes. Therefore, if we wish to measure the complexity of a Boolean Network, we must measure the complexity of its behavior when acting as a function, i.e., we must measure the complexity of the mapping produced by it. The mapping produced by the Boolean Network contains all the information about its topology and its updating functions, moreover it contains the information of these constituents when acting together, so its complexity must correspond to the complexity of the Boolean Network.

5.7.1 A Lossless Representation for Boolean Networks

Following the previous arguments, in this section, a lossless representation of the mapping produced by a Boolean Network is presented. The method will be introduced through an example.

In Table 3.10 and Fig. 3.2 were shown respectively, the updating functions and the topology of a Boolean Network. Meanwhile, in Fig. 3.3 was shown the state-space corresponding to this Boolean Network. This state-space represents the dynamics of the system, i.e., the mapping of the Boolean Network since it says to what state the system should move given any input state. From this state space, the Table 5.2 can be built. This table represents the mapping made by the Boolean Network.

Input State	Output State
0000	1101
0001	1111
0010	1101
0011	1111
0100	1101
0101	1111
0110	1101
0111	1111
1000	0010
1001	0100
1010	0010
1011	0100
1100	0010
1101	0100
1110	0010
1111	0100

Table 5.2: The mapping produced by the Boolean Network presented in Table 3.10 and Figs. 3.2 and 3.3. The input states were placed in canonical order by considering them as binary numbers.

A lossless representation of the mapping produced by the Boolean Network can be built just by putting each output state one after the other in the same order shown in Table 5.2:

S=1101111111011111110111111101111100100100001001000010010000100100

This binary sequence is a lossless representation of the mapping produced by the system and its complexity should correspond as was argued before to the complexity of the Boolean Network [4]. Furthermore, since it is a binary sequence, we can measure its complexity by using the same methods of Section 5.2.

Now that we have a way to represent a Boolean Network as a simple binary sequence and thus a way to measure its complexity, some experiments will be performed in the following sections concerning the complexity of Random Boolean Networks.

[4]It is not necessary to consider the input states in the representation of the Boolean network because its effect only would be to add a constant to the complexity of the Boolean Networks.

5.7.2 The Correlation Between the Complexity of Random Boolean Networks and The Complexity of Its Topology

If we have a given set of Boolean functions and randomly choose a topology for this set, in order to create an RBN, every different topology chosen will make the RBN to have a different complexity value. It is expected that a topology which is simple will render a Boolean Network which also is simple and in the same way, a topology which is complex should produce a Boolean Network which is complex as well. In other words, there should exist a positive correlation between the complexity of a Boolean Network and the complexity of its topology. In this section, this ansatz will be tried to be verified. To do so, the next procedure is used.

Firstly, a random topology is chosen following the classical Random Boolean Network model with a given number of nodes and vertex in-degree (see Section 3.1.3). Then, a given number of random sets of updating functions is chosen following the classical Kauffman model as well (see Section 3.1.4.2). Thereupon, these sets of Boolean functions are assigned to the topology generated before, which means we are creating RBNs with the same topology, but with different updating functions. The dynamics of the RBN is generated by introducing to it all the possible input states and from this, the lossless representation discussed before is created for each RBN and its complexity is measured by using the K-complexity implementation, as was used in Section 5.2 to measure the complexity of random binary sequences. From the complexities obtained for the RBNs, the quartiles of this set of results were calculated, and by only considering the second and third quartiles, the mean complexity was computed, i.e., we computed a trimmed mean. Therefore, we were interested in knowing the mean complexity of RBNs which share the same topology and for which only the updating functions were varied among them.

Afterward, the complexity of the topology is measured by considering some of its different isomorphic representations. In Section 5.5, the true complexity value of the topology was considered to be the lower value obtained with these isomorphic representations. Nevertheless, this time we will consider the mean complexity of the topology from its isomorphic representations and not the minimum value as before. The reason is because we will be measuring the mean complexity of Random Boolean Networks with this same topology, therefore it is desirable to measure also the mean complexity of the topology since it could happen for example, that we assign the same set of updating functions to the topology, but with the functions sorted in different order. That would correspond to Boolean Networks which share the same set of updating functions, but with isomorphic topologies. Therefore, it is better to consider the mean of the complexity of the isomorphic topologies and not just one value (the minimum value).

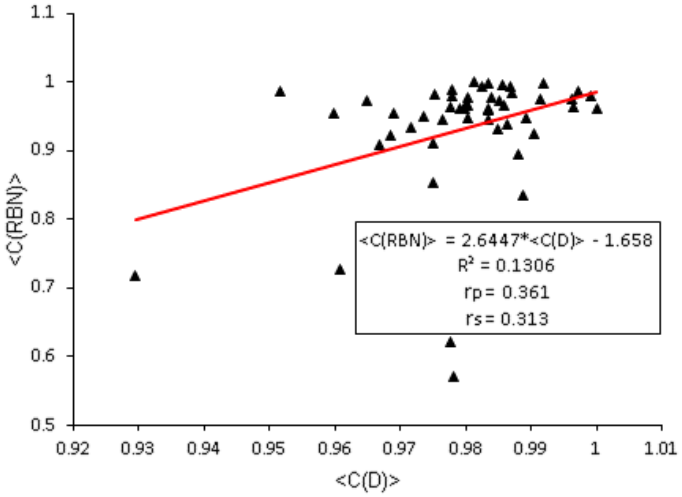
Thereupon, a new topology is randomly chosen, again with the same parameters, and in the same way, new sets of updating functions are randomly generated to create RBNs and measure its mean complexity together with the mean complexity of its topology and so on.

Once we have all the results, they are plotted, and the correlation between the two complexities is evaluated by computing the *Pearson correlation coefficient* to test a possible linear correlation. Additionally, a linear model is fitted to the data. However, since we do not know whether the possible correlation between the variables is linear or not, also the *Spearman's rank correlation coefficient* is computed to evaluate how well the relationship between the two complexities is described by a monotonic function.

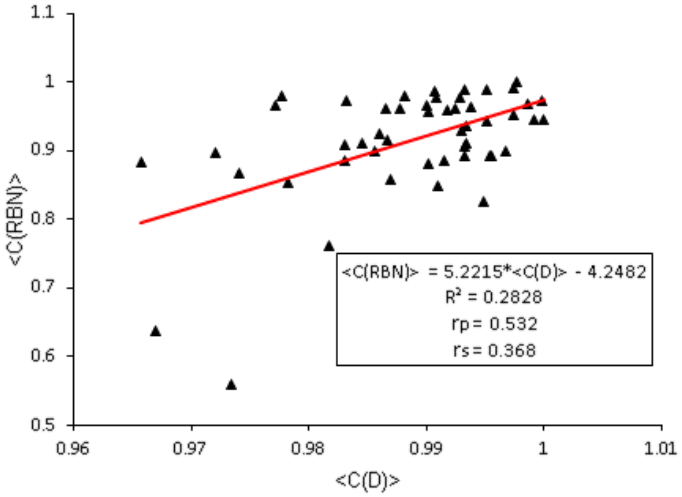
Due to the high computational cost to generate Random Boolean Networks and compute its complexity, this experiment was performed with small topologies, specifically, a topology with five nodes and other with seven nodes were considered. Another reason to consider small topologies is the huge number of possible RBNs with larger topologies. As was said in Section 3.1.5, the number of possible Random Boolean Networks increases so fast with the parameters N and k. Hence, it is hard to perform statistics in this ensemble since larger topologies would make even harder to get enough data to statistically be able to conclude something. For these reasons, we will focus on RBNs with just a few nodes.

The results of two experiments performed with different parameters N and k are shown in Fig. 5.14. As can be seen in both experiments, the data showed a linear relationship between the mean complexity of the RBNs and the mean complexity of the topologies used. The Pearson correlation coefficient computed for the complexities in Fig. 5.14a was $r_P = 0.361$, while the Spearman's rank correlation coefficient computed was $r_S = 0.313$. These values indicated that there exists a correlation between these complexities as was expected. Furthermore, the linear correlation is slightly stronger than a correlation described by a monotonic function. Moreover, the coefficients are positive which means that a greater mean complexity value for the RBN corresponds to greater mean complexity for the topology used. The same happened for the complexities showed in Fig. 5.14b which had a Pearson correlation coefficient of $r_P = 0.532$ and a Spearman's rank correlation coefficient of $r_S = 0.368$. These values were closer to 1 than they were in the first experiment which means that both correlations were stronger. In fact, the Pearson correlation coefficient is greater than 0.5 so we can consider that both complexities showed a strong linear relationship.

According to the previous results, our initial hypothesis was correct, i.e., a simple RBN indicates a simple topology and a complex RBN indicates a complex topology. Furthermore, the correlation seems to be linear. Nonetheless, it seems there exist cases, where this hypothesis is not fulfilled. This can be explained by the fact that the complexity of a Boolean Network also is determined by the updating functions which were chosen and not only by the topology used. To see this, just imagine we assign to a topology very complex the updating functions whose matrix representation is full of zeros, then no matter which input state we evaluate, since the state of the network will always be the same. Thus, this type of updating functions spoils the complexity of the network. This explains why the the Pearson correlation coefficient and the Spearman's rank correlation coefficients found in these experiments are positive but not equal to 1. Unfortunately, this experiment is not enough to say that our hypothesis has been demonstrated empirically for any Boolean Network.



(a)



(b)

Figure 5.14: The mean complexity of the topology versus the mean complexity of Random Boolean Networks with this same topology. Both mean complexities were normalized with respect to its maximum value. A linear model was fitted to the results and the correlation between the variables was studied. (a) All the topologies used had five nodes ($N = 5$) and vertex in-degree 3. The number of RBNs generated with the same topology at each iteration was 5000 and the number of random permutations generated to measure the mean complexity of the topology was 300, though only which resulted not repeated were considered. The model fitted has an intersection with the vertical axis at -1.658 and a slope of 2.6447 with an R^2 coefficient of 0.13. The Pearson correlation coefficient resulted to be $r_P = 0.361$, while the Spearman's rank correlation coefficient resulted to be $r_S = 0.313$. (b) All the topologies used had five nodes ($N = 7$) and vertex in-degree 4. The number of RBNs generated with the same topology at each iteration was 2000 and the number of random permutations generated to measure the mean complexity of the topology was 5000, though only which resulted not repeated were considered. The linear model fitted has an intersection with the vertical axis at -4.2482 and a slope of 5.2215 with an R^2 coefficient of 0.282. The Pearson correlation coefficient resulted to be $r_P = 0.532$, while the Spearman's rank correlation coefficient resulted to be $r_S = 0.368$.

To do so, it it would be necessary to perform more experiments with topologies of distinct sizes. All we can say is that our hypothesis of correlation was correct for Random Boolean Networks with small topologies given an ensemble with a sufficiently large number of Boolean Networks.

The code used in this experiment can be found in the Appendix Section A.1 (Fig. A.10). Almost all the code has been recycled from previous experiments. The only difference is the section which allows testing the output states of the Boolean Network given all the possible input states, i.e., the section which gives the dynamics of the network.

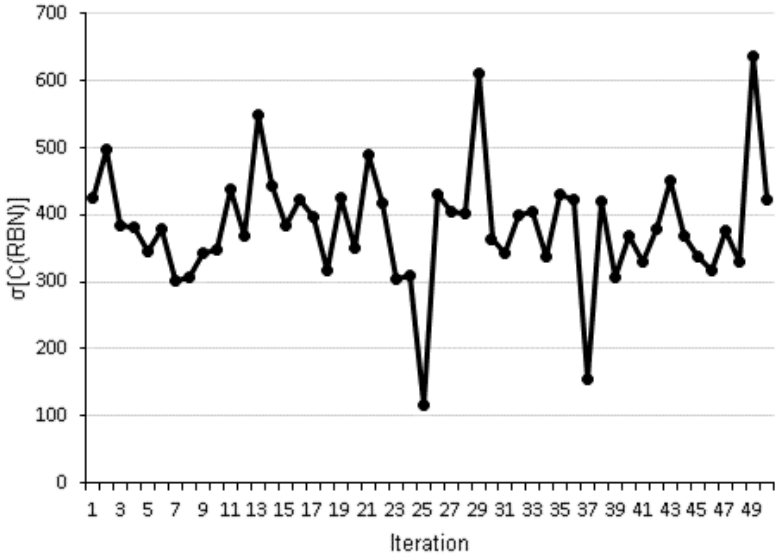
5.7.3 The Width of the Distribution of Complexities for Random Boolean Networks

In the last section, the complexities of Random Boolean Networks which shared the same topology were calculated, and from them, the mean complexity of these RBN was computed. Now, we will study the standard deviation in the complexities from which the mean is calculated, i.e., we will examine the behavior of the width in the distribution of complexities of RBNs with the same topology. Specifically, we are interested in figuring out if this width stays constant or almost constant when the number of nodes is fixed.

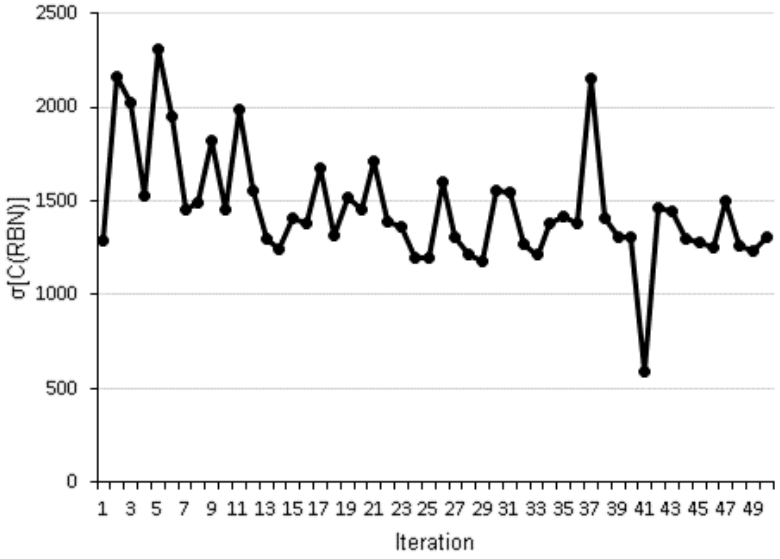
We will use the results of the two experiments performed in the last section. From them, we will compute the standard deviation in the complexities of RBNs which share the same topology and where all the topologies have the same number of nodes and vertex in-degree.

The results of these two experiments are shown in Fig. 5.15. Both experiments confirmed that when the number of nodes is maintained fixed, the width of the distribution of complexities of RBNs with the same topology fluctuates randomly, though it seems to be bounded from above and from below. Given this behavior, it is expected that as the number of nodes increases, the standard deviation also will increase, i.e., the bounds of the fluctuations in the standard deviation will be moved according to the number of nodes.

Therefore, to analyze the behavior of the width in the complexity distribution of Random Boolean Networks against the number of nodes of the topology a modified version of the experiments shown in Fig. 5.14 was implemented. In this version, instead of fixing the number of nodes, at every certain number of iterations, this parameter is increased. The number of nodes starts in 2 and the vertex in-degree remains fixed all the time. Thereupon, a random topology and a random set of updating functions are generated and with them, the dynamics of the network is generated. After, the complexity of the Random Boolean Network is measured as before. Since this occasion we are only interested in the distribution of complexities of the RBNs, it is not necessary to measure the complexity of the topology and the updating functions. Once we have generated and measured the complexity of a certain number of RBNs the number of nodes of the topologies (and the number of updating functions in the set of updating functions) is increased and the procedure



(a)



(b)

Figure 5.15: The standard deviation in the complexities of Random Boolean Networks with fixed number of nodes. (a)RBNs whose topology has 5 nodes and vertex in-degree 3. (b)RBNs whose topology has 7 nodes and vertex in-degree 4.

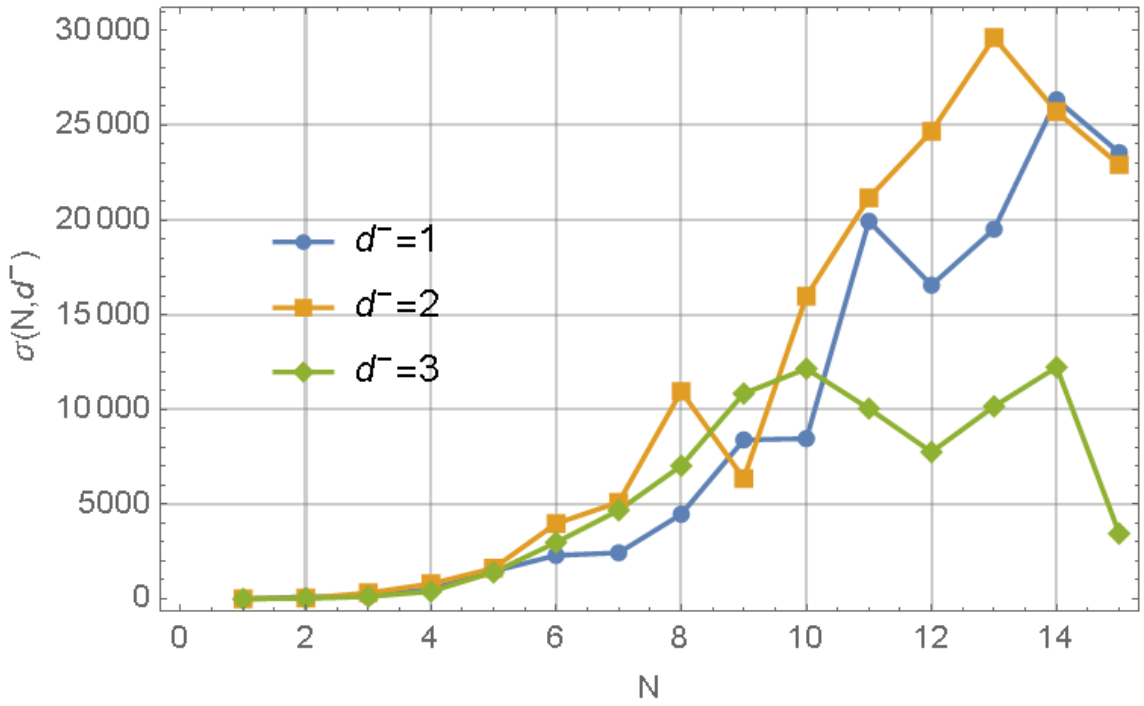


Figure 5.16: The width of the distribution of complexities of Random Boolean Networks versus the number of nodes of the topologies used. The experiment was performed with three different vertex in-degrees: $d^- = 1, 2$ and 3. The number of RBNs generated with the same number of nodes, i.e., the size of the ensembles considered, was 100 for the three parameters. The number of nodes ranged from 2 to 16.

is repeated. The standard deviation is computed for the complexities of each set of Random Boolean Networks which have the same number of nodes.

The result of this experiment is shown in Fig. 5.16. The experiment was implemented with the vertex in-degrees $1, 2$ and 3. In the beginning, the width of the distribution of complexities increased monotonically with the number of nodes, however, it started to fluctuate with larger topologies. These fluctuations could be attributed to the fact that the number of possible RBNs increases fast with the number of nodes, therefore when the number of nodes of the topology was increased, the number of RBNs studied was not enough to keep the increasing of the width. It can be expected that the width will continue increasing with the number of nodes in an unbounded way with a sufficiently large ensemble of Random Boolean Networks. This hypothesis is reinforced by comparing the results with different vertex in-degree values. For topologies with a small number of nodes, the width of the distribution of complexities increased equally and independent of the parameter d^-. The behavior started to depend on the value of the vertex in-degree used, only for larger topologies. Thus, by remembering that the number of possible Random Boolean Networks increases not only with the number of nodes but also with the parameter d^- (Eq. 3.9), and since the size of the ensembles of Random Boolean Networks used was the same for the three values of d^-, it is expected that the fluctuations would be significantly more important for $d^- = 3$ than for the other values since our finite sample is less representative in this case due to the larger number of possible RBNs

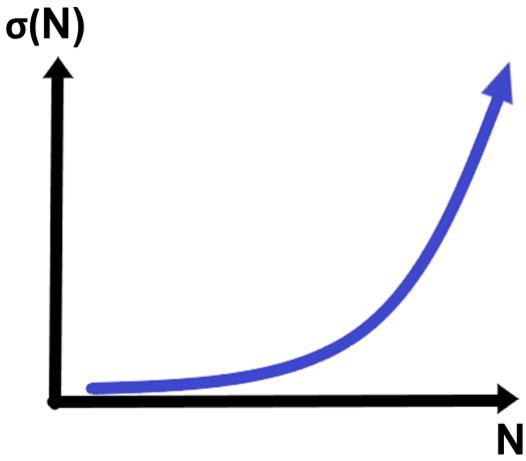


Figure 5.17: The expected behavior of the width of the distribution of complexities of Random Boolean Networks versus the number of nodes of the topologies used for ensembles sufficiently large. The value of σ increases monotonically with the number of nodes and does not depend on the value of the parameter d^-, i.e., $\sigma = \sigma(N)$.

which were not considered. By following the same reasoning, the fluctuations for $d^- = 2$ would have to be more important than for $d^- = 1$. The previous can be checked in Fig. 5.16, as can be seen from the results, when the value $d^- = 3$ was used, the increase in the width was stopped and started to fluctuate earlier than for the other values. Moreover, for $d^- = 2$ the increase was stopped earlier than for $d^- = 1$. Hence, as was mentioned before, we expect that by considering ensembles of Random Boolean Networks sufficiently large, the width of the distribution of complexities will continue increasing monotonically with the number of nodes and independently of the vertex in-degree of the topologies used, i.e., we can expect a monotonically increasing behavior as the one shown in Fig. 5.17.

This experiment can be performed with the code shown in Fig. A.10 or Fig. A.11 (Appendix Section A.1). The only change necessary is to increase the parameter N every determined number of iterations at the beginning of the first iteration loop. For better performance, the parts which measure the complexities of the topology and the updating functions can be ignored.

5.7.4 The Correlation Between the Complexity of Random Boolean Networks and the Complexity of its Set of Updating Functions

In the last sections, the correlation hypothesis between the mean complexity of the RBNs and the mean complexity of its topology was tested. Nonetheless, a Boolean Network is not defined by its topology but also by its updating functions. Hence, it is also expected that if the set of updating functions assigned to the network is simple, then the complexity of the Boolean Network also will be simple. This can be seen by testing the dependence of the variables as was did before.

The experiment is very similar to the one of Section 5.7.2. However, this time a

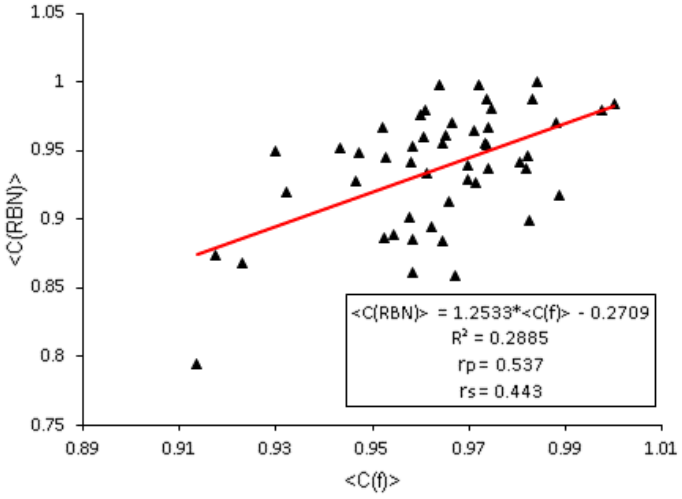
random set of updating functions was generated firstly. Then, some given number of topologies were randomly generated, and we assigned to all of them the same set of updating functions generated before. The complexity of the resulted Random Boolean Networks was measured and from the results, the mean complexity was computed by only considering the second and third quartiles of the data (trimmed mean). For the same reasons argued in Section 5.7.2, the mean complexity of the set of updating functions was computed by considering its isomorphic representations and by discarding the first and fourth quartiles to compute the mean complexity (trimmed mean) instead of considering the minimum value of complexity. Thereupon, another set of updating functions was generated and the procedure was repeated.

Due to the same reasons for the experiment in Section 5.7.2, this experiment was performed with topologies of only a few nodes. To be consistent, it was decided to perform again two experiments with the same parameters used before, i.e., the first experiment uses topologies with five nodes and vertex in-degree 3 and the second experiment uses topologies with seven nodes and vertex in-degree 4.

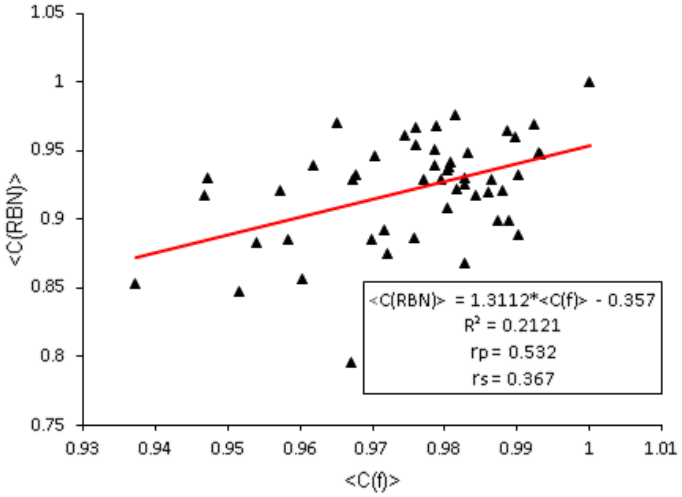
The results of these experiments are shown in Fig. 5.18. In both experiments, the linear relationship between the mean complexities was observed. The linear models fitted showed a positive slope which suggested that the linear correlation between the complexities exists. However, to confirm this we had to check the correlation tests. These dependence tests between the variables also indicated the existence of a correlation which was stronger when it was considered to be a linear correlation. The previous is because for the experiment shown in Fig. 5.18a, the Pearson correlation coefficient computed was $r_P = 0.537$ while the Spearman's rank correlation coefficient computed was $r_S = 0.443$. On the other hand, for the experiment shown in Fig. 5.18b the Pearson correlation coefficient computed was $r_P = 0.460$ while the Spearman's rank correlation computed was $r_S = 0.367$. In both experiments the linear correlation was stronger than a correlation described by a monotonic function, moreover, the Pearson correlation coefficients were around 0.5 which indicated that both complexities are strongly related. Finally, both correlation coefficients were positive which means that when the set of updating functions is complex, the resulted Boolean Network also will be complex.

From these results can be seen that our initial hypothesis was correct. We believe that the correlation coefficients are not closer to 1 because of the presence of Random Boolean Networks whose atypical behavior spoils the correlation as was argued for the experiment in Section 5.7.2. In this case, could happen that even if the set of updating functions is complex, a simple topology can end spoiling the complexity of the network. For instance, a topology whose nodes are insulated and poorly connected can cause the Boolean Network to have simple dynamics even though the complexity of the set of updating functions. Unfortunately, once more these results cannot be generalized to larger topologies and all we can say is that the hypothesis was correct for small Random Boolean Networks given a sufficiently large ensemble of Boolean Networks.

The code used in this experiment is shown in the Appendix Section A.1 (Fig.



(a)



(b)

Figure 5.18: The mean complexity of the updating functions versus the mean complexity of Random Boolean Networks with this same set of updating functions. Both mean complexities were normalized with respect to its maximum value. A linear model was fitted to the results and the correlation between the variables was studied. (a) Each set of updating functions was composed by 5 logic functions of 3 variables. The number of RBNs generated with the same set of updating functions at each iteration was 2000 and the number of random permutations generated to measure the mean complexity of the updating functions was 5000, though only which resulted not repeated were considered. The model fitted has an intersection with the vertical axis at -0.2709 and a slope of 1.253 with an R^2 coefficient of 0.2885. The Pearson correlation coefficient resulted to be $r_P = 0.537$, while the Spearman's rank correlation coefficient resulted to be $r_S = 0.443$. (b) Each set of updating functions was composed by 7 logic functions of 4 variables. The number of RBNs generated with the same set of updating functions at each iteration was 1000 and the number of random permutations generated to measure the mean complexity of the updating functions was 5000, though only which resulted not repeated were considered. The model fitted has an intersection with the vertical axis at -0.357 and a slope of 1.311 with an R^2 coefficient of 0.2121. The Pearson correlation coefficient resulted to be $r_P = 0.460$, while the Spearman's rank correlation coefficient resulted to be $r_S = 0.367$.

A.11). It is very similar to the code shown in A.10 since it also recycles pieces of codes we have been using throughout this chapter.

We finish this series of experiments by commenting again that, even though, the experiments performed here showed that the method proposed to represent and measure the complexity of Boolean Networks seems to be useful since the results make sense. It is still necessary to repeat the experiments with larger topologies and with larger ensembles. This was not possible to do in this study since the computational power needed was out of our reach. The main problem is the method to approximate the Kolmogorov complexity. This method is good measuring complexities, nevertheless, it is highly expensive and the computational time to measure the complexity of large sequences is ridiculous. For instance, in the experiment shown in Fig. 5.16 it was impossible to extend the experiment to Boolean Networks whose topology has more than 16 nodes. Thus, to get around with this problem, in the next chapter a low cost and faster implementation to measure Kolmogorov complexity is proposed. This implementation is less accurate but can reduce significantly the amount of time needed to compute the K-complexity.

5.8 Conclusions

In this chapter, we performed a series of experiments about the complexity of Boolean Networks by using the Random Boolean Network model. We started by trying to establish the best method to measure the complexity of random sequences of bits. The conclusion from this experiment is that the best way to compute the complexity of a sequence of bits, as expected, is to try to approximate Kolmogorov complexity. For short sequences, the technique based on lossless compression showed poor results, while the technique based on Shannon entropy gave similar results to K-complexity. On the other hand, for long sequences, the entropy and the lossless compression methods showed similar results, but both were not capable of imitating the behavior of K-complexity. Therefore, we confirmed that the best and most reliable method to measure complexity was the approximation to Kolmogorov complexity given by the BDM implementation. This was helpful since in the following experiments we could focus on this method and just care about the representation of the object since we already knew that other methods are not as good as this.

Then, we tried to measure separately the complexity of the constituents of a Boolean Network, i.e., the topology and the updating functions. Firstly, we needed to establish the type of representation of these objects through by which we would try to measure its complexity. Therefore, we performed some experiments with the help of some distributions of random graphs to check out the difference between the results obtained with a 1-D representation and a 2-D representation. The results with both representations were similar when using random graphs from the Barabási-Albert graph distribution but showed a small discrepancy with the Watts-Strogatz graph distribution, so it was not possible to define which representation was better, though both seem to work. This question was answered in the following experiments.

We started by considering graphs and not digraphs because with graphs we had

a way to control its complexity by using the mentioned graph distributions. Nevertheless, when we moved on to random digraphs, we needed a way to control its complexity. Our first tried was to control its complexity by increasing the number of incoming directed edges at each node. The results of this experiment were identical to the results obtained with random sequences of bits and by comparing these results with those, we concluded that the best representation to measure the complexity of a digraph was our 1-D proposal. Afterward, we tried to measure the complexity of the random topologies which are used to created Random Boolean networks, so we needed to measure the complexity of random digraphs generated with a constant number of nodes and incoming directed edges at each node. Nonetheless, this caused that we could not control the complexity of the random digraphs generated, so we used a visual approach to verify that our measurements made sense. The measurements confirmed that the 1-D representation was better than the 2-D representation since the behavior of the curve showed a richer variety of complexities. However, it was not possible to confirm visually the difference in complexity among the most complex digraph and the less complex digraph. This result was important because, from it, we inferred that even though our method to measure K-complexity was robust, it was not robust enough when using it to measure the complexity of digraphs and it was needed to consider the isomorphic representations of the digraph. This hypothesis was confirmed in the next experiments. Indeed, we showed that the complexities of the isomorphic representations of a digraph follow a normal distribution, which constitutes a result which was not originally expected. Once we established this, we repeated the experiment of random digraphs with a constant number of nodes and incoming directed edges at each node, but this time we did consider the isomorphisms, so this time it was possible to confirm visually the difference among the most complex digraph and the less complex digraph.

Thus, in this series of experiments we learned that the 1-D representation we proposed is better than the 2-D representation (which we also proposed), to measure the complexity of digraphs. Moreover, unexpectedly, we learned that it is important to consider the isomorphic representations of the digraph when measuring its complexity, especially if we are going to compare its complexity with the complexity of other digraphs. Thereby, with these experiments, we have established the basis to measure the complexity of digraphs and consequently the complexity of the topologies of Boolean Networks. In summary, we have established the best representation for digraphs, the best method the compute the complexity, and the importance to consider isomorphic representations. This knowledge could be used in future experiments to study the complexity of specific types digraphs (and consequently the complexity of specific types of topologies). Or on the other hand, it could be used to characterize digraphs with specific complexities or even to characterize Boolean networks whose topologies have specific complexities.

In the second part of this chapter, we proposed a way to represent a set of Boolean functions as a matrix. Then, we used this representation to measure the complexity of random sets of Boolean functions. In our first experiment, we tested our method by measuring the complexity of random sets of Boolean functions which complexity was controlled during the generating process. The results showed that our approach works. Afterward, we performed an experiment by generating random

sets of Boolean functions with a constant number of inputs. Thus, in this experiment we had no way to control the complexity of the sets of Boolean functions generated, so we had to use a visual approach to confirm that our measurements made sense. The results showed that our method measures correctly the complexity of a set of Boolean functions. Thereby, the representation we proposed could be used in future experiments to study Boolean Networks whose set of Boolean functions has a specific complexity.

In the third and final part of this chapter, we measured the complexity of Random Boolean Networks, i.e., this time we did not measure the complexity of its individual components but the complexity of the Boolean network as an ensemble. We started by proposing a way of representing a Boolean Network as a sequence of bits. This proposal was used to test the hypothesis of the correlation between the complexity of Random Boolean Networks and the complexity of its topology. The results showed that this correlation exists as expected and seems to be linear. Therefore, the representation we proposed for Boolean Networks is useful to measure its complexity. Then, we studied the width of the distribution of complexities for Random Boolean Networks. The results showed that this width increases with the number of nodes. Nevertheless, they also showed that as the number of nodes of the Boolean Network increases, it becomes necessary to consider larger ensembles in order to get results which can truly describe the average behavior of RBNs. Therefore, the experiments of this section must be repeated by considering larger ensembles of Random Boolean Networks in order to be able to generalize our results. Finally, we performed an experiment to test the hypothesis of the correlation between the complexity of RBNs and the complexity of its set of updating functions. Here, the results also showed that our hypothesis was correct so there exists a correlation between the complexities which also seems to be a linear correlation. Nonetheless, as stated before, it is necessary to repeat this experiment by considering ensembles of RBNs with more samples in order to be able to generalize this result.

Therefore, our representation for Boolean Networks works and by using it we were able to check some hypothesis about the complexity of RBNs and the complexity of its topology and its set of updating functions. Unfortunately, these results could not be generalized since it is necessary to consider larger ensembles. This was not possible due to the high computational cost of computing the K-complexity of large sequences of bits. This problem motivated the creation of a faster way to approximate algorithmic complexity. This method will be presented in the next chapter.

This page intentionally left blank.

Chapter 6

A Low-cost and Faster Implementation of Kolmogorov Complexity for Binary Sequences

In the last chapter, some experiments were performed with the help of the implementation of Kolmogorov complexity discussed in Section 4.4. It uses the so-called Block Decomposition Method or BDM for short. This implementation showed that it can measure the complexity of sequences of bits better than any other previous method. Nonetheless, it has a high computational cost, especially for large sequences. This and the enormous amount of possible applications motivates to look for new methods to measure the K-complexity. It is desirable to have a method which can approximate algorithmic complexity for long sequences in less time.

In this chapter, we propose a hybrid method which uses the results obtained with the BDM for short sequences. This hybrid method is based on the combination of a method of block decomposition and non-linear regressions performed with Neural Networks. We call this method The Block Decomposition Method with Neural Networks (BDMNN). The results indicate that this method is faster than the classical BDM, though the cost is an increase in the error of the complexity computed. Therefore, for long sequences, the behavior of the BDMNN approaches to the behavior shown by the entropy method to estimate complexity. Despite this, we strongly believe that the error can be reduced with the help of more sophisticated design of the Neural Networks, the use of larger training sets, and more learning time. Thus, this first attempt of hybrid method is just the beginning in the search of a faster implementation to measure Kolmogorov complexity and its results are promissory.

We will begin by giving the theoretical framework behind the functioning of neural networks as functions to perform non-linear regressions. Then, the description of the methodology used to create the new method to measure complexity will be presented. Finally, some results obtained with this implementation will be discussed and compared with those obtained with the classical BDM.

6.1 Theoretical Framework

6.1.1 Machine Learning

Nowadays the use of machine learning techniques has taken increasing importance. Machine learning helps to perform all kind of tasks. It is based on the idea of teaching and learning from examples, instead of direct programming as usual. Some usual definitions are the following:

Definition 6.1. Machine Learning is the science (and art) of programming computers so they can learn from data [84].

Definition 6.2. (Machine Learning is the) field of study that gives computers the ability to learn without being explicitly programmed [85].

When it is said that a computer learns, it means the following:

Definition 6.3. A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E [86].

We can understand this former definition with the simplest example of a machine learning program, a spam filter. This program can determine by itself whether an email is a spam or not. To do so, it uses as examples the emails which we tag as spam. These examples constitute the so-called *training set* or *training data*. Thus, in this example, the task T is the action of tagging the spam emails, the experience E is the training data, and the performance can be, for instance, the percentage of emails correctly tagged as spam by the machine.

Other common examples of machine learning programs are the following: a program which uses the photo of a person to predict its age, a program which recognizes whether a Tweet is aggressive or not, a program which can classify objects in a photo, a chatbot, a voice assistant, a program which can learn to play a game, a program which finds irregular data, a program which paints works of art, etc. As can be seen, the type of data which can be managed by the machine learning techniques is immense. It can be used to work with images, time series, text, video, music, measurements, medical data, graphs, etc. Every object which can be represented as a vector, or more generally as a tensor, can be manipulated with the techniques of machine learning to perform a specific task. The task can be a classification, a prediction, a regression, a decision making, a clustering, a detection, a sorting, the estimation of a probability density, etc.

There are many machine learning techniques and diverse ways to classify them. The most common way to classify machine learning techniques is by the type of human supervision during the training process. In this way, we distinguish between supervised, unsupervised, semi-supervised and reinforcement learning. In supervised learning we supply the program with training data and with the desired solutions, i.e., we target the training data with the solution corresponding to each example [86]. Conversely, in unsupervised learning, the training data does not have a labeled solution, so the program must learn without target solutions [86]. On the other hand, semi-supervised learning combines supervised and unsupervised learning since the

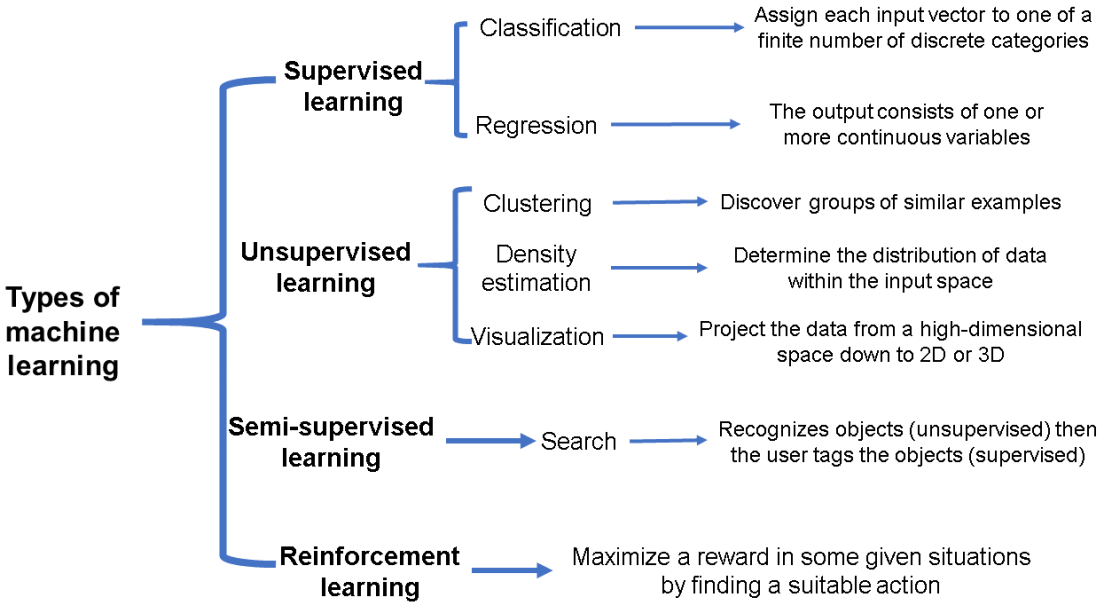


Figure 6.1: The types of machine learning techniques classified according to the type of human supervision during the training process and some tasks which can be performed with them [87].

training data is partially labeled with the solutions [86]. Finally, in reinforcement learning, an agent observes the environment and creates a strategy called policy to perform an action and receive a reward in return, its objective is to maximize this reward by means of the policy chosen [86].

In Fig. 6.1 can be seen the types of machine learning techniques and some tasks which can be performed with them. As was mentioned, every category includes distinct types of machine learning techniques which can be used for many different tasks. In this work, we are only interested in supervised learning. As was mentioned before, in supervised learning, the data we provide comprises examples with their respective labeled solutions. If the solutions (desired outputs) are a finite number of discrete categories, then the task is called classification. On the other hand, if they consist of one or more continuous variables the task is called regression [87]. This task tries to predict a target numeric value given a set of features known as predictors [86]. For instance, we could try to predict the height of a person (the target numeric value), given its age, sex, and nationality (the predictors).

There many machine learning techniques to perform a regression. Among the most important techniques, we can mention k-Nearest Neighbors, Linear Regression, Logistic Regression, Support Vector Machines (SVMs), Decision Trees and Random Forests, and Neural Networks [86]. The most appropriate technique will depend on many factors such as the type of data considered, the training speed, the accuracy desired, the memory needed to save the model, etc. Generally, the election of one model or other will depend on our experience and our knowledge and assumptions about the data. Nevertheless, the basic ideas behind all of them are the same. We provide the computer with a training set $\vec{X} = \{\vec{x_1}, \vec{x_2}, ..., \vec{x_N}\}$ and label the desired output of each example in this set with a target vector

$\vec{t} = \{t_1, t_2, ..., t_N\}$. Therefore, we have a list of training examples where each element is a pair of the form $(\vec{x}, t)$ which can be written for the N training examples as $((\vec{x_1}, t_1), (\vec{x_2}, t_2), ..., (\vec{x_N}, t_N)) = ((\vec{x}, t)_1, (\vec{x}, t)_2, ..., (\vec{x}, t)_N)$.

In the case of a regression, the output t_i represents one or more continuous variables and the input $\vec{x_i}$ can be a tensor of any dimensions [1]. Then, we perform the machine learning algorithm which returns a function y. This function can map new inputs $\vec{x_{new}}$ to a target value $t_{prediction}$ (the prediction), i.e., $y(\vec{x_{new}}) = t_{prediction}$. Evidently, the new input and output values $\vec{x_{new}}$ and $t_{prediction}$ will have the same dimensions used during the training phase. It is common to perform a preprocessing of the input data before the training phase to make the task simpler to learn [87]. This preprocessing is known as feature extraction, and when used before training the new data must be preprocessed as well.

Later in this chapter, we will choose the technique of Neural Networks to perform a regression. For this reason, in the next sections, we will briefly review this model and its training process.

6.1.2 Neural Networks

6.1.2.1 The Perceptron

The artificial Neural Networks were born inspired by biological Neural Networks. These networks are made up of millions of units called neurons which are interconnected in a complex web. In artificial Neural Networks (ANNs), the precursors of the units (the neurons) used in modern multilayer networks are the *perceptrons*, so we will begin the study of this unit model before considering networks with many interconnected units. Suppose we have n inputs $\vec{x} = \{x_1, ..., x_n\}$, then the output computed by the perceptron is the following [86]:

$$f(x_1, ..., x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases} \tag{6.1}$$

Where each w_i is a real-valued constant called *weight* and w_0 is a bias parameter. The weight determines the contribution of each input x_i to the output. As can be seen, the output of the perceptron depends on the linear combination $w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$, therefore it is usual to take $w_0 = 1$ and write the linear combination as $\sum_{i=0}^{n} w_i x_i$, then in vector notation the linear combination is $\vec{w} \cdot \vec{x}$, where $\vec{w} = \{w_1, ..., w_n\}$. Thus Eq. 6.1 is rewritten as [86]:

$$f(\vec{x}) = h(\vec{w} \cdot \vec{x}) \tag{6.2}$$

This is the general equation for a unit (a neuron). The function h is a nonlinear activation function, and in the case of the perceptron unit, this function is the *sign* function which is given by:

[1] Evidently, all the inputs $\vec{x_i}$ need to have the same dimensions, though there exist machine learning techniques were this is not necessary. On the other hand, the outputs t_i need always to have the same dimensions to be consistent.

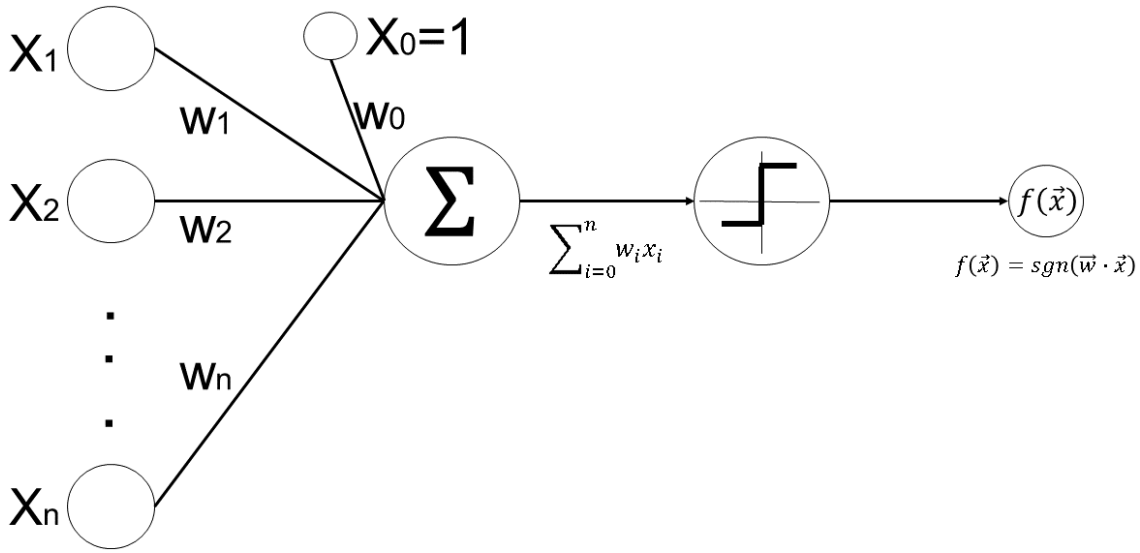


Figure 6.2: Graphical representation of the perceptron.

$$sgn(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise} \end{cases} \tag{6.3}$$

In Fig. 6.2 a graphical representation of the perceptron is shown. The perceptron can have two possible outputs, 1 and -1. This makes the perceptron a function which can represent Boolean functions such as the basic functions AND, OR, NAND, NOR. Besides, the perceptron is said to be a linear classifier, since it can classify a set of examples into two classes, corresponding to the values 1 and -1. Although, it only can classify sets of linearly separable examples, which means that it cannot represent functions like the XOR function [86]. This is represented in Fig. 6.3.

The behavior of the perceptron is adjusted by means of the weights $\vec{w} = \{w_1, ..., w_n\}$. These parameters allow the perceptron to learn. There are different algorithms to adjust these parameters, in the following section, we will review the method of gradient descent because it is the base of other learning algorithms which even can be used to train networks with many interconnected units. Unfortunately, the gradient descent algorithm requires the nonlinear activation function h in Eq. 6.2 to be differentiable. Then, it can only be applied to a to an unthresholded perceptron where h is the identity or a unit with a differentiable activation function.

6.1.2.2 The Gradient Descent Method

In this section, we will present the Gradient Descent algorithm to learn the weights $\vec{w} = \{w_1, ..., w_n\}$ of a unit. This method can be applied to any unit of the form given in Eq. 6.2 with the condition that the nonlinear activation function h must be differentiable. In this section, for convenience, we will consider as example and unthresholded perceptron, i.e., $f(\vec{x}) = \vec{w} \cdot \vec{x}$. Thus, in this section when we mention perceptron, we are referring to a perceptron whose nonlinear activation function h is the identity. In this way, if the problem to solve is linearly separable, then this

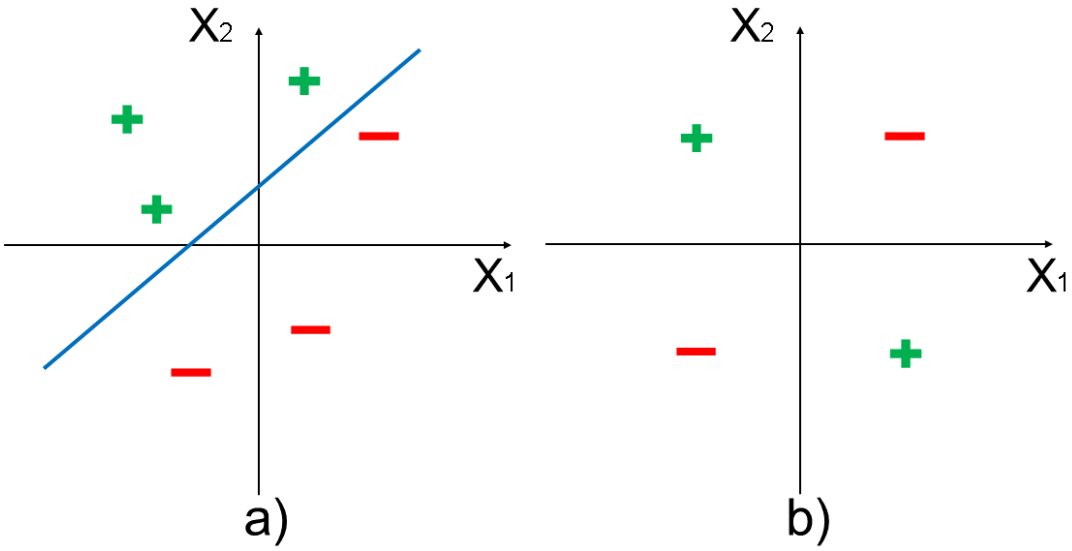


Figure 6.3: Graphical representation of the decision surface of a perceptron with two inputs: x_1 and x_2. If a set of examples is linearly separable, then it is possible to use a straight line to separate them in the decision surface. a) A set of linearly separable examples. b) The XOR function represents a set of non-linearly separable examples since its value is 1 if and only if $x_1 \neq x_2$ [86].

algorithm converges to the exact solution, however, if the problem is not linearly separable, then it converges to the best-fit for the given target set [86].

Firstly, as was mentioned in Definition 6.3, we need a performance measure of the learning process, i.e., we need to define an error function. There are different ways to define an error function, for a perceptron, we can define it as follows [86]:

$$E(\vec{w}) = \frac{1}{2}\sum_{i \in D}(t_i - f_i)^2 \tag{6.4}$$

Where D is the set of training examples, t_i is the target output for the training example i, and f_i is the output of the perceptron for the training example i. This error function is a function of the vector $\vec{w}$ because the output f of the perceptron is a function of this vector.

The error function of Eq. 6.4 measures the discrepancy of the output produced by the perceptron while using a hypothesis weight vector $\vec{w}$, against the desired output given by the set of training examples. Therefore, if the training examples are linearly separable, we expect this function to be 0 when a weight vector $\vec{w}$ makes the perceptron to exactly reproduce the targets of these training examples. On the other hand, in case that the training examples were not linearly separable, the best-fit approximation must occur when this function reaches its minimum value. This can be seen in Fig. 6.4. Therefore, the problem of training the perceptron is the problem of finding the minimum of the error function given in Eq. 6.4. In our case, this function always is parabolic with a global minimum, though in general, for networks with multiple units (and not necessarily the same kind of error function) there can exist multiple local minimums.

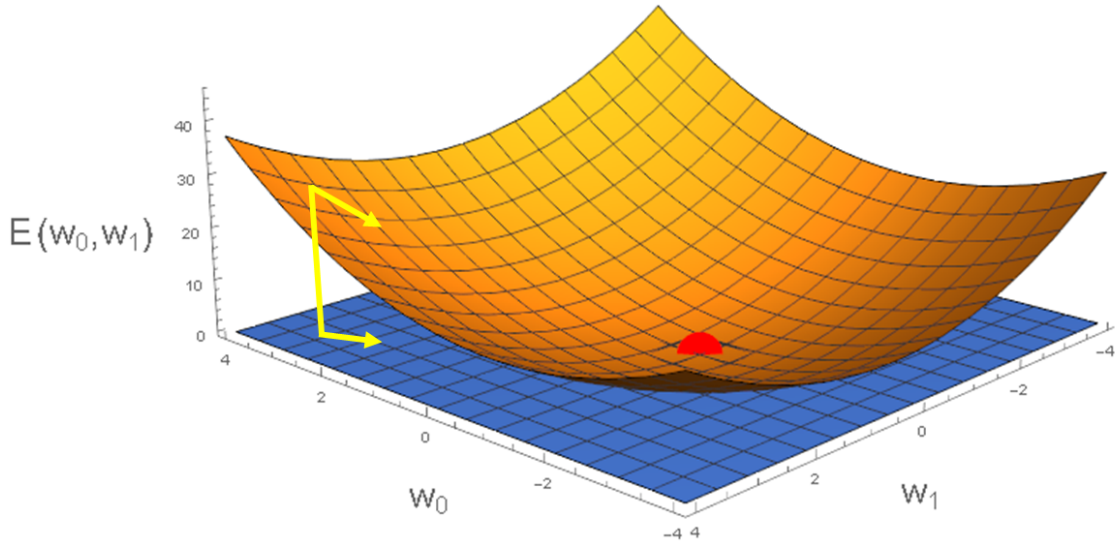


Figure 6.4: Error surface for a perceptron with weight vector $\vec{w} = \{w_0, w_1\}$. The axes w_0 and w_1 create a plane which represents the possible values of the weights w_0 and w_1. The axis E indicates the value of the error function. The shape of the parabola depends on the set of training examples. The arrow on the surface indicates the direction of the negative of the gradient at one particular point and its projection onto the plane w_0w_1 indicates the direction in this plane which produces the steepest descent along the error surface [86]. The red point indicates the global minimum.

The gradient descent algorithm searches the weight vector which minimizes the error function. The weight vector $\vec{w}$ is usually randomly initialized and at each step it is moved in the direction that produces the steepest descent along the error surface in steps of a determined size until the global minimum is reached.

As can be remembered, the result of the gradient is another vector which negative gives the direction of steepest decrease. Therefore, to find the direction of steepest descent along the error surface in a point, we need to compute the gradient of the error function with respect to the vector $\vec{w}$:

$$\vec{\nabla} E(\vec{w}) = \left\langle \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right\rangle \tag{6.5}$$

The negative of this gradient evaluated in a point (w_0w_1) gives the direction of steepest descent along the error surface from that point, as is shown in Fig. 6.4.

Thereby, at each step the vector $\vec{w}$ is updated following the rule [86]:

$$\vec{w} \leftarrow \vec{w} + \triangle \vec{w} \tag{6.6}$$

where

$$\triangle \vec{w} = -\eta \vec{\nabla} E(\vec{w}) \tag{6.7}$$

The constant η is a positive constant called the learning rate, and it determines

```
GRADIENT_DESCENT (training_examples, η) :
   ** where, training_examples = ((x⃗, t)_1, (x⃗, t)_2, ..., (x⃗, t)_N)
    **  x⃗_i = (x_1, x_2, ..., x_n)_i
    ** w⃗ = (w_1, w_2, ..., w_n)

1. w⃗ ←Initialize with small random values
2. Until the termination condition is met, Do :
     Δw_i = 0
     For each (x⃗, t)_i in training_examples, Do :
          f_i ←w⃗ · x⃗_i
          For each weight w_k, Do :
               Δw_k ←Δw_k + η (t_i − f_i) x_k
     For each weight w_i, Do :
          w_i ←w_i + Δw_i
3. Return w
```

Figure 6.5: Gradient descent algorithm for training a linear unit [86].

the step size to take in the direction of the steepest descent at each iteration. This rule can be written for each component as [86]:

$$w_i \leftarrow w_i + \triangle w_i \tag{6.8}$$

where

$$\triangle w_i = -\eta \frac{\partial E}{\partial w_i} \tag{6.9}$$

Thus, each component w_i is modified by an amount proportional to $\frac{\partial E}{\partial w_i}$. It is easy to construct an algorithm to perform the gradient descent rule by means of Eq. 6.9, we just need to compute the partial derivatives of E and evaluate them at the corresponding point. From Eq. 6.4 it is easy to show that the partial derivative of the error function with respect to w_i is given by:

$$\frac{\partial E}{\partial w_i} = \sum_{k \in D} (t_i - f_i)(-x_{ik}) \tag{6.10}$$

Where x_{ik} is the single input component x_i for the training example d. Thus, we can rewrite Eq. 6.9 as follows:

$$\triangle w_i = \eta \sum_{k \in D} (t_i - f_i) x_{ik} \tag{6.11}$$

The learning rate η can increase the speed of the convergence, however, if its value is too large, the algorithm can get us away from the minimum instead of getting us closer. Thus, to guarantee the convergence it is necessary to use a sufficiently small value for η, especially being close to the minimum [86].

The gradient descent algorithm to train a perceptron is shown in Fig. 6.5. This method can be applied if the error function is differentiable with respect to the weights. Nonetheless, it may converge to the minimum quite slow, moreover, if the error surface has multiple local minima, it is not guaranteed that it will find the global minima.

As was mentioned before, there are many other algorithms to train a unit. These methods can overcome the difficulties encountered with the gradient descent method and fortunately, many of them are simple variations to the method presented in this section, such as the algorithm of stochastic gradient descent which will be discussed in the next section given its importance in the majority of the most important optimization methods to train Neural Networks.

6.1.2.3 The Stochastic Gradient Descent Method

The idea behind this method is to avoid the sum of Eq. 6.23. Instead, we consider an error function defined for each individual training example as follows [86]:

$$E_k(\vec{w}) = \frac{1}{2}(t_k - f_k)^2 \tag{6.12}$$

Thus, the weights are updated using each training example $(\vec{x}, t)_k$ following:

$$\triangle w_i = \eta(t_k - f_k)x_i \tag{6.13}$$

Where for the kth training example, t_k is the target value, f_k is the output of the unit, and x_i is the ith input.

Therefore, in this method the search of the minimum is guided by $\nabla E_k(\vec{w})$ instead of $\nabla E(\vec{w})$, i.e., instead of Eq. 6.7 we use $\triangle \vec{w} = -\eta \vec{\nabla} E_k(\vec{w})$.

6.1.2.4 Feed-Forward Neural Networks

In this section, we will see how we can combine many units (neurons) to create a multilayer network also known as Neural Network or NN for short, but first, we need to choose the kind of units which will compose the network. The former optimization methods required the activation function of the unit to be differentiable. In fact, many optimization methods demand this requirement. Hence, this is the main reason for not considering the perceptron as the basic unit for constructing a multilayer network., even though, this model played a key role in the history of machine learning [87]. Thus, it is common to build multilayer networks by considering units whose activation function is differentiable.

A basic Neural Network model is built with units of the form given by Eq. 6.2. Nonetheless, for convenience, we will establish a new nomenclature. To do so, we will describe how to build a multilayer network from the beginning. Let $\{x_1, ..., x_D\}$ be the input variables, then firstly, we form M linear combination of these input variables as follows [87]:

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \tag{6.14}$$

Where $j = 1, ..., M$. The number M of linear combinations indicates that this first layer has M units. The superscript (1) indicates that these units are in the first layer of the network. As before, the parameters $w_{ji}^{(1)}$ are the weights (in the first layer according to the superscript). It must be remarked, that we have adopted the convention of taking the bias parameter as $w_{j0}^{(1)} = 1$ with $x_0 = 1$ as we did for

the perceptron. Each unit of the layer a_j is activated through a nonlinear activation function $h^{(1)}$:

$$f_j^{(1)} = h^{(1)}(a_j) \tag{6.15}$$

Each $f_j^{(1)}$ represents an activated unit which in the context of Neural Networks are known as hidden units [87]. As was said, the activation functions must be differentiable, but we will talk about the choosing process later. Equations 6.14 and 6.15 define the hidden units in the first layer, to add a new layer to the network, we follow the same procedure, i.e., we construct linear combinations of the input variables for the second layer. For the second layer these input variables are the hidden units in the first layer:

$$a_k = \sum_{j=0}^{M} w_{kj}^{(2)} f_j^{(1)} \tag{6.16}$$

Where $k = 1, ..., K$. The number K of linear combinations indicates that this second layer has K units, or in other words, the second layer has K outputs. Again, we have taken the bias parameter as $w_{k0}^{(2)} = 1$ with $f_0^{(1)} = 1$. These units are also activated with a nonlinear activation function $h^{(2)}$:

$$f_k^{(2)} = h^{(2)}(a_k) \tag{6.17}$$

Thus, up to now, we have constructed a Neural Network with two layers (we will not consider the input variables of the first layer as units). The function of this Neural Network is:

$$f_k^{(2)}(\vec{x}, \vec{w}) = h^{(2)}\left(\sum_{j=0}^{M} w_{kj}^{(2)} h^{(1)}\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \right) \right) \tag{6.18}$$

Thereby, a Neural Network model is simply a nonlinear function of the input variables $\vec{x}$ and the weights $\vec{w}$ with the set of outputs $\{f_k^{(2)}\}$. The topology of the Neural Network defined by Eq. 6.18 can be represented by the graph shown in Fig. 6.6. This type of topology is said to be fully connected since every node transfers information to every node in the next adjacent layer [88]. Evidently, this is not the only kind of possible topology, but it is one of the most common and is easily generalized, besides it is the topology we will use later in this chapter. This type of topology for a Neural Network is known as feed-forward Neural Network since the process of evaluating is a forward propagation of information through the network [87]. This model is also known as multilayer perceptron because it resembles the perceptron model discussed in Section 6.1.2.1, nevertheless, the perceptron uses an activation function which is not differentiable, whereas the Neural Network uses differentiable activation functions. Thus, the function of a Neural Network is differentiable with respect to the network parameters which is important for network training.

The Neural Network defined by Eq. 6.18 has two layers, however, the process described in this section can be repeated to add as many layers as we wish. The kth output of a Neural Network with m layers can be represented by the following equation:

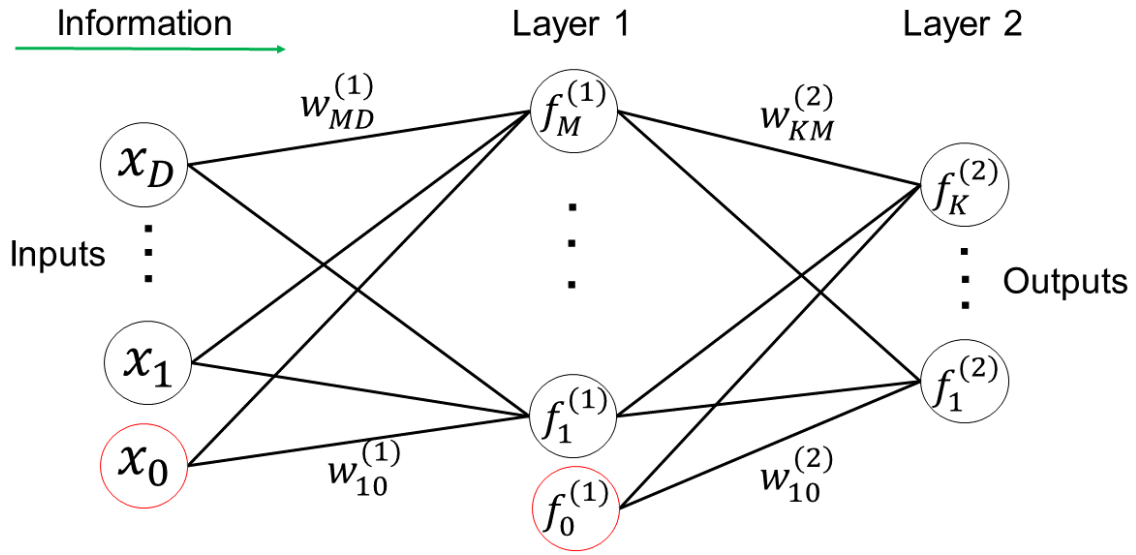


Figure 6.6: The topology of a feed-forward Neural Network with two layers. The information flows through the network in the direction indicated by the green arrow. The nodes of the graph indicate the input variables, the hidden units (first layer) and the output units (second layer). The edges of the graph indicate the weight parameters. The red nodes indicate the hidden variables whose edges are the bias parameters. The layers which are not input or output layers are known as hidden layers. In this case, we have a single hidden layer, the first layer.

$$f_k^{(m)}(\vec{x},\vec{w}) = h^{(m)}\left(\sum_{j=0}^{M} w_{kj}^{(m)} h^{(m-1)}\left(\sum_{i=0}^{Q} w_{ji}^{(m-1)} h^{(m-2)}\left(\cdots\left(\cdots h^{(1)}\left(\sum_{q=0}^{D} w_{qp}^{(1)} x_p\right)\right)\right)\right)\right) \tag{6.19}$$

Hence, it is possible to imagine a Neural Network with m layers simply as a composition of functions:

$$f_k^{(m)}(\vec{x},\vec{w}) = h^{(m)}(f_k^{(m-2)}(f_k^{(m-1)}(\cdots(f_k^{(1)}(\vec{x},\vec{w}))))) \tag{6.20}$$

Where $f_i(\vec{x},\vec{w}) = h^{(i)}(\sum_j w_{ij}^{(i-1)} x_j)$, and the sum runs over all units which send information to unit i, including the bias parameter. This way to see the system is important since it means we can compute the derivatives of the Neural Network by means of the chain rule.

6.1.2.5 Feed-forward Neural Networks as Approximators

The model described by Eq. 6.19 seems to be quite complex, so it is expected it will be a great model to perform approximations. In fact, Neural Networks are said to be universal approximators [87]. We have some theorems regarding several types of functions:

Theorem 6.1 (Boolean Functions)**.** Every Boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs ([86], page 105).

Theorem 6.2 (Continuous Functions)**.** Every bounded continuous function can be uniformly approximated on a compact input domain with arbitrarily small error (under a finite norm) by a network with two layers of units provided the network has a sufficiently large number of hidden units. The units must have a sigmoid activation function at the hidden layer and (unthresholded) linear units at the output layer ([86], page 105; and [87] page 230).

Theorem 6.3 (Arbitrary Functions)**.** Any function can be approximated to arbitrary accuracy by a network with three layers of units. The output layer must have linear units, and the two hidden units must have units with a sigmoid activation function. The number of units required at each layer is not known in general ([86], page 105).

6.1.2.6 Network Training

In this section, we will present the Backpropagation algorithm to train the weight parameters of a multilayer network with a fixed set of units and interconnections. This algorithm is based on the Gradient Descent Method studied in Section 6.1.2.2. Therefore first, we need to define an appropriate error function. To do so, we will generalize the error function presented in Eq. 6.4 to consider the error over all the network output units [86]:

$$E(\vec{w}) = \frac{1}{2} \sum_{i \in D} \sum_{k \in outputs} (t_{ki} - f_{ki})^2 \tag{6.21}$$

Here, *outputs* is the set of output units of the network, t_{ki} is the target output for the training example i in the kth output unit, and f_{ki} is the output of the network in the kth output unit for the training example i.

Thus, as was argued in Section 6.1.2.2, the problem of learning the weight parameters is the problem of minimizing the error function of Eq. 6.21. This error function can be visualized by an error surface like the one shown in Fig. 6.4. Nonetheless, now this space is defined by the error function defined in Eq. 6.21 and the weight parameters of all the units in the multilayer network. The idea is the same, we need to find the values of the weight parameters which minimize the error function, i.e., we need to find the minimum of the error surface. Therefore, we can use the gradient descent method to find the minimum. However, this time the error surface can have multiple local minima, so the convergence towards the global minimum is not guaranteed.

The idea is the same, we randomly initialize the weight parameters and compute the gradient of the error function. The negative of the vector given by the gradient will indicate the direction of steepest descent along the error surface in that given point. Then, we update the weight parameters according to $w_{ji} \leftarrow w_{ji} + \triangle w_{ji}$. Following the same procedure of Section 6.1.2.2, we can compute the gradient of the error function and obtain the rule to update the value of the weights. We will not show it here, but the gradient descent weight-rule is given by [86]:

$$\triangle w_{ji} = -\eta \nabla E(\vec{w}) = \eta \delta_j x_{ji} \tag{6.22}$$

Where η is the learning rate, x_{ji} is the input value to which the weight is applied, and δ_j is given by:

$$\delta_j = \eta f_j(1 - f_j) \sum_{k \in Downstream(j)} \delta_k w_{kj} \tag{6.23}$$

Here, *Downstream* denotes the set of units whose direct inputs include the output of unit j. This last equation shows that the value of δ for a hidden unit is obtained by propagating backward the δ values from units higher up in the network. This is the reason for the name backpropagation.

There are other algorithms to train the parameters of a multilayer network which follow immediately as variations of the rule given in Eq. 6.22, nevertheless, we will not consider its details since it lies out of our reach. In practice, we only need to know the convergence properties of each method to choose one to solve a specific task.

6.1.2.7 Designing the Network

The design of a Neural Network to solve a specific task is an art which only can be learned with the experience. However, there are certain rules which can be used as a starting point.

The number of input units will be given by the dimensionality of the network, e.g., if our input data are 1-D vectors whose length is 8, then the number of input units will be 8 plus a unit for the bias parameter. In the same way, the number of output units will be given by the expected output of our model. For instance, if we perform a regression, then the output of the network will be a scalar which means the number of output units will be 1. On the other hand, if for example, our model gives as output the presence or the absence of a cat and/or a dog in an image, then it would be adequate to use two output units. An output unit which gives the presence or absence of the cat and another which gives the presence or absence of the dog.

The number of hidden units must be adjusted to give the best predictive performance; however, we need to have in mind that more layers mean more units which also means more parameters to be trained. Therefore, it is necessary to get a balance between a good predictive performance and the training time needed to reach it. If we need to perform a regression, we can see from the theorems presented in Section 6.1.2.5 that usually two hidden layers will suffice for most of the problems. The number of units in these hidden units also must be adjusted to give the best balance between performance and training time. Even though it is possible to use the number of inputs units as the number of units in the hidden layers, this is not recommended [89]. It is a better practice to have fewer units in the hidden layers than in the input layer to force the network to learn compressed representations of the original input. This does not mean that using the same number of units in the input layer and in the hidden layers will give bad results, however, it is possible that in this way we are introducing redundancy in our model which do not contribute to enhancing its performance and just makes larger the computational time needed to train the network.

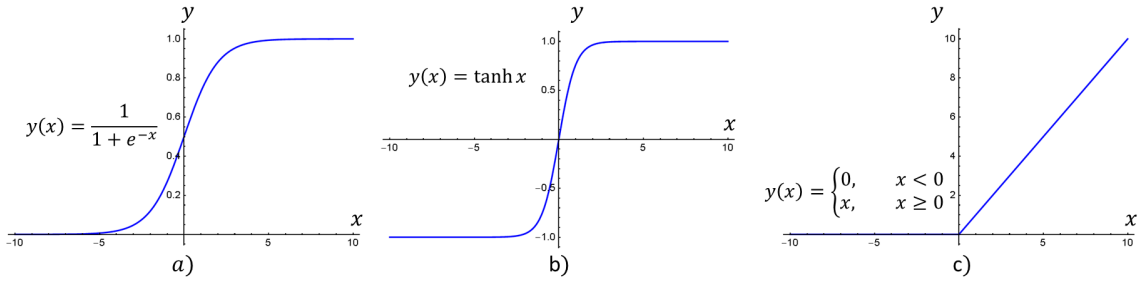


Figure 6.7: The most commonly used activation functions for the units of multilayer networks. a) The Logistic Sigmoid function, b) The Hyperbolic Tangent function, and c) The Rectified Linear Unit function (ReLU).

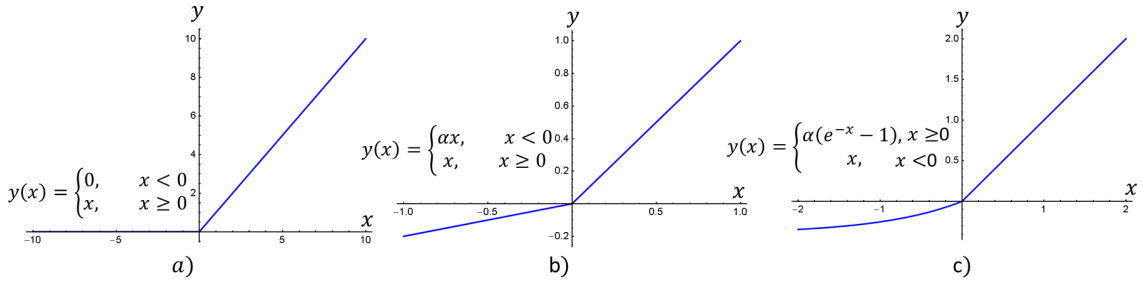


Figure 6.8: Some of the most common variations for the Rectified Linear Unit function (ReLU) [90] [91]. a) The ReLU function. b) The Parametric ReLU function (PReLU) (in this function the slope α is given to the network as a parameter to be learned. When this parameter is fixed with the value $\alpha = 0.01$, this function is known as Leaky ReLU). c) The Exponential Linear Unit.

Moreover, it is not required to connect every unit (neuron) with all the units in the next layer, i.e., the fully connected topology shown in Fig. 6.6 is not a demand when designing the network. However, again it is an art the election of which neurons will be connected to another neuron [89].

As can be remembered, the activation functions introduce non-linearities to the multilayer networks which allow them to represent almost any arbitrary function. The election of the activation function for the units depends on whether the unit belongs to a hidden layer or an output layer, in the specific task of the model, and in the expected performance. In the hidden layers, the most commonly used activation functions are: Sigmoid (or Logistic), Tanh (Hyperbolic Tangent), and Rectified Linear Unit (ReLU) (and its variants) [88]. These activation functions can be seen in Fig 6.7. Every activation function has its advantages and disadvantages, nevertheless, the ReLU activation function or its variants are usually chosen for most of the tasks [89], some variants of this function have been created to solve some drawbacks in the training process (see Fig. 6.8).

In the case of the output layers, the election of the activation functions is driven by the type of output expected. For regression problems, we use a linear activation function, i.e., $y(x) = x$. For binary classification, we use a single neuron with a sigmoid function which delivers a probability distribution for a single class (the

probability of the other class is $(1-p)$). For multiclass classification we use the so-called *SoftMax output layer* (also known as SoftMax layer) which uses the outputs of all the units in the output layer to compute a probability distribution for each output unit as follows [89]:

$$y_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{6.24}$$

Where the sum of all the outputs fulfils $\sum_j e^{x_j} = 1$. A perfect prediction will give an output from the SoftMax layer with the probability value of 1 while the other outputs will have a probability value of 0, indicating that this output is the class predicted by the model. An imperfect prediction will give several values for each output. The class will be indicated by the output with the highest probability value.

We finish this section by commenting that there are other types of Neural Networks which can be used to solve determined types of tasks more efficiently, like the Convolutional Neural Networks or the Recurrent Neural Networks, though we will not consider them here.

6.1.2.8 Choosing the Error Function

As we saw before, the error function measures the performance of the Neural Network while using some set of weight parameters with respect to a given a set of training examples, i.e., it measures the misfit between the output of the network and the corresponding target in the training data set. This function, which is minimized by the optimizer to get the best performance, is also known as the loss function. We will use the terms error function and loss function as synonyms.

In Eq. 6.21 we defined an error function, nevertheless, there are many other possible error functions which may be more appropriate to use depending on the type of task to solve and the nature of the data. Here we will review some of the most frequently used error functions. In our notation, we will use f to indicate the output of the network, t_i to indicate the target of the ith sample in the training set, and N to indicate the number of samples in the training set. Finally, t_{ij} will indicate the output of the jth output unit obtained with the ith sample of the training set.

Loss Functions for Regression

These loss functions can be used when the task to solve is a regression problem. The most popular is the *Mean Squared Error Loss (MSE)* function, which is given by [88]:

$$E(\vec{w}) = \frac{1}{N}\sum_{i=1}^{N}(f_i - t_i)^2 \tag{6.25}$$

This function resembles the error function used to perform the ordinary least squares in linear regression, but here we divide the sum by the number of training samples. This function is mainly used to perform regression when the output is a real value, i.e., a scalar. If the output is a vector of size M, or in other words, if the

outputs are M continuous variables, then the former loss function is generalized as [88]:

$$E(\vec{w}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{M}\sum_{j=1}^{M}(f_{ij} - t_{ij})^2 \tag{6.26}$$

Note that Eq. 6.25 is the especial case where $M = 1$. For convenience this function is sometimes modified as [88]:

$$E(\vec{w}) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M}(f_{ij} - t_{ij})^2 \tag{6.27}$$

An alternative to the MSE function is the *Mean Absolute Error (MAE) Loss* function [88]:

$$E(\vec{w}) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M}|f_{ij} - t_{ij}| \tag{6.28}$$

When the target value has a spread of values the model can be rudely affected by the outliers. To avoid so, we use the *Mean Squared Logarithmic Error (MSLE) Loss* function [88] [92]:

$$E(\vec{w}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(log(f_{ij}) - log(t_{ij}))^2 \tag{6.29}$$

Finally, another possible error function is the *Mean Absolute Percentage Error (MAPE) Loss* function [88]:

$$E(\vec{w}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\frac{100 \times |f_{ij} - t_{ij}|}{t_{ij}} \tag{6.30}$$

The choice of the loss function for regression problems will depend mainly on the type of data. For most of the cases, the MSE and MAE functions will suffice. However, when the data vary largely in range, the MSLE and MAPE functions should be considered.

Loss Functions for Classification

These functions are used when we need to classify data into categories. For binary classification, the most commonly used loss function is the *Hinge Loss* function which is given by [88]:

$$E(\vec{w}) = \frac{1}{N}\sum_{i=1}^{N} max(0, 1 - t_{ij} \times f_{ij}) \tag{6.31}$$

This function categorizes the data as -1 or 1. If we are interested in the probability of belonging to a given class, more than a raw classification, then we are interested in logistic regression. The most common function for such cases is the *Cross-Entropy* function, also known as *Logarithmic loss* function [87] [93]:

$$E(\vec{w}) = -\sum_{i=1}^{N}[t_i log(f_i) + (1 - t_i) log(1 - f_i)] \tag{6.32}$$

This function can be extended to consider the classification of M classes (multi-class classification) [87]:

$$E(\vec{w}) = -\sum_{i=1}^{N}\sum_{j=1}^{M}[t_{ij} log(f_{ij})] \tag{6.33}$$

6.1.2.9 Choosing the Optimization Method

In Section 6.1.2.2, we had the first contact with the Gradient Descent Method, and in Section 6.22, we saw how to apply it to multilayer networks. Nevertheless, this method has the disadvantage of being quite slow. The training process could take less time if we choose to use a faster optimization method. The most common and faster optimizers are the following: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, Adam optimization, and Nadam optimization. As was mentioned in Section 6.22, we cannot describe its details here since it lies out of our reach. Nevertheless, we can always safely choose to use Adam and Nadam methods. These methods are more powerful and usually converge faster than any other method since they combine ideas from other optimization algorithms ([84], page 293). The Nadam optimization method is an improvement to Adam optimization method, so if possible, we must give preference to Nadam over the Adam method [94].

6.1.2.10 Regularization Techniques

When training a Neural Network, we minimize the loss function by evaluating the given training set. We could think that the closer the value of the loss function to 0, the better the model we get. However, we need to have in mind, that the purpose of our model is not to exactly reproduce the training data but to generalize its behavior to brand new data. We already know the behavior of the training data, so it is not interesting to build a model which exactly reproduces it. Instead, we would like to have a model which can generalize the behavior of the training data to unknown data. Therefore, it is not a clever idea to let the network learn to reproduce exactly the training data. This problem of producing a model that performs well on the training data but generalizes poorly to new data is known as overfitting ([95], page 142).

The problem of overfitting is shown in Fig. 6.9. Suppose that we generate random data from the function $Sin(x)$ by introducing some random noise as shown in Fig. 6.9a. Then, we use this random data as training data for a Neural Network. If we let the Neural Network overfit the training set, then we will get a model like the one shown in Fig. 6.9b. This model reproduces almost exactly the training data; however, it is far from the original model from which the data was generated. Thus, we can say that the overfitted model has learned the random noise in the training data, so this model is not capable of generalizing new data to the original model $Sin(x)$. Nevertheless, if we use a technique to avoid the overfitting, we can get a

model which is more similar to the original model, even though, the training set is not perfectly adjusted as shown in Fig. 6.9c.

Fortunately, there exist some techniques to avoid the overfitting of the training data. The most common technique is to reserve a portion of the training data as test data while using the rest to train the network. This test data will be used to evaluate the model in an independent way to the optimization process which is driven by the training data. In this way, we can stop the training process when the error in the test data stops decreasing and starts to increase which means the model is beginning to overfit to the training set. The test data allow us to see how well the model generalizes on new unknown data ([89], page 29).

Additionally, it is usual to divide the training process into epochs. An epoch is a single iteration over the entire training set. Moreover, the optimization process is not performed with all the training data at the same time. Instead, we perform the optimization process by using batches with a determined size obtained from the training data [96]. The reason behind this can be better understood through an example: suppose we have a training set with a million of images as training samples, then the time needed to optimize the model by evaluating the error obtained with each one of these images would be immeasurable. Thus, it is better to perform the optimization process by using mini-batches (subsets) of the training data at each iteration of the optimization process. This approach allows to use the hardware resources more efficiently ([88], page 201). The performance of the model is measured at the end of each one of these mini optimization processes using the test data. However, we would also like to measure the performance of the model at the end of each epoch. Therefore, it is also usual to reserve another portion of the original training data as validation data. This validation set is used to measure how well the model is generalizing to new data at the end of each epoch ([89], page 29). If the error in the training data continues decreasing but the error in the validation data stays the same or starts to increase, then it means we should stop the training process because the model has begun to overfit the training data.

In summary, we split up the original data set into three sets: the training data, the test data, and the validation data. The training data is supplied to the optimizer in batches. The validation data is used to measure the performance of the model on new data at the end of each mini-optimization process. Finally, the validation data is used to measure the performance of the model at the end of each epoch. This method is quite successful, so it is widely used ([86], page 111).

Another common technique to avoid overfitting is *Dropout*. The idea is simple, at every training step, every neuron in the network (excluding the output neurons) has a probability p of being temporarily turned off ([84], page 304). This means that at that training step, the neuron is not used in the process of optimization, but it may be turned on during the next step. The parameter p which controls the activation of the neurons is called the dropout rate. This kind of parameters which control the distribution of the model parameters, and thus, the optimization and model selection during the training process are known as hyperparameters ([87] page 30, and [88] page 150). This algorithm is quite successful in increasing the training speed

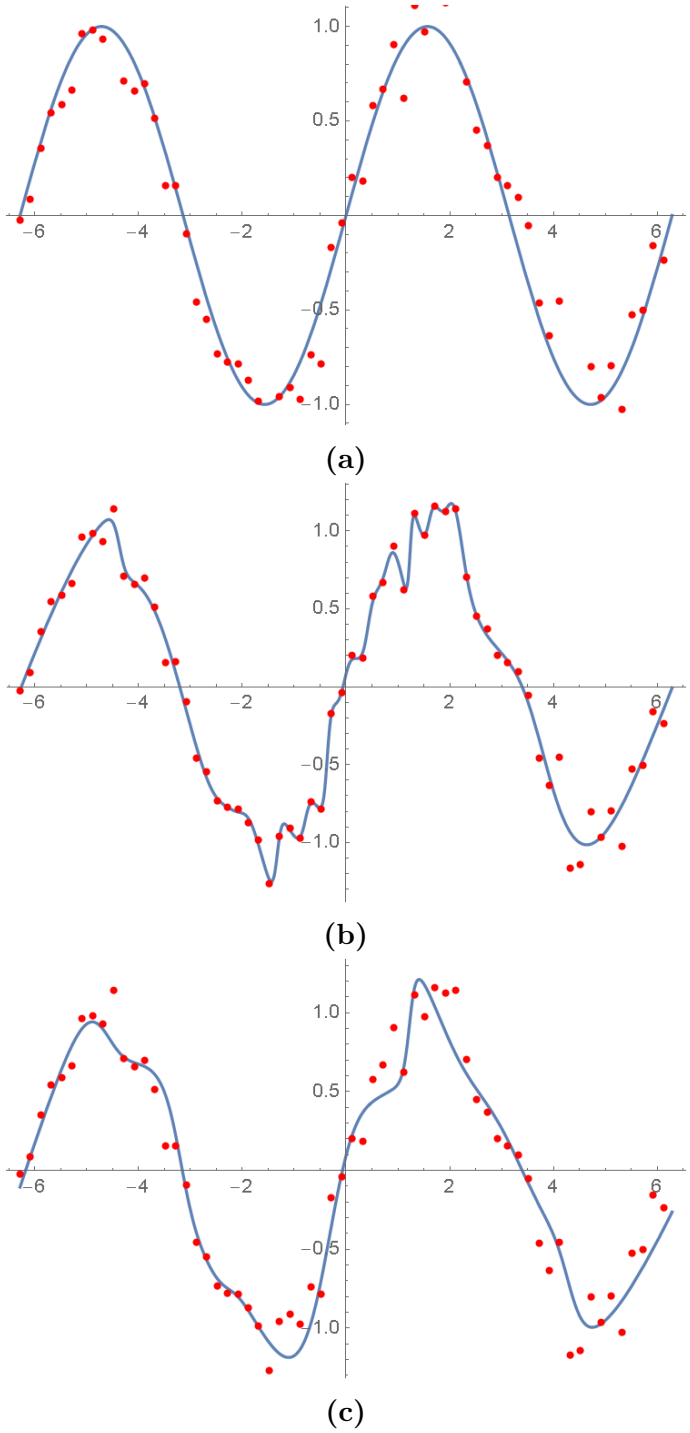


Figure 6.9: (a) Random data (red dots) obtained from introducing random noise to the function $Sin(x)$ (blue curve). (b) A Neural Network was trained by using the previous random data as training data. No method was used to avoid overfitting, so the model obtained has learned the noise of the training data and shows overfitting. (c) The same previous Neural Network was trained by using the same previous random data as a training set, but this time a validation set was used as a method to avoid overfitting. The model obtained does not show overfitting to the training data and reproduces better the original function $Sin(x)$ from which the data was obtained.

and boosting the accuracy of the resulting model ([88], page 200). This method prevents the neuron to adapt to its neighbors by obligating it to rely less on them and by making it independent and useful by its own. What is really happening here, is that by ignoring some neurons at each training step, we are evaluating different topologies. These architectures share some neurons with their respective weights. Therefore, the resulting model is an average of the behavior of these topologies ([89] page 34, and [84] page 305).

The last technique which will be considered here is the *early stopping*. This technique consists in stopping the training process when the error with respect to the validation data set stops decreasing ([84], page 303). This allows obtaining a model which has a good generalization performance ([84], page 259).

There exist more regularization techniques to avoid overfitting, however, we will not consider them here. Better performance can be obtained by combining the use of different regularization methods at the same time ([84], page 303).

6.1.3 Machine Learning Frameworks

The techniques and methods we have reviewed in this section are already implemented in many machine learning frameworks. Thus, we do not have to spend time programming the algorithms and we can simply focus on the design and deployment of our models. Among the best and most common libraries to implement machine learning methods we can name: *Scikit-learn*, *TensorFlow*, and *Keras*. These libraries can be easily installed and used in any Python distribution and altogether with other famous packages for scientific computing in Python like *NumPy*, *SciPy*, *matplotlib*, *pandas*, etc., they make very easy and intuitive the programming and training of models like multilayer Neural Networks (see [95] chapter 25, [84] chapter 9, [97] chapter 1, [98] page 343, and [89] chapter 3).

6.2 Methodology

The hypothesis behind our implementation is that the computational time needed to compute the approximation of the K-complexity with the Block Decomposition method can be significantly reduced by creating a function which imitates the behavior of the results delivered by the BDM. In other words, if we could build a function to perform a regression from some examples computed with the BDM. Therefore, once we have built this function, its evaluation could be immediate, so we could predict the complexity of new data just by performing simple arithmetic operations without the needed of performing an intricate algorithm every time we need to evaluate the complexity of a new sequence. The computational time would be absorbed in the process of creation of this function. Fortunately, this process only would need to be performed one time. Thereby, we could save a lot of time by recycling this computation. Nonetheless, the function to perform a regression of the K-complexity must be a very complicated non-linear function. Hence, it should be impossible to build it with simple regression techniques such as linear regression. Nevertheless, machine learning provides powerful techniques to perform non-linear regressions which could be used instead. For instance, a Neural Network can be

used as a powerful non-linear regression model. Therefore, given the complexity of the problem, it was decided to use this model to perform the regression.

Therefore, our implementation to measure the K-complexity was built in two stages. First, three Neural Networks are individually trained with examples of complexities computed with the Block Decomposition Method. Each Neural Network is specialized in predicting the complexity of sequences of bits over a given range of lengths. The second stage consists in the creation of a function which accepts sequences of any length (sequences which length can be even larger than the initial sequences used to train the Neural Networks) and returns an approximation of Kolmogorov complexity. The previous is achieved by means of a method similar to the original Block Decomposition Method. Once we have this function, we will only have to evaluate it every time we wish to compute the complexity of a sequence, which makes it faster than the original BDM. The stages of the methodology proposed are shown in Fig. 6.10. In the following sections, the details of these stages will be presented.

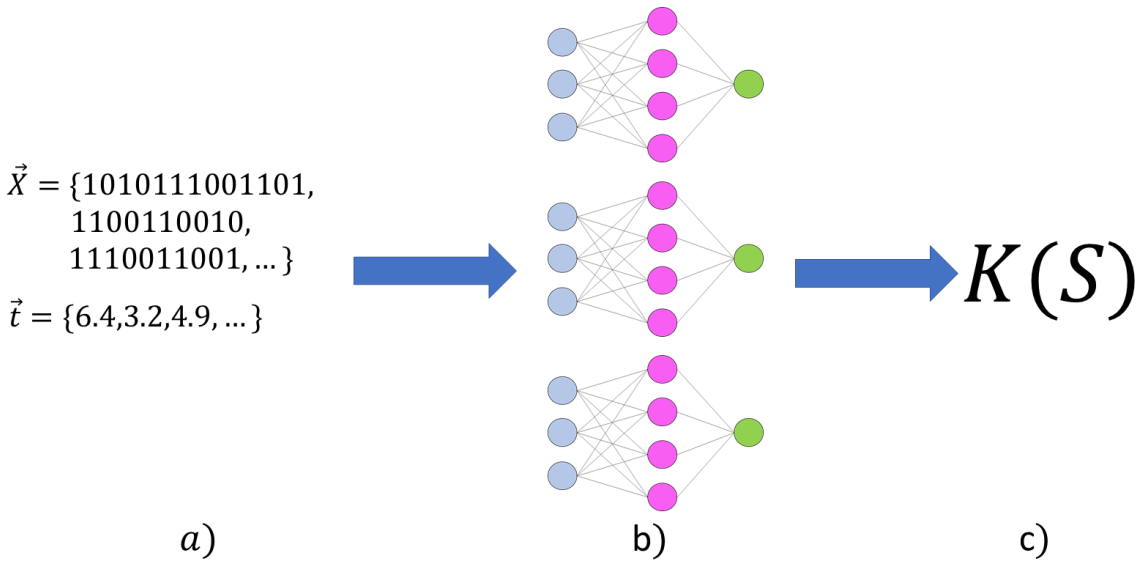


Figure 6.10: Stages of the methodology used to create a faster implementation to approximate K-complexity. a) Some random sequences of bits are randomly generated, and its complexity is measured by means of the BDM. The random sequences form the training set $\vec{X}$ and its complexities the target vector $\vec{t}$. b) The training data is used to train three Neural Networks which perform a regression to approximate the algorithmic complexity of sequences of bits. Each Neural Network is specialized in approximate the K-complexity of sequences of bits between a given range of lengths. c) The three Neural Network models can be used to predict the complexity of new sequences of bits. If the new sequence has a length which lies out of the range used to train any of the three Neural Networks, the sequence is decomposed into blocks to approximate its complexity. Thus, this method resembles the original Block Decomposition Method, so we have decided to name it the Block Decomposition Method with Neural Networks (BDMNN). The details of these stages are presented in the text.

6.2.1 Software and Hardware Features

The most widely used language to program machine learning techniques, especially Neural Networks, is Python. In this language, we can find many frameworks which make fast and easily the task of implementing Neural Networks in just a few lines of code, so we only have to worry about the design and training of the models. For this reason, all the codes of this section will be written in Python Language, specifically in Python 3. The framework chosen to build and train the Neural Networks was the library *Keras*. This library is user-friendly and allows to rapidly build and train Neural Network models in an intuitive way.

To train the Neural Networks, we needed sequences of bits whose complexity was known a priori (as will be described in the next section). Therefore, we had to rely again on the implementation of Kolmogorov complexity discussed in Section 4.4. Nevertheless, this time we are interested in the Python version of this implementation which can be downloaded from the same page[2]. This library contains the function *BDM* which can compute the 1-D and the 2-D versions of the Block Decomposition method, though we only will use the 1-dimensional version in this chapter. It accepts as argument an array (specifically a NumPy array) of $0's$ and $1's$ of length greater or equal to 12 and returns an approximation to the algorithmic complexity of the sequence.

All the algorithms were implemented on the Platform *Colaboratory*[3] of Google (Google Colab) since it provides immediate access to most of the frameworks for machine learning and executes Python code in the cloud by using virtual machines. The virtual machines can be executed using GPUs instead of CPUs which makes possible to train Neural Networks faster than with a common local machine, even though they are recycled every certain amount of time so the Neural Networks can be trained in a continuous way only for a few hours.

6.2.2 The Training and Target Sets

One of the main complications when training Neural Networks is the amount and quality of the data used as training examples. Fortunately, in our case, we can generate and provide as many examples as we wish.

The procedure is very similar to the one used in Section 5.2 to generate random sequences of bits. First, we establish the maximum and minimum lengths that can have the sequences to generate. Then, we randomly (with a uniform probability) choose a length L in this closed interval $[L_{min}, L_{max}]$ of maximum and minimum lengths. This will be the length of the sequence to generate. Moreover, we randomly (with a uniform probability) choose a number between the interval $[0, 1]$. This former number will be the probability parameter p to control the complexity of the sequence generated. Afterward, we generate a sequence of length L, where each element has a probability p of being 1 and a probability $(1 - p)$ of being 0. Finally,

[2] *www.algorithmicdynamics.net/software.html*

[3] https://colab.research.google.com

the complexity of this sequence is computed by means of the BDM implementation. This algorithm is repeated a given number of times and the data generated is saved in two vectors (implemented as lists in Python). The sequences generated are stored in a vector $\vec{x}$ and its complexities in a vector $\vec{t}$. The vector $\vec{x}$ is the training vector and the vector $\vec{t}$ is the target vector.

The training data must be a representative sample of the possible input sequences; thus, it is important to provide as many examples as possible, but also is important ensure a uniform election of these examples among all the possible input sequences. Nevertheless, in our procedure to generate the random sequences, the cases where the sequences are only composed of $1's$ seldom occur since they appear almost only when the parameter p equals 1. To overcome this problem, we increase the frequency of apparition of these rare cases simply by rounding of the parameter p to 1 when it is greater than 0.95.

Finally, once we have all the sequences and its complexities, we perform a padding procedure to fill the sequences which length was less than the maximum length. We append the value -1 at the beginning of each sequence until the sequence a length equal to the maximum length. In this way, for instance, if we have the sequence $\{0, 1, 1, 0, 1\}$ and the maximum length is 8, then the padding procedure would produce the sequence $\{-1, -1, -1, 0, 1, 1, 0, 1\}$. Therefore, if the complexity of the original sequence before the padding was $K(S)$, we assign to the padded sequence this same complexity $K(S)$. The final vector $\vec{x}$ with the padded sequences and the vector $\vec{t}$ with the complexities assigned to the padded sequences are the vectors which will be used to train the Neural Network.

The code to generate the training and target sets as described above can be found in the Appendix Section A.2 (Fig. A.12). We generated the data to train three Neural Networks. The intervals for the length of the initial sequences (before the padding) were $[L_{min}, L_{max}] = [12, 20]$,$[21, 100]$ and $[101, 1001]$. In the three cases, the number of training examples generated was $30,000$.

6.2.3 Design of The Neural Networks

Neural Networks are widely used to solve classification problems, however, with the correct design, they also can be used to solve regression problems. They are not the only machine learning technique which can be used to perform a regression task, nevertheless, it is expected that the function which estimates Kolmogorov complexity will be highly complicated to approximate, thus, we believe that a Neural Network with the appropriate design should be able to give good results. The previous follows from the theorems presented in Section 6.1.2.5. Nonetheless, the question of whether this is the best approach still open, though we will no try to answer this question here. Finally, the design of a Neural Network is an art since there is no standard procedure to build its topology. Our design was based on common practices.

The design of the three Neural Networks used in this work was the same. The only difference was the number of neurons present in each layer. The number of

sequential dense layers used was 3. The first layer had a number of neurons equal to the length of the final sequences (the maximum length) and the activation function used was the Exponential Linear activation function (ELU). This activation function has the form $f(x) = x$ for $x \geq 0$ and $f(x) = \alpha(e^x - 1)$ for $x < 0$ (the value of the parameter α was chosen to be 1). The second layer had a number of neurons equal to $\frac{2}{3}$ times the length of the final sequences (the maximum length) and its activation function also was the ELU function. The parameter α also was chosen to be 1 for this layer. The final layer only had one neuron and a linear activation function, i.e., $f(x) = x$.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 100)               10100
_________________________________________________________________
elu_1 (ELU)                  (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 67)                6767
_________________________________________________________________
elu_2 (ELU)                  (None, 67)                0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 68
=================================================================
Total params: 16,935
Trainable params: 16,935
Non-trainable params: 0
_________________________________________________________________
```

Figure 6.11: Summary of the design for the Neural Network which predicts the K-complexity for sequences of length between 21 and 100. The number of parameters to train increases with the number of layers and units, i.e., with the number of neurons.

In Fig. 6.11, the summary of the design of one of the three Neural Networks built is shown. The code to build the Neural Network using the library Keras is shown in the Appendix Section A.2 (Fig. A.13).

6.2.4 The Learning Settings

The method used to perform the optimization of the parameters of the Neural Networks was the Nesterov adaptive moment estimation optimizer (Nadam). This method usually converges to the optimal solution faster than other methods like the stochastic gradient descent method. The loss function chosen was the Mean Squared Error Loss function. The batch size used was 20, and the entire data set was shuffled before making the batches with this size. We used a validation set to check at each epoch the accuracy of the model to data it has not seen. This validation set was randomly chosen among the training data and its size was 20% the size of the original training set. As shown in Fig. 6.11, the number of trainable parameters increases with the number of units (neurons), so the time of training on

each epoch also increases, besides, the number of epochs to reduce the validation error also increases. Thus, the number of epochs used to train each Neural Network was different. For the Neural Network which predicts the K-complexity of sequences which length is in between 12 and 20 the number of epochs was 2500, for the Neural Network which predicts the K-complexity of sequences which length is in between 21 and 100 the number of epochs was 3000 and finally, for the Neural Network which predicts the K-complexity of sequences which length is in between 100 and 1001 the number of epochs was 2500. To avoid overfitting of the model to the training data, it is usual to establish a dropout parameter, which says the percentage of neurons to be off on each epoch. Nonetheless, in this case, it was not necessary to perform the dropout procedure.

6.2.5 The Block Decomposition Method with Neural Networks

Once we have performed the training of the Neural Network models, they can be used to predict the complexity of new sequences. Nevertheless, each Neural Network was trained to predict the complexity of sequences whose size is in between a given range of maximum and minimum lengths. In order to get the prediction of a new sequence which length is in between this interval, it is necessary to perform the same padding procedure used before in the generation of the training examples, i.e., we append the value -1 at the beginning of the sequence until it has a length equal to the maximum length. It is not possible to provide the Neural Network with sequences whose size is different since it is built and trained to only accept sequences with this feature.

Therefore, by using the three Neural Networks which we trained, it is possible to predict the K-complexity of sequences whose length lies in between the range 12 and 1001. Nonetheless, we would like to have a unique function which can predict the Kolmogorov complexity of any sequence, no matter its size. For this reason, we implemented a modified version of the Block Decomposition Method by using the three Neural Network models trained.

The idea is to take advantage of the three Neural Networks trained with sequences of specific lengths to predict the complexity of sequences of any size. The algorithm is the following. If the sequence we need to measure its complexity has a size which lies in between the range of sizes accepted for one of the three Neural Network models, then we simply use this Neural Network to predict the complexity of the sequence. However, if the sequence we need to approximate its complexity has a size which lies out of the range of sizes accepted for the three Neural Network models, then we perform a Block Decomposition Method, similar to the one described in Section 4.4.1.2. For example, if the sequence has a size longer than the maximum length of 1001, then we split this sequence into block sequences of length equal or less than 700. Finally, the complexity of these block sequences is measured with the corresponding Neural Network model and the complexity of the initial sequence is obtained by summing the individual complexities of these block sequences following the Eq. 4.20. In our case, we have chosen to always split the original sequence into non-overlapping sequences of length equal or less than 700. If

the sequence is not a multiple of 700, then there will be a remainder. This remainder was chosen to be ignored if its length is less than 12 and we just simply assign to it a complexity value of 2.285794 which is the minimum complexity which can have any sequence (see [63]), otherwise we measure its complexity using the corresponding Neural Network. We call this method the Block Decomposition Method with Neural Networks (BDMNN) in analogy with the original BDM.

The code use to define the function described above is presented in the Appendix Section A.2 (Fig. A.14). In the following section, the results of some experiments implementing this function will be shown.

6.3 Results

The implementation described above was tested by performing some experiments concerning the complexity of random sequences of bits. These experiments and its results will be shown up next.

6.3.1 The Complexity of Random Sequences of Bits

This experiment is based on the experiment performed in Section 5.2, but obviously, this version is implemented in Python instead of Mathematica, so there are some slight changes. First, we select the length of the sequences of bits to be generated. After, we set the probability parameter $p = 0$ and begin a loop. At each iteration of the loop, we generate a sequence of bits of the length chosen initially[4]. As can be remembered, the parameter p controls the complexity of the sequence generated, since each element in the generated sequence has a probability p of being 1, and a probability $(1 - p)$ of being 0. This parameter is increased in an amount dp at the end of each iteration until it reaches the value of 1. The complexity of the generated sequence is measured by means of the original BDM implementation and our BDMNN implementation.

As was mentioned, this is the same experiment is basically the same that the experiment performed in Section 5.2. Therefore, BDM implementation must show the same behavior shown in Fig. 5.1. On the other hand, we expect BDMNN to show similar behavior to BDM, which would mean that our method is also a good approximation to Kolmogorov complexity.

We corroborated the previous, by performing this experiment with sequences of different lengths. The results are shown in Fig. 6.12. As can be seen, the results obtained with our implementation, the BDMNN, showed a behavior which is pretty similar to the behavior and the results obtained with the BDM for sequences of length less than 1000. This means that the regression performed by the Neural Networks was able to imitate the approximation to Kolmogorov complexity given by the original Block Decomposition Method. Nevertheless, for sequences whose size was greater than 1000 bits, the results delivered by the BDMNN began to differ

[4]This time we will not generate several sequences with the same p value to compute its complexity and obtain a mean value as was did in Section 5.2. We only generate a sequence for each p value.

from the results of the BDM. We can see from the experiments with sequences of lengths 1200, 5000 and $10,000$ that the curve obtained with the BDMNN gets wider with the increase in the length of the sequences considered in the experiment.

The previous means that the error of the BDMNN increased with the size of the sequence considered, but moreover, we can see from the experiment shown in Fig. 5.1b that the behavior of the BDMNN started to seem more like the behavior given by the techniques based on entropy or compression to measure complexity. Thereby, our method seemed to be capable of approximate the K-complexity with comparable results to the ones delivered by the BDM, moreover, it performed like the Shannon entropy for long sequences, for which it lost accuracy. This convergence to an entropic behavior also is present in the original implementation of the Block Decomposition Method (see [57]). Thus, it is natural that our implementation which is an imitation of the original BDM also should show this behavior. However, in the case of the BDMNN, it was amplified by the accumulation of error when predicting the complexity of the decomposed block sequences. This caused the entropic behavior to be manifested faster than it did with the BDM.

This does not mean that our proposal, the Block Decomposition Method with Neural Networks was inferior. It must be remembered that its initial purpose was to be a faster implementation to approximate the K-complexity. Therefore, if it proves to be faster than the BDM, it could become into a powerful alternative tool, especially because it could be used when it is possible and needed to sacrifice some accuracy of the approximation in exchange of speed in the computation.

6.3.2 The Error and Computational Time of the BDMNN

Now, we will try to show that our implementation works faster than the original Block Decomposition Method. To do so, we will perform an experiment to measure the computational time needed by both methods, the BDM, and the BDMNN, to measure the K-complexity of sequences of bits.

We will start by generating a random sequence of bits with a given length. This time it is not necessary to control its complexity, so the sequence will be randomly chosen among all the possible sequences of the given length with a uniform probability distribution. Then, we will measure its complexity with the BDM and the BDMNN. For both methods, we will perform a timing to determine the computational time needed to execute the function which computes the complexity with each method. The timing will be performed with the aid of the function *process_time* which belongs to the library *time* of Python. The resolution of this function, when implemented in Colab, was 1 ns. Besides the timing process, we will compare the complexity computed by the BDMNN with the complexity computed by the BDM to determine the absolute error obtained with our method. We will perform this procedure starting with a sequence of length 12, then we will perform it with a sequence of length 13 and so on.

The results of this experiment are shown in Fig. 6.13. As can be seen in Fig. 6.13a, this experiment showed as expected, that our implementation is faster than

the BDM, moreover, the computational time of the BDMNN stayed constant with the sequence length, while it increased linearly for the BDM. Furthermore, the absolute error stayed bounded below 10% for sequences whose size was less than 4000 with the only exception of the initial sequences for which the absolute error was around 17%. For sequences with sizes larger than 4000, the absolute error increased linearly. However, it increased slowly since for sequences of length $10,000$ its value was barely around 10%.

These results have shown that our initial hypothesis was correct and the BDMNN computes faster the K-complexity. Moreover, the loss of accuracy occurs slowly with the length of the sequence and only is significant for very long sequences. This justifies its use for calculations were the computational time is important and the BDM takes so so long to be implemented that it becomes a restriction for the experiment.

The code to execute the experiment of this section can be found in the Appendix Section A.2 (Fig. A.15).

6.3.3 The BDMNN Versus a Parallel Implementation of the BDM

Now, we want to go further and evaluate the speed of our implementation versus a parallel implementation of the BDM. This parallel implementation is included in the library of the Block Decomposition Method for Python. This function splits the sequence and distributes the pieces into the available kernels. In this way, every kernel computes a piece of the process to accelerate the BDM.

This parallel implementation to compute the BDM was created with the objective of decrease the computational time needed approximate the K-complexity. In fact, this implementation was not available in the Wolfram Language implementation, so it is expected to be possibly one of the fastest possible implementations since Python by itself is a fast language. Hence, if we can beat this implementation, our implementation automatically could be considered the fastest implementation to compute the Kolmogorov complexity. Moreover, even if our implementation cannot beat the parallel implementation of the BDM, it still is possible to consider also a possible parallel implementation for the BDMNN.

We will repeat the same experiment of the last section, but this time instead of the classic BDM we will use this parallel implementation. The timing performed in the following experiment does not consider the time needed to divide the sequence into smaller sequences nor the distribution of them to the available kernels. The number of parallel kernels used was 4.

The results of this experiment are shown in Fig. 6.14. It can be seen in Fig. 6.14b that the absolute error obtained with the BDMNN showed a similar behavior that the absolute error shown in Fig. 6.13b. Nevertheless, the absolute error obtained with the initial sequences (short sequences with sizes of around 12) was greater. This discrepancy not necessarily means that our implementation gave a big error for these short sequences. This result can be explained because the parallel

implementation of the BDM has a different behavior for short sequences and since the Neural Networks of our implementation were trained with examples obtained from the original BDM, it was not expected the BDMNN to agree with the parallel implementation of the BDM. The parallel implementation of the BDM seems to have boundary problems for short sequences unlike the original BDM, so the result shown in Fig. 6.13b for short sequences was not alarming.

On the other hand, the result in Fig. 6.14a showed that the computational time increased linearly for both methods. Nonetheless, the slope of this increase was very low for both methods. In fact, the BDMNN started with computational times larger than the computational times for short sequences with the parallel implementation of the BDM, however, the slope of the BDMNN was smaller than the slope of the parallel implementation for the BDM, thus, this would mean that for long sequences the BDMNN should overcome the parallel implementation. Nevertheless, for sequences whose size was around $16,000$ existed a jump in the computational time which made the computational time obtained with the BMDNN to increase. Even so, the parallel implementation of the BDM was just around $2ms$ faster than the BDMNN in the worst case. Thereby, a refined implementation for the BDMNN should easily overcome any implementation of the BDM.

Notwithstanding the previous results, we can argue that the former experiment was not fair since we did not take into consideration the time needed to divide the sequence into smaller sequences and to distribute them to the available kernels in the parallel implementation. For this reason, we repeated the experiment, but this time the timing for the parallel method included the time needed to split and distribute among the kernels the given sequence.

The result of this experiment is shown in Fig. 6.15. The results for the absolute error shown in Fig. 6.15b were the same as before. Nevertheless, the results in computational time were quite different. The results which are shown in Fig. 6.15a showed that when we took into account the time needed to split and distribute the sequence among the available kernels, the parallel implementation was easily surpassed by our method, the BDMNN.

The code to execute the two former experiments can be found in the Appendix Section A.2 (Fig. A.16).

6.4 Conclusions

The results obtained with our proposal, the Block Decomposition Method with Neural Networks (BDMNN), were outstanding, considering the simplicity of the idea behind it. It showed by far, to be faster than the original Block Decomposition Method. It even competes in speed with the parallel implementation of the BDM. The loss in the accuracy is not significant and remains below 20% of the absolute error for sequences with sizes below $20,000$ bits. In the worst cases, its behavior resembles an entropic like behavior.

We strongly believe that this method can be improved even further. We could reduce the error by enhancing the training of the Neural Network models by means

of more data examples, more training time, a more sophisticated model, or even another regression technique could be considered. Also, better programming of the block decomposition algorithm could help. For example, by considering an overlapping parameter or the use of more neural networks to perform more refined predictions. Finally, a parallel implementation with Neural Network models should render spectacular timing results.

The original Block Decomposition Method uses the complexity results for sequences whose length is equal or less than 12 to compute the complexity of larger sequences. Instead, our method uses the complexity results for sequences whose length is in between 12 and 1000, to extend them to larger sequences. Therefore, both methods are basically the same, the only difference is the size of the sequences whose complexity is known. In the case of the BDM, its complexity was computed from the Coding Theorem Method (CTM), and in the case of the BDMNN, its complexity is computed from the regression performed by Neural Networks.

In the end, our method can be used as a good approximation to Kolmogorov complexity, especially when the computational time is important, and we do not care to lose some accuracy in the results.

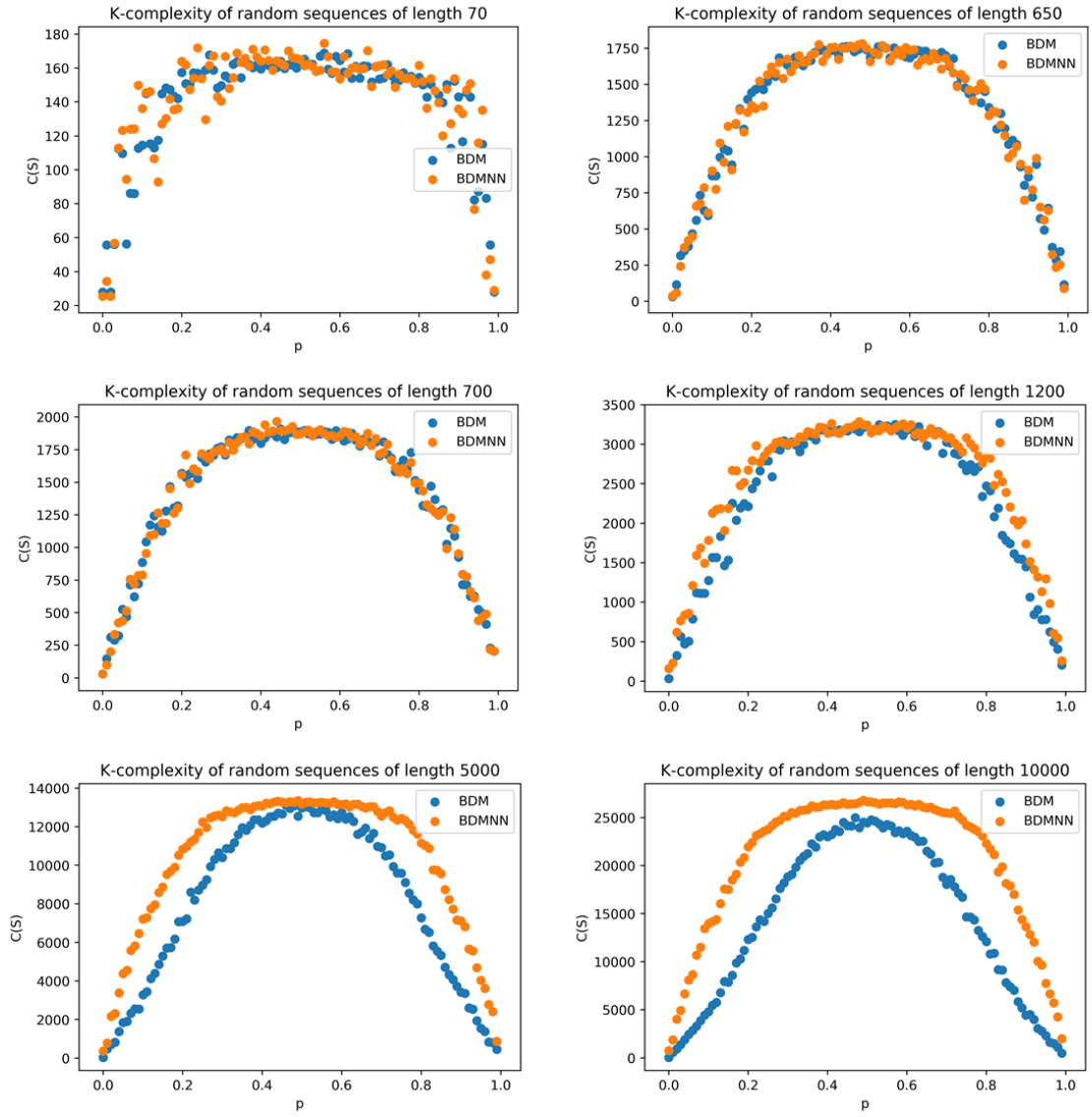
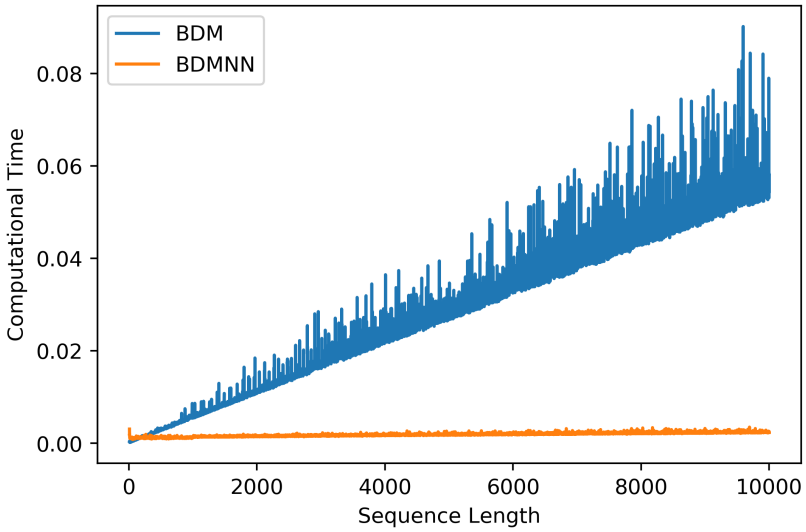
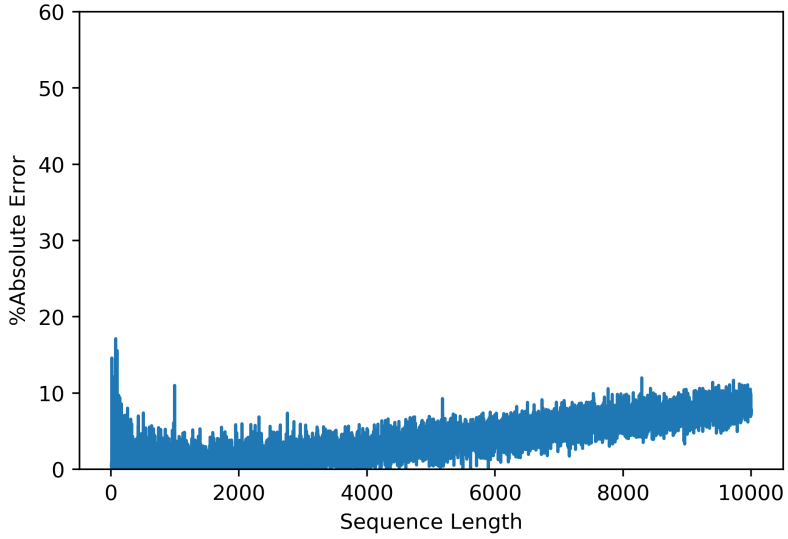


Figure 6.12: The complexities of random sequences of bits of different lengths versus the probability parameter. The complexities were measured with the original Block Decomposition Method and with our proposal, the Block Decomposition Method with Neural Networks. The number of random sequences generated in each experiment was 100.

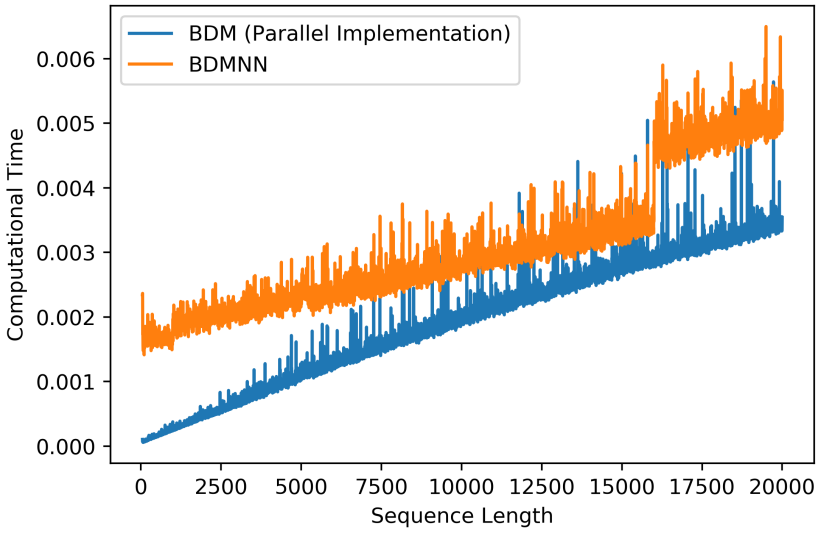


(a)

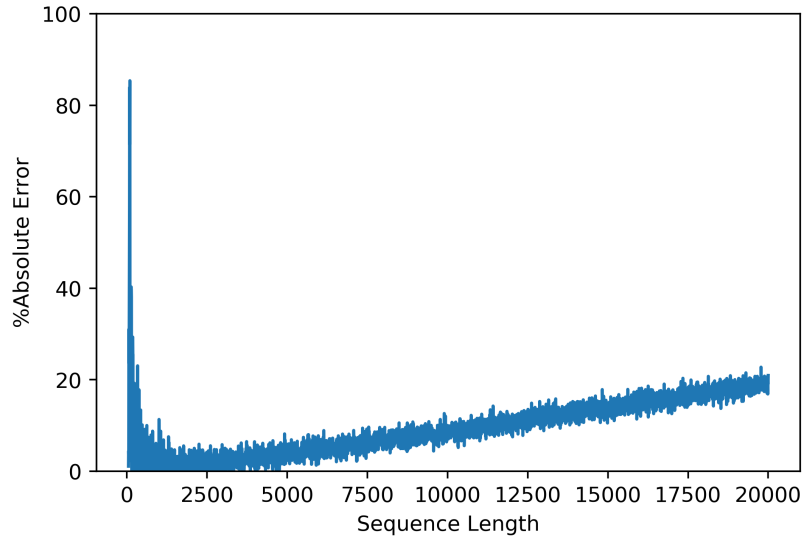


(b)

Figure 6.13: (a) The computational time needed to implement the Block Decomposition Method (BDM) and the Block Decomposition Method with Neural Networks (BDMNN) to measure the K-complexity, versus the sequence length. The computational time is measured in seconds. (b) The percentage of absolute error of the complexity computed with the BDMNN compared with the complexity computed with the BDM, versus the sequence length.

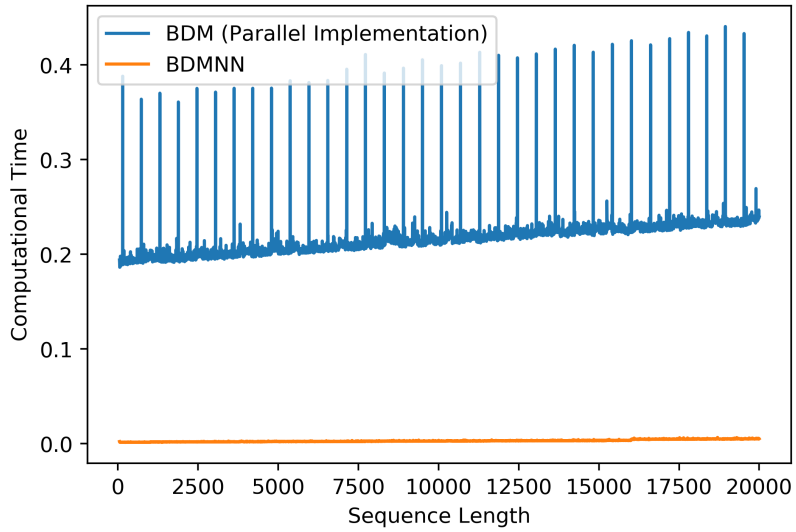


(a)

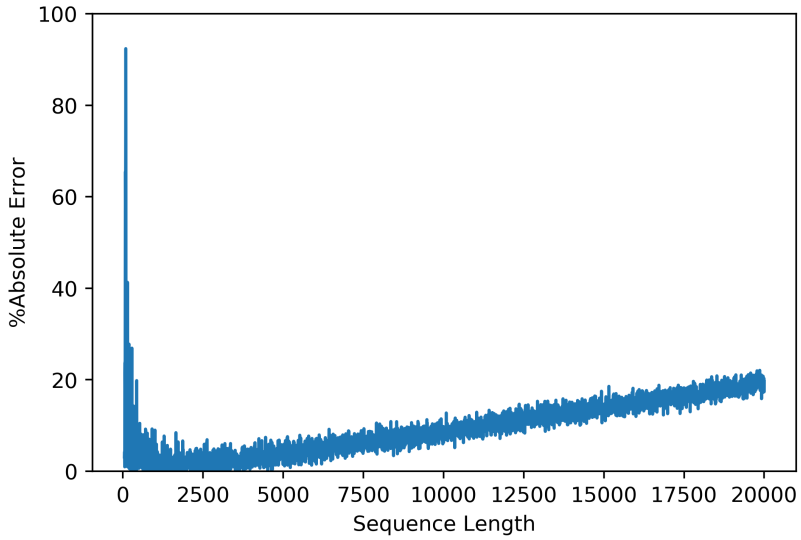


(b)

Figure 6.14: (a) The computational time needed to compute a parallel implementation for the Block Decomposition Method (BDM) and the Block Decomposition Method with Neural Networks (BDMNN) to measure the K-complexity, versus the sequence length. The computational time is measured in seconds and for the parallel implementation, it does not consider the time needed to split and distribute among the available kernels the given sequence. (b) The percentage of absolute error of the complexity computed with the BDMNN compared with the complexity computed with the parallel implementation of the BDM, versus the sequence length.



(a)



(b)

Figure 6.15: (a) The computational time needed to compute a parallel implementation for the Block Decomposition Method (BDM) and the Block Decomposition Method with Neural Networks (BDMNN) to measure the K-complexity, versus the sequence length. The computational time is measured in seconds and for the parallel implementation, it does consider the time needed to split and distribute among the available kernels the given sequence. (b) The percentage of absolute error of the complexity computed with the BDMNN compared with the complexity computed with the parallel implementation of the BDM, versus the sequence length.

Chapter 7

General Conclusions

Throughout this thesis, we have been probing distinct methods to measure the complexity of distinct objects. We have proved that in all the cases the best measurement of complexity is obtained by trying to approximate Kolmogorov complexity. Thus, this implementation is recommended and can be safely used in any application for which there is a necessity to measure complexity. Nonetheless, we have shown that for certain mathematical objects this implementation must be used with care. If the mathematical object we are trying to measure its complexity has distinct isomorphic representations, then the true complexity of the object is the minimum value obtained with one of these isomorphic representations. Besides, for graphs or any mathematical object which could be represented as a matrix, we have seen how it is better to transform this representation to a 1-dimensional representation to get better measurements of complexity. These results are important since they will make easier to perform future experiments about complexity.

Moreover, we proposed a novel application to Kolmogorov complexity by measuring the complexity of Boolean networks. To do so, we needed to propose a representation for Boolean Networks which were representative of its behavior and could be used to measure its complexity. This representation was evaluated by using it to prove some hypothesis of correlation among the complexity of a Boolean Network and the complexity of its constituents. The results agreed with which was expected which means our approach works correctly and the representation used correctly captures the essential features which characterize the complexity of a Boolean Network. Although, future experiments with larger ensembles of Boolean Networks will be needed to be able to generalize the results and confirm these hypotheses for Boolean Networks of any size. Even so, the results of our proposal to measure the complexity of Boolean Networks are promising and they could be used in the study of real-world systems which can be modeled by means of Random Boolean Networks. For instance, we could use the measurements of the complexities of the same genetic regulatory network for different species (modeled as Boolean Networks) to try to establish phylogenetic relationships. This is because we could expect the complexity of a genetic regulatory network to be lower for species who evolved earlier than others. Of course, our methods and observations obtained by measuring the complexity of graphs, digraphs, and binary sequences could be used to study real-world systems as well.

Our experiments also showed that the library we used to measure K-complexity works well, though it needs a lot of time to compute the complexity for long sequences. This problem makes it not suitable to study the complexity of objects whose representation is large. Therefore, given the enormous amount of possible applications where it is needed to compute Kolmogorov complexity. We decided to propose a novel method which could be used in situations where we need a faster implementation to measure complexity. Our implementation called BDMNN works faster and needs very low computational power. The only cost to pay is a little loss in accuracy, however, we believe that this implementation can be easily enhanced to reduce the error of approximating K-complexity by training more Neural Networks with larger data sets and larger training times. This novel approach works quite similar to the original BDM and in the worst cases behaves just like Shannon entropy, so it can be used immediately for any application. Future experiments could help to determine if a regression with any other machine learning technique works better. Certainly, it could be used in future experiments to generalize the results of our hypothesis of correlation to Boolean Networks with larger topologies.

Appendix A

Algorithms

A.1 Wolfram Language Algorithms

```
L = 100; (*sequence length*)
repsp = 10; (*times we use the same probability value p*)
dp = 0.01; (*probability steps' size*)

numexp = (1/dp); (*number of sequences*)
complexity = Table[0, repsp]; (*list to save the K-complexities for the same p value*)
entropy = Table[0, repsp];  (*list to save the entropies for the same p value*)
compress = Table[0, repsp]; (*list to save the file sizes of c. seqs. for the same p value*)
Ckolmo = Table[0, numexp + 1]; (*list to save the K-complexity for each p value*)
ShannonEntropy = Table[0, numexp + 1]; (*list to save the entropy for each p value*)
complexcompress = Table[0, numexp + 1];  (*list to save the file size for each p value*)

Do[
  Do[
    output = Table[RandomChoice[Rule[{1 - p, p}, {0, 1}]], L]; (*random binary sequence*)
    If[First[output] == 0, output = "0" <> ToString[FromDigits[output]], output =
ToString[FromDigits[output]]]; (*transform the list into a string*)

    complexity[[j]] = StringBDM[output];
    entropy[[j]] = Entropy[ToString[FromDigits[output]]];
    Export["C:\file_directory\name.txt.gz", output];
    compress[[j]] = QuantityMagnitude[FileSize["C:\file_directory\name.txt.gz"], "Bits"]
    , {j, repsp}];

    (*we perform a trimmed mean discarding the first and fourth quartiles*)
Ckolmo[[(p*numexp) + 1]] = TrimmedMean[complexity, {(Length[Select[complexity, # <
Quantile[complexity, 1/4] &]])/Length[complexity], (Length[Select[complexity, # >
Quantile[complexity, 3/4] &]])/Length[complexity]}];
  ShannonEntropy[[(p*numexp) + 1]] = TrimmedMean[entropy, {(Length[Select[entropy, # <
Quantile[entropy, 1/4] &]])/Length[entropy], (Length[Select[entropy, # > Quantile[entropy, 3/4]
&]])/Length[entropy]}];
  complexcompress[[(p*numexp) + 1]] = TrimmedMean[compress, {(Length[Select[compress, # <
Quantile[compress, 1/4] &]])/Length[compress], (Length[Select[compress, # > Quantile[compress,
3/4] &]])/Length[compress]}];
  , {p, 0, 1, dp}]

    (*we can plot the results*)
ListLinePlot[{Rescale[Ckolmo, {0, Max[Ckolmo]}], Rescale[ShannonEntropy, {0,
Max[ShannonEntropy]}], Rescale[complexcompress, {0, Max[complexcompress]}]}, DataRange -> {0, 1},
AxesLabel -> Automatic, PlotRange -> All, PlotLegends -> Placed[{"Kolmogorov Complexity",
"Entropy", "File Size (GZIP)"}, {.5, .2}], Frame -> True, FrameLabel -> {"p(0⇒1)", "C(S)"},
GridLines -> Automatic];
```

Figure A.1: The code to generate and measure the complexity of random sequences of bits.

```
n = 100; (*number of nodos*)
repsp = 10; (*times we use the same probability value p*)
dp = 0.01; (*probability steps' size*)

complexity = Table[0, repsp]; (*list to save K-complexity (1-D representation) for the same p
value*)
complexitymat = Table[0, repsp];  (*list to save K-complexity (adjacency matrix) for the same p
value*)
entropy = Table[0, repsp]; (*list to save entropy values for the same p value*)
seqnet = Table[0, (1/dp) + 1]; (*list to save the K-complexity for each p value (1-D
representation)*)
Matnet = Table[0, (1/dp) + 1]; (*list to save the K-complexity for each p value (2-D
representation)*)
ShannonEntropy = Table[0, (1/dp) + 1];  (*list to save the entropy for each p value*)

Do[
 Do[
  rg = RandomGraph[WattsStrogatzGraphDistribution[n, l, 5]]; (*random graph*)
  mat = Flatten[Normal[AdjacencyMatrix[rg]]]; (*1-D representation of the random graph*)
  If[First[mat] == 0, output = "0" <> ToString[FromDigits[mat]], output =
ToString[FromDigits[mat]]]; (*transform the list into a string*)

  (*measure the complexity using three different methods*)
  complexity[[j]] = StringBDM[output];
  complexitymat[[j]] = BDM[Normal[AdjacencyMatrix[rg]], 4] // N;
  entropy[[j]] = Entropy[output];
  , {j, repsp}];

 (*we perform a trimmed mean discarding the first and fourth quartiles*)
 seqnet[[(1/dp)*(l + dp)]] = TrimmedMean[complexity, {(Length[Select[complexity, # <
Quantile[complexity, 1/4] &]])/Length[complexity], (Length[Select[complexity, # >
Quantile[complexity, 3/4] &]])/Length[complexity]}];
 Matnet[[(1/dp)*(l + dp)]] = TrimmedMean[complexitymat, {(Length[Select[complexitymat, # <
Quantile[complexitymat, 1/4] &]])/Length[complexitymat], (Length[Select[complexitymat, # >
Quantile[complexitymat, 3/4] &]])/Length[complexitymat]}];
 ShannonEntropy[[(1/dp)*(l + dp)]] = TrimmedMean[entropy, {(Length[Select[entropy, # <
Quantile[entropy, 1/4] &]])/Length[entropy], (Length[Select[entropy, # > Quantile[entropy, 3/4]
&]])/Length[entropy]}];
 , {l, 0, 1, dp}]

(*we can plot the results*)
ListLinePlot[{Rescale[seqnet, {0, Max[seqnet]}], Rescale[Matnet, {0, Max[Matnet]}],
Rescale[ShannonEntropy, {0, Max[ShannonEntropy]}]}, DataRange -> {0, 1}, AxesLabel -> Automatic,
PlotRange -> All, PlotLegends -> Placed[{"K-Complexity (1-D representation)", "K-Complexity
(adjacency matrix)", "Entropy"}, {.5, .2}], Frame -> True, GridLines -> Automatic, FrameLabel ->
{"p", "C(G)"}]
```

Figure A.2: The code to generate and measure the complexity of random graphs using the Watts-Strogatz graph distribution.

```
n = 100; (*number of nodos*)
numexp = 100; (*number of random graphs generated*)
repsp = 10; (*times we use the same probability value p*)

complexity = Table[0, repsp]; (*list to save K-complexity (1-D representation) for the same p
value*)
complexitymat = Table[0, repsp];(*list to save K-complexity (adjacency matrix) for the same p
value*)
entropy = Table[0, repsp]; (*list to save entropy values for the same p value*)
ShannonEntropy = Table[0, numexp]; (*list to save the entropy for each p value*)
seqnet = Table[0, numexp]; (*list to save the K-complexity for each p value (1-D representation)*)
Matnet = Table[0, numexp]; (*list to save the K-complexity for each p value (2-D representation)*)

Do[
 Do[
  rg = RandomGraph[BarabasiAlbertGraphDistribution[100, l]]; (*random graph*)
  mat = Flatten[Normal[AdjacencyMatrix[rg]]]; (*1-D representation of the random graph*)
  If[First[mat] == 0, output = "0" <> ToString[FromDigits[mat]], output =
ToString[FromDigits[mat]]]; (*transform the list into a string*)

  complexity[[j]] = StringBDM[output];
  complexitymat[[j]] = BDM[Normal[AdjacencyMatrix[rg]], 4] // N;
  entropy[[j]] = Entropy[output];
  , {j, repsp}];

    (*we perform a trimmed mean discarding the first and fourth quartiles*)
seqnet[[l]] = TrimmedMean[complexity, {(Length[Select[complexity, # < Quantile[complexity, 1/4]
&]])/Length[complexity], (Length[Select[complexity, # > Quantile[complexity, 3/4]
&]])/Length[complexity]}];
 Matnet[[l]] = TrimmedMean[complexitymat, {(Length[Select[complexitymat, # <
Quantile[complexitymat, 1/4] &]])/Length[complexitymat], (Length[Select[complexitymat, # >
Quantile[complexitymat, 3/4] &]])/Length[complexitymat]}];
 ShannonEntropy[[l]] = TrimmedMean[entropy, {(Length[Select[entropy, # < Quantile[entropy, 1/4]
&]])/Length[entropy], (Length[Select[entropy, # > Quantile[entropy, 3/4] &]])/Length[entropy]}];
, {l, numexp}]

    (*we can plot the results*)
ListLinePlot[{Rescale[seqnet, {0, Max[seqnet]}], Rescale[Matnet, {0, Max[Matnet]}],
Rescale[ShannonEntropy, {0, Max[ShannonEntropy]}]}, AxesLabel -> Automatic, PlotRange -> All,
PlotLegends -> Placed[{"K-Complexity (1-D representation)", "K-Complexity (adjacency matrix)",
"Entropy"}, {.43, .16}], Frame -> True, GridLines -> Automatic, FrameLabel -> {"k", "C(G)"}]
```

Figure A.3: The code to generate and measure the complexity of random graphs using the Barabási-Albert graph distribution.

```
n = 100; (*number of nodos*)
maxindegree = 100; (*max in-degree*)
repsp = 10; (*times we use the same probability value p*)

complexity = Table[0, repsp];
(*list to save K-complexity (1-D representation) for the same p value*)
complexitymat = Table[0, repsp];
(*list to save K-complexity (adjacency matrix) for the same p value*)
entropy = Table[0, repsp]; (*list to save entropy values for the same p value*)
ShannonEntropy = Table[0, maxindegree]; (*list to save the entropy for each p value*)
seqnet = Table[0, maxindegree];
(*list to save the K-complexity for each p value (1-D representation)*)
Matnet = Table[0, maxindegree];
(*list to save the K-complexity for each p value (2-D representation)*)

Do[
 Do[
  (*create the random digraph*)
  k = l; (*in-degree*)
  Mnodes = {}; (*adjacency matrix*)
  For[i = 1, i ≤ n, i++×
     AppendTo[Mnodes, Table[0, n]]] ×
   For[i = 1, i ≤ n, i++,
    flag = 0;
    While[flag < k, position = RandomInteger[{1, n}];
     If[Mnodes[[i, position]] == 0, Mnodes[[i, position]] = 1; flag++,]]];

  rg = Transpose[Mnodes]; (*random digraph*)
  mat = Flatten[rg]; (*1-D representation of the random graph*)
  If[First[mat] == 0, output = "0" <> ToString[FromDigits[mat]],
   output = ToString[FromDigits[mat]]]; (*transform the list into a string*)

  complexity[[j]] = StringBDM[output];
  complexitymat[[j]] = BDM[rg, 4] // N;
  entropy[[j]] = Entropy[output];
  , {j, repsp}];

     (*perform a trimmed mean discarding the first and fourth quartiles*)
 seqnet[[l]] = TrimmedMean[complexity,
   {(Length[Select[complexity, # < Quantile[complexity, 1/4] &]])/Length[complexity],
    (Length[Select[complexity, # > Quantile[complexity, 3/4] &]])/Length[complexity]}];
 Matnet[[l]] = TrimmedMean[complexitymat,
   {(Length[Select[complexitymat, # < Quantile[complexitymat, 1/4] &]])/Length[complexitymat],
    (Length[Select[complexitymat, # > Quantile[complexitymat, 3/4] &]])/Length[complexitymat]}];
 ShannonEntropy[[l]] = TrimmedMean[entropy,
   {(Length[Select[entropy, # < Quantile[entropy, 1/4] &]])/Length[entropy],
    (Length[Select[entropy, # > Quantile[entropy, 3/4] &]])/Length[entropy]}];
 , {l, 1, maxindegree}]

    (*plot the results*)
ListLinePlot[{Rescale[seqnet, {0, Max[seqnet]}],
  Rescale[Matnet, {0, Max[Matnet]}], Rescale[ShannonEntropy, {0, Max[ShannonEntropy]}]},
 AxesLabel → Automatic, PlotRange → All, PlotLegends →
  Placed[{"K-Complexity (1-D representation)", "K-Complexity (adjacency matrix)", "Entropy"},
   {.5, .15}], Frame → True, GridLines → Automatic, FrameLabel → {"d⁻
```

Figure A.4: The code to generate and measure the complexity of random digraphs with increasing vertex in-degree.

```
n = 6; (*number of nodos*) k = 3; (*in-degree*) numexp = 100000; (*number of random digraphs to
generate*)

complexity = Table[0, numexp]; (*list for K-complexity (1-D representation)*)
complexitymat = Table[0, numexp]; (*list for K-complexity (adjacency matrix)*)
entropy = Table[0, numexp]; (*list for entropy values*)

Do[(*create the random digraph*)
 Mnodes = {};(*adjacency matrix*)
 For[i = 1, i <= n, i++
    AppendTo[Mnodes, Table[0, n]]]
  For[i = 1, i <= n, i++, flag = 0;
   While[flag < k, position = RandomInteger[{1, n}];
    If[Mnodes[[i, position]] == 0, Mnodes[[i, position]] = 1; flag++,]]];

 rg = Transpose[Mnodes]; (*random digraph*)
 mat = Flatten[rg]; (*1-D representation of the random graph*)

 If[First[mat] == 0, output = "0" <> ToString[FromDigits[mat]], output=
ToString[FromDigits[mat]]]; (*transform the list into a string*)
 complexity[[l]] = {l, StringBDM[output]};
 complexitymat[[l]] = {l, BDM[rg, 4] // N};
 entropy[[l]] = {l, Entropy[output]};
 Export[NotebookDirectory[] <> ToString[l] <> "file_name.txt", rg];
 , {l, 1, numexp}]

    (*Order the complxities by increasing value*)
sorted = Sort[complexity, #1[[2]] < #2[[2]] &] ;(*sorted list with positions and values*)
list = Flatten[First[sorted[[#]]] & /@ Table[i, {i, Length[sorted]}]] ;(*sorted list with
positions*)
data = Flatten[Take[sorted[[#]], {2, 2}] & /@ Table[i, {i, Length[sorted]}]];   (*sorted list
with values*)

sortedmat = Sort[complexitymat, #1[[2]] < #2[[2]] &];
listmat = Flatten[First[sortedmat[[#]]] & /@ Table[i, {i, Length[sortedmat]}]];
datamat = Flatten[Take[sortedmat[[#]], {2, 2}] & /@ Table[i, {i, Length[sortedmat]}]];

sortedent = Sort[entropy, #1[[2]] < #2[[2]] &] // N;
listent = Flatten[First[sortedent[[#]]] & /@ Table[i, {i, Length[sortedent]}]];
dataent = Flatten[Take[sortedent[[#]], {2, 2}] & /@ Table[i, {i, Length[sortedent]}]];

    (*plot the results*)
ListLinePlot[{Rescale[data, {0, Max[data]}], Rescale[datamat, {0, Max[datamat]}],
Rescale[dataent, {0, Max[dataent]}]}, TargetUnits -> {"experimento", "C(red)"}, AxesLabel ->
Automatic, PlotRange -> All, PlotLegends -> {"K-Complexity (1-D representation)", "K-Complexity
(adjacency matrix)", "Entropy"}, Frame -> True, GridLines -> Automatic, FrameLabel -> {"Ordered
Digraphs", "C(D)"}]

    (*draw some of the digraphs by increasing order of complexity*)
numdig = 27; (*number of digraphs to plot*) r = 1; (*flag*)
Table[If[net == Round[(numexp*r/numdig)] || net == 1,
   state = Import[NotebookDirectory[] <> ToString[list[[net]]] <> "file_name.txt", "Lines"]; r++;
   AdjacencyGraph[ToExpression[state], PlotLabel -> "Digraph " <> ToString[lista[[net]]]],
VertexStyle -> RGBColor[1, .78, .72], EdgeStyle -> Black]
  ,{net, 1, numexp}] /. Null -> Sequence[]
```

Figure A.5: The code to generate and measure the complexity of random digraphs with a fixed number of nodes and in-degree.

```
n = 8; (*number of nodos*)
k = 7; (*vertex degree*)
numperms = 400; (*number of random permutations*)

matrix = Normal[AdjacencyMatrix[RandomGraph[{n, k}]]]; (*random graph*)
graph= AdjacencyGraph[matrix, VertexStyle ->RGBColor[1, .78, .72], EdgeStyle ->Black] (*plot
the random graph*)

(*generate the isomorphisms*)
perms = Cycles[{#}] & /@ RandomChoice[Permutations[Table[i, {i, n}]], numperms];(*generators of
permutations*)
matpermuted = DeleteDuplicates[Permute[Table[Permute[matrix[[i]], #], {i, n}], #] & /@ perms]
;(*permuted matrices*)

complexity = Table[Null, Length[matpermuted] + 1]; (*list for K-complexity (1-D
representation)*)
complexitymat = Table[Null, Length[matpermuted] + 1]; (*list for K-complexity (adjacency
matrix)*)
entropy = Table[Null, Length[matpermuted] + 1]; (*list for entropy values*)

(*complexity of the random graph*)
mat = Flatten[matrix];
If[First[mat] == 0, output = "0" <> ToString[FromDigits[mat]], output =
ToString[FromDigits[mat]]];
complexity[[1]] = StringBDM[output] // N;
complexitymat[[1]] = BDM[rg, 4] // N;
entropy[[1]] = Entropy[output] // N;

(*measure the complexity of the isomorphisms*)
Do[
 rg = matpermuted[[l]];
 mat = Flatten[rg];
 If[First[mat] == 0, output = "0" <> ToString[FromDigits[mat]], output =
ToString[FromDigits[mat]]];
 complexity[[l + 1]] = StringBDM[output] // N;
 complexitymat[[l + 1]] = BDM[rg, 4] // N;
 entropy[[l + 1]] = Entropy[output] // N;
  If[ IsomorphicGraphQ[AdjacencyGraph[matrix], AdjacencyGraph[matpermuted[[l]]]] == False,
Abort[]] (*check isomorphism*)
 , {l, 1, Length[matpermuted]}]

(*plot the results*)
seqcomplexity = DeleteCases[complexity, Null];
matcomplexity = DeleteCases[complexitymat, Null];
ShannonEntropy = DeleteCases[entropy, Null];
ListLinePlot[{Rescale[seqcomplexity, {0, Max[seqcomplexity]}], Rescale[matcomplexity, {0,
Max[matcomplexity]}], Rescale[ShannonEntropy, {0, Max[ShannonEntropy]}]}, AxesLabel ->
Automatic, PlotRange -> {{0, Length[seqcomplexity]}, {0.4, 1.01}}, PlotLegends -> Placed[{"K-
Complexity (1-D representation)", "K-Complexity (adjacency matrix)", "Entropy"}, {.5, .2}],
Frame -> True, GridLines -> Automatic, FrameLabel -> {ToString[Length[seqcomplexity]] <> "
Isomorphic Graphs G'≅G", "C(G')"}]
```

Figure A.6: The code to generate and measure the complexity of the isomorphisms of a random graph. To work with a random digraph of n nodes and k directed edges instead of a graph of n nodes and k edges, the argument $DirectedEdges \to True$ has to be added in the function *RandomGraph*.

```
n = 6; (*number of nodos*) k = 3; (*in-degree*) numexp = 100000; (*number of random digraphs to
generate*) numperms = 1000; (*number of random permutations*)
complexity = Table[0, numexp]; (*list for K-complexity (1-D representation)*)
complexitymat = Table[0, numexp]; (*list for K-complexity (adjacency matrix)*)
entropy = Table[0, numexp]; (*list for entropy values*)

    (*create the random digraph*)
Do[Mnodes = {};(*adjacency matrix*)
 For[i = 1, i <= n, i++
    AppendTo[Mnodes, Table[0, n]]]
  For[i = 1, i <= n, i++, flag = 0;
   While[flag < k, position = RandomInteger[{1, n}];
    If[Mnodes[[i, position]] == 0, Mnodes[[i, position]] = 1; flag++,]]];
matadj = Transpose[Mnodes]; (*random digraph*)
Export[NotebookDirectory[]<>ToString[l]<>"file_name.txt",matadj];

    (*generate the isomorphisms*)
perms = Cycles[{#}] & /@ RandomChoice[Permutations[Table[i, {i, n}]], numperms];(*generators of
permutations*)
matpermuted = DeleteDuplicates[Permute[Table[Permute[matadj[[i]], #], {i, n}], #] & /@ perms]
;(*permuted matrices*)
complexitypermut = Table[Null, Length[matpermuted] + 1]; (*list for K-complexity (1-D
representation) permuted*)
complexitymatpermut = Table[Null, Length[matpermuted] + 1]; (*list for K-complexity (adjacency
matrix) permuted*)
entropypermut = Table[Null, Length[matpermuted] + 1]; (*list for entropy values permuted*)

    (*complexity of the random graph*)
matadjflatten = Flatten[matadj]; (*1-D representation of the random graph*)
If[First[matadjflatten] == 0, output = "0" <> ToString[FromDigits[matadjflatten]], output =
ToString[FromDigits[matadjflatten]]];
complexitypermut[[1]] = StringBDM[output] // N;
complexitymatpermut[[1]] = BDM[matadj, 4] // N;
entropypermut[[1]] = Entropy[output] // N;

    (*measure the complexity of the isomorphisms*)
Do[rg = matpermuted[[l]];
 mat = Flatten[rg];
 If[First[mat] == 0, output = "0" <> ToString[FromDigits[mat]], output =
ToString[FromDigits[mat]]];
 complexity[[l + 1]] = StringBDM[output] // N;
 complexitymat[[l + 1]] = BDM[rg, 4] // N;
 entropy[[l + 1]] = Entropy[output] // N;
  If[ IsomorphicGraphQ[AdjacencyGraph[matrix], AdjacencyGraph[matpermuted[[l]]]] == False,
Abort[]] (*check isomorphism*)
 , {l, 1, Length[matpermuted]}]
    (*the complexity is the minimun vale*)
   complexity[[l]] = {l, Min[complexitypermut]};
   complexitymat[[l]] = {l, Min[complexitymatpermut]};
   entropy[[l]] = {l, Min[entropypermut]};
   Export[NotebookDirectory[] <> ToString[l] <> "Vecinos_red.txt", matadj];
   , {l, 1, numexp}]

    (*order the complxities by increasing value*)
sorted = Sort[complexity, #1[[2]] < #2[[2]] &] ;
list = Flatten[First[sorted[[#]]] & /@ Table[i, {i, Length[sorted]}]] ;
data = Flatten[Take[sorted[[#]], {2, 2}] & /@ Table[i, {i, Length[sorted]}]];

sortedmat = Sort[complexitymat, #1[[2]] < #2[[2]] &];
listmat = Flatten[First[sortedmat[[#]]] & /@ Table[i, {i, Length[sortedmat]}]];
datamat = Flatten[Take[sortedmat[[#]],{2, 2}]&/@Table[i,{i, Length[sortedmat]}]];

sortedent = Sort[entropy, #1[[2]] < #2[[2]] &] // N;
listent = Flatten[First[sortedent[[#]]] & /@ Table[i, {i, Length[sortedent]}]];
dataent = Flatten[Take[sortedent[[#]],{2,2}]&/@Table[i, {i, Length[sortedent]}]];

    (*plot the results*)
ListLinePlot[{Rescale[data, {0, Max[data]}], Rescale[datamat, {0, Max[datamat]}],
Rescale[dataent, {0, Max[dataent]}]}, TargetUnits -> {"experimento", "C(red)"}, AxesLabel ->
Automatic, PlotRange -> All, PlotLegends -> {"K-Complexity (1-D representation)", "K-Complexity
(adjacency matrix)", "Entropy"}, Frame -> True, GridLines -> Automatic, FrameLabel -> {"Ordered
Digraphs", "C(D)"}]

    (*draw some of the digraphs by increasing order of complexity*)
numdig = 27; (*number of digraphs to plot*) r = 1; (*flag*)
Table[If[net == Round[(numexp*r/numdig)] || net == 1,
   state = Import[NotebookDirectory[] <> ToString[list[[net]]] <> "file_name.txt", "Lines"]; r++;
   AdjacencyGraph[ToExpression[state], PlotLabel -> "Digraph " <> ToString[lista[[net]]]],
VertexStyle -> RGBColor[1, .78, .72], EdgeStyle -> Black]
  ,{net, 1, numexp}] /. Null -> Sequence[]
```

Figure A.7: The code to generate and measure the complexity of random digraphs with a fixed number of nodes and in-degree by considering its isomorphisms.

```
maxk = 14; (*max k parameter*) n = 10; (*number of functions in each set*)
repsp = 10; (*times we use the same probability value k*)

complexity = Table[0, repsp]; (*list of K-complexity (1-D representation) for the same k value*)
complexitymat = Table[0, repsp];
(*list of K-complexity (2-D representation) for the same k value*)
entropy = Table[0, repsp]; (*list ofentropy values for the same k value*)
ShannonEntropy = Table[0, maxk]; (*list to save the entropy for each k value*)
seqnet = Table[0, maxk];
(*list to save the K-complexity for each k value (1-D representation)*)
Matnet = Table[0, maxk]; (*list to save the K-complexity for each k value (2-D representation)*)

Do[Do[k = l; (*parameter k*)
  booleanfunction = {}; (*boolean function*)
  Matrixbooleanfunctionlist = {}; (*matrix representation*)
  Clear[a];
  A = Table[a[i], {i, 1, k}]; (*boolean variables*)

      (*generate the set of boolean functions*)
  functions = Table[0, n];
  For[i = 0, i < n, i++,
   f = BooleanFunction[RandomInteger[{0, (2^2^k - 1)}], k];
   (*choose randomly one of the 2^2^k possible boolean functions*)
   AppendTo[booleanfunction, f];
   AppendTo[Matrixbooleanfunctionlist,
    Boole[BooleanTable[BooleanConvert[Apply[f, A], "NOR"], A]]];
   functions[[i + 1]] = f];

      (*measure the complexity of the set*)
  matrixrepresen = Matrixbooleanfunctionlist;
  sequencerepresen = Flatten[matrixrepresen];
  If[First[sequencerepresen] == 0, output = "0" <> ToString[FromDigits[sequencerepresen]],
   output = ToString[FromDigits[sequencerepresen]]];
  complexity[[j]] = StringBDM[output];
  complexitymat[[j]] = BDM[matrixrepresen, 4] // N;
  entropy[[j]] = Entropy[output];
  , {j, repsp}];

    (*perform a trimmed mean discarding the first and fourth quartiles*)
 seqnet[[l]] = TrimmedMean[complexity,
   {(Length[Select[complexity, # < Quantile[complexity, 1/4] &]])/Length[complexity],
    (Length[Select[complexity, # > Quantile[complexity, 3/4] &]])/Length[complexity]}];
 Matnet[[l]] = TrimmedMean[complexitymat,
   {(Length[Select[complexitymat, # < Quantile[complexitymat, 1/4] &]])/Length[complexitymat],
    (Length[Select[complexitymat, # > Quantile[complexitymat, 3/4] &]])/Length[complexitymat]}];
 ShannonEntropy[[l]] = TrimmedMean[entropy,
   {(Length[Select[entropy, # < Quantile[entropy, 1/4] &]])/Length[entropy],
    (Length[Select[entropy, # > Quantile[entropy, 3/4] &]])/Length[entropy]}];
 , {l, 1, maxk}]

    (*plot the results*)
ListLinePlot[{Rescale[seqnet, {0, Max[seqnet]}], Rescale[Matnet, {0, Max[Matnet]}],
  Rescale[ShannonEntropy, {0, Max[ShannonEntropy]}]}, AxesLabel → Automatic, PlotRange → All,
 PlotLegends → Placed[{"K-Complexity (1-D)", "K-Complexity (2-D)", "Entropy"}, {.25, .3}],
 Frame → True, GridLines → Automatic, FrameLabel → {"k", "C(f)"}]
```

Figure A.8: The code to generate and measure the complexity of random sets of boolean functions with increasing parameter k.

```
n = 4; (*number of functions in each set*)
k = 2 (*parameter k*) numexp = 40 000; (*number of sets to generate*)
numperms = 16; (*number of permutations to test*)

complexity = Table[0, numexp]; (*list for K-complexity (1-D representation)*)
complexitymat = Table[0, numexp]; (*list for K-complexity (2-D representation)*)
entropy = Table[0, numexp]; (*list for entropy values*)

Do[booleanfunction = {};
 Matrixbooleanfunctionlist = {}; (*matrix representation*) Clear[a];
 A = Table[a[i], {i, 1, k}]; (*boolean variables*)
     (*generate the set of boolean functions*)
 functions = Table[0, n];
 For[i = 0, i < n, i++,
  f = BooleanFunction[RandomInteger[{0, (2^2^k - 1)}], k];
  (*choose randomly one of the 2^2^k possible boolean functions*)
  AppendTo[booleanfunction, f];
  AppendTo[Matrixbooleanfunctionlist,
   Boole[BooleanTable[BooleanConvert[Apply[f, A], "NOR"], A]]];
  functions[[i + 1]] = f];

     (*generate the isomorphisms*)
 matrixrepresen = Matrixbooleanfunctionlist;
 perms = RandomChoice[Permutations[Table[i, {i, n}]], numperms];
 isos = DeleteDuplicates[Permute[matrixrepresen, #] & /@ perms];
 complexityiso = Table[0, Length[isos]];
 complexitymatiso = Table[0, Length[isos]];
 entropyiso = Table[0, Length[isos]];
     (*measure the complexity of the isomorphisms*)
 Do[flatten = Flatten[isos[[L]]];
  If[First[flatten] == 0,
   output = "0" <> ToString[FromDigits[flatten]], output = ToString[FromDigits[flatten]]];
  complexityiso[[L]] = StringBDM[output];
  complexitymatiso[[L]] = BDM[isos[[L]], 4] // N;
  entropyiso[[L]] = Entropy[output], {L, Length[isos]}];
     (*the true complexity value is the minimum value obtained from the isos*)
 complexity[[l]] = {l, Min[complexityiso]};
 complexitymat[[l]] = {l, Min[complexitymatiso]};
 entropy[[l]] = {l, Min[entropyiso]};
 Export[NotebookDirectory[] <> ToString[l] <> "file_name.txt", Matrixbooleanfunctionlist],
 {l, 1, numexp}]

    (*sort the results in increasing complexity order*)
sorted = Sort[complexity, #1[[2]] < #2[[2]] &]
list = Flatten[First[sorted[[#]]] & /@ Table[i, {i, Length[sorted]}]];
data = Flatten[Take[sorted[[#]], {2, 2}] & /@ Table[i, {i, Length[sorted]}]];
sortedmat = Sort[complexitymat, #1[[2]] < #2[[2]] &]
datamat = Flatten[Take[sortedmat[[#]], {2, 2}] & /@ Table[i, {i, Length[sortedmat]}]];
sortedent = Sort[entropy, #1[[2]] < #2[[2]] &] // N
dataent = Flatten[Take[sortedent[[#]], {2, 2}] & /@ Table[i, {i, Length[sortedent]}]];

    (*plot the sorted results*)
ListLinePlot[{Rescale[data, {0, Max[data]}], Rescale[datamat, {0, Max[datamat]}],
  Rescale[dataent, {0, Max[dataent]}]}, AxesLabel → Automatic, PlotRange → All,
 PlotLegends → Placed[{"K-Complexity (1-D)", "K-Complexity (2-D)", "Entropy"}, {0.75, 0.3}],
 Frame → True, GridLines → Automatic, FrameLabel → {"Ordered Sets", "C(f)"}]

    (*plot some sets in increasing complexity order*)
numsets = 16; (*number of sets to visualize*)
r = 1; sets = {};
Table[If[nset == Round[(numexp * r / numsets)] || nset == 1, r++;
   statemat =
    ReadList[NotebookDirectory[] <> ToString[list[[nset]]] <> "file_name.txt", Expression];
   AppendTo[sets, list[[nset]]];
   Show[MatrixPlot[statemat, ColorFunction → "Monochrome"]]
  ], {nset, 1, numexp}] /. Null → Sequence[]
Print["sets by increasing complexity order = ", sets]
```

Figure A.9: The code to generate and measure the complexity of random sets of boolean functions with fixed parameter k and N.

```
numberofRBN = 2000; (*number of RBN's with the same topology*)
numberofexps = 50; (*number of topologies to test*)
numperms = 5000; (*random permuations of the topology to test*)
n = 7; (*number of nodes*) k = 4; (*in-degree*)

complexityRBNtrimmed = Table[0, numberofexps];
complexityGraphtrimmed = Table[0, numberofexps];
complexityRBN = Table[0, numberofRBN];

    (*generate the random topology*)
Do[Mnodes = Table[Table[0, n], n];
 For[i = 1, i ≤ n, i++, Flag = 0;
  While[Flag < k, position = RandomInteger[{1, n}];
   If[Mnodes[[i, position]] == 0, Mnodes[[i, position]] = 1; Flag++]]];
 Matrixnodes = Transpose[Mnodes];

     (*generate the updating functions*) r = 0;
 Do[r++; iterations = 1; booleanfunction = {}; booleanfunctionMatrix = {}; Clear[a];
  A = Table[a[i], {i, 1, k}];
  For[i = 0, i < n, i++, f = BooleanFunction[RandomInteger[{0, (2^k - 1)}], k];
   AppendTo[booleanfunction, f];
   AppendTo[booleanfunctionMatrix, BooleanConvert[Apply[f, A], "NOR"]]] ;

      (*generate the dynamics of the network*)
  inputstates = Tuples[{0, 1}, n]; (*possible input states*)
  outputstates = Table[Table[0, iterations], 2^n]; (*output states*)
  For[q = 1, q ≤ 2^n, q++, neighbors = Table[0, k]; (*positions of the k-neighbors*)
   For[i = 1, i ≤ iterations, i++, statesnodes = Table[0, n]; (*states of the nodes*)
    For[j = 1, j ≤ n, j++ , (*run over the nodes*) Flag = 0;
     For[p = 1, p ≤ n, p++, (*find the k-neighbors*)
      If[Matrixnodes[[p, j]] == 1, Flag++; neighbors[[Flag]] = p]];
     For[m = 1, m ≤ k , m++, (*run over the in-degrees*)a[m] = inputstates[[q, neighbors[[m]]]]];
     statesnodes[[j]] = FullSimplify[booleanfunctionMatrix[[j]]]];
    (*transform the states of the nodes into the state of the network*)
    outputstates[[q]] = FullSimplify[Boole[statesnodes]];
    inputstates[[q]] = FullSimplify[Boole[statesnodes]]];

      (*measure the complexity of the RBN*)
  If[First[Flatten[outputstates]] == 0,
   codedinamica = "0" <> ToString[FromDigits[Flatten[outputstates]]],
   codedinamica = ToString[FromDigits[Flatten[outputstates]]]];
  complexityRBN[[r]] = StringBDM[codedinamica, numberofRBN];

     (*measure the mean complexity of the topology*)
 matrix = Matrixnodes; mat = Flatten[matrix];
 perms = Cycles[{#}] & /@ RandomChoice[Permutations[Table[i, {i, n}]], numperms];
 permutedMatrix =
  DeleteDuplicates[Permute[Table[Permute[matrix[[i]], #], {i, n}], #] & /@ perms] ;
 complexity = Table[Null, Length[permutedMatrix] + 1];
 If[First[mat] == 0, output = "0" <> ToString[FromDigits[mat]], output = ToString[FromDigits[mat]]];
 complexity[[1]] = StringBDM[output] // N;
 Do[rg = permutedmat[[l]]; mat = Flatten[rg];
  If[First[mat] == 0, output = "0" <> ToString[FromDigits[mat]], output = ToString[FromDigits[mat]]];
  complexity[[l + 1]] = StringBDM[output], {l, 1, Length[permutedmat]}];
 MatNet = DeleteCases[complexity, Null];
 complexityGraphtrimmed[[y]] =
  TrimmedMean[MatNet, {(Length[Select[MatNet, # ≤ Quantile[MatNet, 1/4] &]])/Length[MatNet],
    (Length[Select[MatNet, # ≥ Quantile[MatNet, 3/4] &]])/Length[MatNet]}];

     (*mean complexity of the RBN's*)
 complexityRBNtrimmed[[y]] = TrimmedMean[complexityRBN,
   {(Length[Select[complexityRBN, # ≤ Quantile[complexityRBN, 1/4] &]])/Length[complexityRBN],
    (Length[Select[complexityRBN, # ≥ Quantile[complexityRBN, 3/4] &]])/
     Length[complexityRBN]}], {y, numberofexps}]
complexityGraphtrimmed;
```

Figure A.10: The code to generate and measure the complexity of Random Boolean Networks which share the same topology.

```
numberofRBN = 2000; (*number of RBN's with the same topology*)
numberofexps = 50; (*number of topologies to test*)
numperms = 5000; (*random permuations of the updating functions to test*)
n = 5; (*number of nodes*) k = 3; (*in-degree*)

complexityRBNtrimmed = Table[0, numberofexps];
complexityBoolFun = Table[0, numberofexps];
complexityRBN = Table[0, numberofRBN];

    (*generate the updating functions*)
Do[booleanfunction = {}; booleanfunctionMatrix = {}; Clear[a];
 A = Table[a[i], {i, 1, k}];
 For[i = 0, i < n, i++,
  f = BooleanFunction[RandomInteger[{0, (2^2^k - 1)}], k];
  AppendTo[booleanfunction, f];
  AppendTo[booleanfunctionsMatrixList,
    Boole[BooleanTable[BooleanConvert[Apply[f, A], "NOR"], A]]] ×
   AppendTo[booleanfunctionMatrix, BooleanConvert[Apply[f, A], "NOR"]]];

 r = 0; (*generate the topology*)
 Do[r++; iterations = 1;
  Mnodes = Table[Table[0, n], n];
  For[i = 1, i ≤ n, i++, Flag = 0;
   While[Flag < k, position = RandomInteger[{1, n}];
    If[Mnodes[[i, position]] == 0, Mnodes[[i, position]] = 1; Flag++]]];
  Matrixnodes = Transpose[Mnodes];

      (*generate the dynamics of the network*)
  inputstates = Tuples[{0, 1}, n]; (*possible input states*)
  outputstates = Table[Table[0, iterations], 2^n]; (*output states*)
  For[q = 1, q ≤ 2^n, q++,
   neighbors = Table[0, k]; (*positions of the k-neighbors*)
   For[i = 1, i ≤ iterations, i++,
    statesnodes = Table[0, n]; (*states of the nodes*)
    For[j = 1, j ≤ n, j++, (*run over the nodes*) Flag = 0;
     For[p = 1, p ≤ n, p++, (*find the k-neighbors*)
      If[Matrixnodes[[p, j]] == 1, Flag++;
       neighbors[[Flag]] = p]];
     For[m = 1, m ≤ k, m++,
      (*run over the in-degrees*)a[m] = inputstates[[q, neighbors[[m]]]];
     statesnodes[[j]] = FullSimplify[booleanfunctionMatrix[[j]]]];
    (*transform the states of the nodes into the state of the network*)
    outputstates[[q]] = FullSimplify[Boole[statesnodes]];
    inputstates[[q]] = FullSimplify[Boole[statesnodes]]];

      (*measure the complexity of the RBN*)
  If[First[Flatten[outputstates]] == 0,
   codedynamics = "0" <> ToString[FromDigits[Flatten[outputstates]]],
   codedynamics = ToString[FromDigits[Flatten[outputstates]]]];
  complexityRBN[[r]] = StringBDM[codedynamics], numberofRBN];

     (*measure the complexity of the updating functions*)
 rg = booleanfunctionsMatrixList; mat = Flatten[rg];
 perms = RandomChoice[Permutations[Table[i, {i, n}]], numperms];
 isos = DeleteDuplicates[Permute[rg, #] & /@ perms] ;
 complexityiso = Table[0, Length[isos]];
 Do[flatten = Flatten[isos[[L]]];
  If[First[flatten] == 0,
   output = "0" <> ToString[FromDigits[flatten]], output = ToString[FromDigits[flatten]]];
  complexityiso[[L]] = StringBDM[output], {L, Length[isos]}];
 complexityBoolFun[[y]] = Min[complexityiso];

     (*mean complexity of the RBN's*)
 complexityRBNtrimmed[[y]] = TrimmedMean[complexityRBN,
   {(Length[Select[complexityRBN, # ≤ Quantile[complexityRBN, 1/4] &]])/Length[complexityRBN],
    (Length[Select[complexityRBN, # ≥ Quantile[complexityRBN, 3/4] &]])/
     Length[complexityRBN]}], {y, numberofexps}]
Export[NotebookDirectory[] <> "C(f)_min.txt", complexityBoolFun];
Export[NotebookDirectory[] <> "C(BN)_trim.txt", complexityRBNtrimmed];
```

Figure A.11: The code to generate and measure the complexity of Random Boolean Networks which share the same updating functions.

A.2 Python Algorithms

```
import keras
import numpy as np
from bdm import BDM
import random
from keras.preprocessing.sequence import pad_sequences

# Initialize BDM object
# ndim argument specifies dimensionality of BDM
bdm = BDM (ndim = 1)

length_max = 100
length_min = 21
num = 30000 #number of sequences to generate
X = [] #Training vector
Y = np.zeros ((num + 1, 1)) #Target Vector


for i in range (num + 1) :
    length = random.randint (length_min, length_max) #length of the sequence
    ones_prob = random.randint(0, 100)/100 #probability parameter
    if ones_prob > 0.95 : #increase the probability of a sequence full of 1s
        sec = np.random.choice (2, length, p = [0, 1]).reshape (1, length) #random sequence
        X.append (sec[0].tolist ())
        Y[i] = bdm.bdm (sec[0])
    else :
        prob = random.randint (0, 100)/100 #probability parameter
        sec = np.random.choice (2, length, p = [1 - prob, prob]).reshape (1, length)
        X.append (sec[0].tolist ())
        Y[i] = bdm.bdm (sec[0])

X = pad_sequences (X, value = -1) #padding procedure
```

Figure A.12: The code to generate training data for the Neural Network which predicts Kolmogorov complexity.

```
from keras import Sequential
from keras.layers import Dense
from keras.layers import ELU

def build_regressor () :
    regressor = Sequential ()
    regressor.add (Dense (units = length_max, input_shape = (length_max,)))
    regressor.add (ELU (alpha = 1))
    regressor.add (Dense (units = int (round (2*(length_max)/3))))
    regressor.add (ELU (alpha = 1))
    regressor.add (Dense (units = 1, activation = ' linear'))
    regressor.compile (optimizer = ' Nadam', loss = ' mean_squared _error', metrics = ['
mean_squared _error', ' mean_absolute _error', "mean_absolute_percentage_error"])
    return regressor

build_regressor ().summary ()
```

Figure A.13: The code to build with the library Keras a Neural Network to perform regression.

```
import keras
from keras.preprocessing.sequence import pad_sequences
import math
import collections
import numpy as np

length_max = 1001 #max length of NN 1
length_max _ 2 = 100 #max length of NN 2
length_max _ 3 = 20 #max length of NN 3
splitted_length=500 #length of the splitted sequences

def kolmo (seq, counter = 1, complexity = 0) :
    if (len (seq[0]) < length_max) :
        if (len (seq[0]) > length_max _ 2) :
            sequence = pad_sequences (seq, value = -1, maxlen = length_max)
            complexity = NN_ 1000. predict (sequence) + math.log (counter, 2) #BDMNN
        elif (len (seq[0]) <= length_max _ 2 and len (seq[0]) > length_max _ 3) :
            sequence = pad_sequences (seq, value = -1, maxlen = length_max _ 2)
            complexity = NN_ 100. predict (sequence) + math.log (counter, 2)
        elif (len (seq[0]) <= length_max _ 3 and len (seq[0]) > 11) :
            sequence = pad_sequences (seq, value = -1, maxlen = length_max _ 3)
            complexity = NN_ 20. predict (sequence)[0] + math.log (counter, 2)
        else :
            complexity = 2.285794 #complexity of sequences 0 and 1
    elif (len (seq[0]) >= length_max) :
        splitted = np.array_split (seq[0], math.ceil (len (seq[0])/(splitted_length)))
        splitted_tuple = map (tuple, splitted)
        counts = collections.Counter (splitted_tuple) #count repeated sequences
        complexity_list = list (map (kolmo, [splitted], counts.values ()))
        complexity = sum (complexity_list[0])
    else :
        complexity = 2.285794 #complexity of sequences 0 and 1
    return complexity
```

Figure A.14: The code to define a function which takes advantage of three Neural Network models to predict the K-complexity of sequences of any length. The functions NN_20, NN_100, and NN_1000 are the Neural Networks trained to predict the complexity of sequences in the corresponding interval of lengths.

```
import time

long_max = 10000 #number of sequences to consider
time_lib = np.zeros ((long_max - 11)) #time for BDM
time_mi _lib = np.zeros ((long_max - 11)) #time for BDMNN
lib = np.zeros ((long_max - 11)) #complexity by BDM
mi_lib = np.zeros ((long_max - 11)) #complexity by BDMNN
longs = np.zeros ((long_max - 11))

for i in range (12, long_max + 1) :
    prueba = np.random.choice (2, i).reshape (1, i)
    start = time.process_time () #start timimng for BDMNN
    mi_lib[i - 12] = kolmo (prueba)
    time_mi _lib[i - 12] = time.process_time () - start
    start = time.process_time () #start timimng for BDM
    lib[i - 12] = bdm.bdm (prueba[0])
    time_lib[i - 12] = time.process_time () - start
    longs[i - 12] = i
```

Figure A.15: The code to perform the timing of the BDM and the BDMNN.

```
import time
import numpy as np
from joblib import Parallel, delayed
from bdm import BDM
from bdm.utils import slice_dataset

bdm = BDM (ndim = 1)

kernels = 4 #numer of kernels
long_max = 5000 #number of sequences to consider
time_lib = np.zeros ((long_max - 11)) #time for BDM
time_mi _lib = np.zeros ((long_max - 11)) #time for BDMNN
lib = np.zeros ((long_max - 11)) #complexity by BDM
mi_lib = np.zeros ((long_max - 11)) #complexity by BDMNN
longs = np.zeros ((long_max - 11))

for i in range (12, long_max + 1) :
    prueba = np.random.choice (2, kernels*i).reshape (1, kernels*i)
    start = time.process_time ()
    mi_lib[i - 12] = kolmo (prueba)
    time_mi _lib[i - 12] = time.process_time () - start

    if len (prueba[0]) >= 48 :
        counters_demo = Parallel (n_jobs = 4) (delayed (bdm.count_and _lookup) (d) for d in
    slice_dataset (prueba[0], (round (len (prueba[0])/kernels),)))
        start = time.process_time ()
        lib[i - 12] = bdm.compute_bdm(*counters_demo) time_lib[i-12]=time.process_time()-
    start
    else:
        start=time.process_time() lib[i-12]=bdm.bdm(prueba[0])
        time_lib[i-12]=time.process_time()-start longs[i-12]=kernels*i
```

Figure A.16: The code to perform the timing of the BDMNN and the parallel implementation of the BDM. For the parallel implementation, it does not consider the time needed to split and distribute among the available kernels the given sequence. If we want to take into account this extra time we just need to move the line: $start = time.process\_time()$ of the *if* section, immediately before the declaration of $counters\_demo$.

References

[1] Douglas, B. W. *Introduction to Graph Theory*. USA: Prentice-Hall, 1996. ISBN: 0-13-227828-6.

[2] Wilson, R. J. and Watkins, J. J. *Graphs and Introductory Approach*. USA: John Wiley & Sons, 1990. ISBN: 0-471-61554-4.

[3] Reinhard, D. *Graph Theory*. USA: Springer-Verlag New York, 1997. ISBN: 0-387-98211-6.

[4] Weisstein, Eric W. "Multigraph." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Multigraph.html.

[5] Trudeau, R. J. *Introduction to Graph Theory*. USA: Dover Publications, 1976. ISBN: 0-486-67870-9.

[6] Weisstein, Eric W. "Automorphism Group." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/AutomorphismGroup.html.

[7] Wolfram. *GraphData Source Information*. URL: `https://reference.wolfram.com/language/note/GraphDataSourceInformation.html`.

[8] Weisstein, Eric W. "Hypercube Graph." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HypercubeGraph.html.

[9] Ore, O. *Graphs and their uses*. USA: Random House, 1963. ISBN: 394-01566-5.

[10] Weisstein, Eric W. "Circulant Graph." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/CirculantGraph.html.

[11] Weisstein, Eric W. "Adjacency Matrix." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/AdjacencyMatrix.html.

[12] Krivelevich, Michael and Sudakov, Benny. "Pseudo-random graphs". In: *arXiv Mathematics e-prints*, math/0503745 (Mar. 2005), math/0503745. arXiv: `math/0503745 [math.CO]`.

[13] Watts, Duncan and Strogatz, Steven. "Collective dynamics of "small-world" networks". In: *Nature* 393.June (1998), pp. 440–442.

[14] Small, Michael. "Scale-Free Network." From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein. http://mathworld.wolfram.com/Scale-FreeNetwork.html.

[15] Barabási, Albert-László and Albert, Réka. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999), pp. 509–512. ISSN: 0036-8075. DOI: `10.1126/science.286.5439.509`. eprint: `https://science.sciencemag.org/content/286/5439/509.full.pdf`. URL: `https://science.sciencemag.org/content/286/5439/509`.

[16] Kauffman, S.A. "Metabolic stability and epigenesis in randomly constructed genetic nets". In: *Journal of Theoretical Biology* 22.3 (1969), pp. 437–467. ISSN: 0022-5193. DOI: https://doi.org/10.1016/0022-5193(69)90015-0. URL: http://www.sciencedirect.com/science/article/pii/0022519369900150.

[17] Kauffman, S. "Homeostasis and Differentiation in Random Genetic Control Networks". In: *Nature* 224.5215 (Oct. 1969), pp. 177–178. DOI: 10.1038/224177a0. URL: https://doi.org/10.1038/224177a0.

[18] He, Qinbin, Chen, Fangyue, and Liu, Zengrong. "Boolean networks: Coding, linearizing and dynamics". In: *Proceedings - 2010 International Workshop on Chaos-Fractal Theories and Applications, IWCFTA 2010* 1.1 (2010), pp. 216–220. DOI: 10.1109/IWCFTA.2010.33.

[19] Greil, F. "Dynamics of Boolean Networks". (Doctoral dissertation, Technische Universitat Darmstadt). Retrieved from http://tuprints.ulb.tu-darmstadt.de/1905/2/FlorianGreil_PhDThesis_TuDarmstadt.pdf. (2009).

[20] Drossel, Barbara. "Random Boolean Networks". In: *arXiv e-prints*, arXiv:0706.3351 (June 2007), arXiv:0706.3351. arXiv: 0706.3351 [cond-mat.stat-mech].

[21] Gershenson, Carlos. "Introduction to Random Boolean Networks". In: *arXiv e-prints*, nlin/0408006 (Aug. 2004), nlin/0408006. arXiv: nlin/0408006 [nlin.AO].

[22] Aldana, Maximino. "Boolean dynamics of networks with scale-free topology". In: *Physica D: Nonlinear Phenomena* 185.1 (2003), pp. 45–66. ISSN: 01672789. DOI: 10.1016/S0167-2789(03)00174-X.

[23] Dueck, R. *Digital design with CPLD applications and VHDL*. USA: Delmar Cengage Learning, 2000. ISBN: 978-0766811607.

[24] Weisstein, Eric W. "Connective." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Connective.html.

[25] Weisstein, Eric W. "Boolean Function." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BooleanFunction.html.

[26] Scharle, Thomas W. "Axiomatization of propositional calculus with sheffer functors". In: *Notre Dame Journal of Formal Logic* 6.3 (1965), pp. 209–217. ISSN: 19390726. DOI: 10.1305/ndjfl/1093958259.

[27] Hopfensitz, Martin et al. "Attractors in Boolean networks: A tutorial". In: *Computational Statistics* 28.1 (2013), pp. 19–36. ISSN: 09434062. DOI: 10.1007/s00180-012-0324-2.

[28] Gershenson, Carlos. "Guiding the self-organization of random Boolean networks". In: *Theory in Biosciences* 131.3 (2012), pp. 181–191. ISSN: 14317613. DOI: 10.1007/s12064-011-0144-x.

[29] Alberts, B. et al. *Molecular Biology of the Cell*. New York, USA: Garland Science, Taylor and Francis Group, 2008. ISBN: 978-0-8153-4105-5.

[30] Gershenson, Carlos. "Classification of Random Boolean Networks". In: *Proceedings of the Eighth International Conference on Artificial Life*. ICAL 2003. Cambridge, MA, USA: MIT Press, 2003, pp. 1–8. ISBN: 0-262-69281-3. URL: http://dl.acm.org/citation.cfm?id=860295.860297.

[31] Gershenson, Carlos. "Updating Schemes in Random Boolean Networks: Do They Really Matter?" In: *arXiv e-prints*, nlin/0402006 (Feb. 2004), nlin/0402006. arXiv: nlin/0402006 [nlin.AO].

[32] Dubrova, Elena, Teslenko, Maxim, and Martinelli, Andres. "Kauffman networks: Analysis and applications". In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD* 2005 (2005), pp. 478–483. ISSN: 10923152. DOI: 10.1109/ICCAD.2005.1560115.

[33] Kauffman, Stuart et al. "Random Boolean network models and the yeast transcriptional network". In: *Proceedings of the National Academy of Sciences of the United States of America* 100.25 (2003), pp. 14796–14799. ISSN: 00278424. DOI: 10.1073/pnas.2036429100.

[34] Li, Fangting et al. "The yeast cell-cycle network is robustly designed". In: *Proceedings of the National Academy of Sciences* 101.14 (2004), pp. 4781–4786. ISSN: 0027-8424. DOI: 10.1073/pnas.0305937101. eprint: https://www.pnas.org/content/101/14/4781.full.pdf. URL: https://www.pnas.org/content/101/14/4781.

[35] Cheng, Daizhan, Qi, Hongsheng, and Li, Zhiqiang. "Model construction of Boolean network via observed data". In: *IEEE Transactions on Neural Networks* 22.4 (2011), pp. 525–536. ISSN: 10459227. DOI: 10.1109/TNN.2011.2106512.

[36] Xiao, Yufei. "A Tutorial on Analysis and Simulation of Boolean Gene Regulatory Network Models". In: *Current Genomics* 10.7 (2009), pp. 511–525. ISSN: 13892029. DOI: 10.2174/138920209789208237.

[37] Fumiã, Herman F. and Martins, Marcelo L. "Boolean Network Model for Cancer Pathways: Predicting Carcinogenesis and Targeted Therapy Outcomes". In: *PLoS ONE* 8.7 (2013). ISSN: 19326203. DOI: 10.1371/journal.pone.0069008.

[38] Bornholdt, Stefan. "Boolean network models of cellular regulation: Prospects and limitations". In: *Journal of the Royal Society Interface* 5.SUPPL. 1 (2008). ISSN: 17425662. DOI: 10.1098/rsif.2008.0132.focus.

[39] Albert, Réka and Othmer, Hans G. "The topology of the regulatory interactions predicts the expression pattern of the segment polarity genes in Drosophila melanogaster". In: *Journal of Theoretical Biology* 223.1 (2003), pp. 1–18. ISSN: 0022-5193. DOI: https://doi.org/10.1016/S0022-5193(03)00035-3. URL: http://www.sciencedirect.com/science/article/pii/S0022519303000353.

[40] Fauré, Adrien et al. "Dynamical analysis of a generic Boolean model for the control of the mammalian cell cycle". In: *Bioinformatics* 22.14 (2006), pp. 124–131. ISSN: 13674803. DOI: 10.1093/bioinformatics/btl210.

[41] Huepe, Cristián and Aldana-González, Maximino. "Dynamical Phase Transition in a Neural Network Model with Noise: An exact Solution". In: *Journal of Statistical Physics* 108.3/4 (2002), pp. 527–540. DOI: 10.1023/a:1015777824097. URL: https://doi.org/10.1023/a:1015777824097.

[42] Schelling, Thomas C. "Dynamic models of segregation". In: *The Journal of Mathematical Sociology* 1.2 (1971), pp. 143–186. DOI: 10.1080/0022250X.1971.9989794. eprint: https://doi.org/10.1080/0022250X.1971.9989794. URL: https://doi.org/10.1080/0022250X.1971.9989794.

[43] Dorin, Alan. "Boolean Networks for the Generation of Rhythmic Structure". In: *Proceedings of ACMC 2000* July (2000), pp. 38–45.

[44] Derrida, B. and Flyvbjerg, H. "Multivalley structure in Kauffman's model: Analogy with spin glasses". In: *Journal of Physics A: Mathematical and General* 19.16 (1986), pp. 1003–1008. ISSN: 03054470. DOI: 10.1088/0305-4470/19/16/010.

[45] Klarner, Hannes, Streck, Adam, and Siebert, Heike. "PyBoolNet: A python package for the generation, analysis and visualization of boolean networks". In: *Bioinformatics* 33.5 (2017), pp. 770–772. ISSN: 14602059. DOI: 10.1093/bioinformatics/btw682.

[46] Oxford. *Complex*. URL: https://www.lexico.com/en/definition/complex.

[47] Li, M. and Vitányi, P. *An introduction to Kolmogorov complexity and its applications*. New York, USA: Springer-Verlag, 1997. ISBN: 0-387-94868-6.

[48] Hopcroft, J. *Introduction to Automata Theory, Languages and Computation*. USA: Pearson Education Company, 2001. ISBN: 0-201-44124-1.

[49] Minsky, M. L. *Computation: Finite and Infinite Machines*. USA: Prentice-Hall, 1967.

[50] Gersting, J. L. *Mathematical structures for computer science*. USA: W. H. Freeman and Company, 1998. ISBN: 0-7167-8306-1.

[51] Forouzan, B. A. *Foundations of Computer Science, From Data Manipulation to Theory of Computation*. Brooks Cole, 2003. ISBN: 0-534-37968-0.

[52] Turing, A. M. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[53] Weisstein, Eric W. "Universal Turing Machine." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/UniversalTuringMachine.html.

[54] Taylor, R. G. *Models of computation and formal languages*. USA: Oxford University Press, 1997. ISBN: 0-19-510983-X.

[55] Weisstein, Eric W. "Halting Problem." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HaltingProblem.html.

[56] Downey, R. G. and Hirschfeldt, D. R. *Algorithmic Randomness and Complexity*. New York, USA: Springer-Verlag New York, 2010. ISBN: 978-1-4939-3820-9.

[57] Zenil, Hector et al. "A decomposition method for global evaluation of shannon entropy and local estimations of algorithmic complexity". In: *Entropy* 20.8 (2018), pp. 1–46. ISSN: 10994300. DOI: 10.3390/e20080605. arXiv: arXiv:1609.00110v7.

[58] Soler-Toscano, Fernando et al. "Calculating Kolmogorov complexity from the output frequency distributions of small turing machines". In: *PLoS ONE* 9.5 (2014). ISSN: 19326203. DOI: 10.1371/journal.pone.0096223.

[59] Fenner, Stephen and Fortnow, Lance. "Compression Complexity". In: *arXiv e-prints*, arXiv:1702.04779 (Feb. 2017), arXiv:1702.04779. arXiv: 1702.04779 [cs.CC].

[60] Hankerson, D. R., Harris, G. A., and Jhonson, P. D. *Introduction to Information Theory and Data Compression*. USA: CRC Press LLC, 2003. ISBN: 1-58488-313-8.

[61] Oswal, S., Singh, A., and Kumari, K. "Deflate Compression Algorithm". In: *International Journal of Engineering Research and General Science* (2016).

[62] Toscano, F. S. and Zenil, H. "De los universos digitales a la mente. Nuevas herramientas para cuantificar nuestra intuición sobre la complejidad y el azar". In: *Investigación y Ciencia* 447 (2013), pp. 10–12.

[63] Delahaye, Jean-Paul. "Le défi des fabiles complexités". In: *Pour la science* 405 (2011), pp. 82–87.

[64] Nalbantoglu, Özkan, Russell, David, and Sayood, Khalid. "Data Compression Concepts and Algorithms and Their Applications to Bioinformatics". In: *Entropy* 12.1 (Dec. 2009), pp. 34–52. DOI: 10.3390/e12010034. URL: https://doi.org/10.3390/e12010034.

[65] Giancarlo, Raffaele, Scaturro, Davide, and Utro, Filippo. *Textual data compression in computational biology: A synopsis*. 2009. DOI: 10.1093/bioinformatics/btp117.

[66] Cilibrasi, Rudi and Vitányi, Paul M.B. "Clustering by compression". In: *IEEE Transactions on Information Theory* 51.4 (2005), pp. 1523–1545. ISSN: 00189448. DOI: 10.1109/TIT.2005.844059.

[67] Shannon, C. E. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423. ISSN: 15387305. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[68] Zemansky, M. W. and Dittman, R. H. *Heat and Thermodynamics*. USA: McGraw-Hill, 1997. ISBN: 0-07-017059-2.

[69] Morzy, Mikołaj, Kajdanowicz, Tomasz, and Kazienko, Przemysław. "On measuring the complexity of networks: Kolmogorov complexity versus entropy". In: *Complexity* 2017 (2017). ISSN: 10990526. DOI: 10.1155/2017/3250301.

[70] Weisstein, Eric W. "Entropy." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Entropy.html.

[71] Émile Borel. La mécanique statique et l'irreversibilité. J. Phys. Theor. Appl., 3(1), pp.189-196. 10.1051/jphystap:019130030018900. jpa-00241832.

[72] Weisstein, Eric W. "Busy Beaver." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BusyBeaver.html.

[73] Nater, Alexander et al. "Resolving evolutionary relationships in closely related species with whole-genome sequencing data". In: *Systematic Biology* 64.6 (2015), pp. 1000–1017. ISSN: 1076836X. DOI: 10.1093/sysbio/syv045.

[74] Gauvrit, Nicolas et al. "Algorithmic complexity for short binary strings applied to psychology: A primer". In: *Behavior Research Methods* 46.3 (2014), pp. 732–744. ISSN: 15543528. DOI: 10.3758/s13428-013-0416-0.

[75] Zenil, H. "Compression-Based Investigation of the Dynamical Properties of Cellular Automata and Other Systems". In: *Complex Systems* (2010).

[76] Zenil, Hector and Villarreal-Zapata, Elena. "Asymptotic behavior and ratios of complexity in cellular automata". In: *International Journal of Bifurcation and Chaos* 23.9 (2013), pp. 1–18. ISSN: 02181274. DOI: 10.1142/S0218127413501599.

[77] Faloutsos, Christos and Megalooikonomou, Vasileios. "On data mining, compression, and Kolmogorov complexity". In: *Data Mining and Knowledge Discovery* 15.1 (2007), pp. 3–20. ISSN: 13845810. DOI: 10.1007/s10618-006-0057-3.

[78] Schmidhuber, Jürgen. "Discovering neural nets with low Kolmogorov complexity and high generalization capability". In: *Neural Networks* 10.5 (1997), pp. 857–873. ISSN: 08936080. DOI: 10.1016/S0893-6080(96)00127-X.

[79] Brandouy, Olivier et al. "Algorithmic complexity of financial motions". In: *Research in International Business and Finance* 30 (2014), pp. 336–347. ISSN: 02755319. DOI: 10.1016/j.ribaf.2012.08.001. URL: http://dx.doi.org/10.1016/j.ribaf.2012.08.001.

[80] Zenil, Hector and Delahaye, Jean Paul. "An algorithmic information theoretic approach to the behaviour of financial markets". In: *Journal of Economic Surveys* 25.3 (2011), pp. 431–463. ISSN: 09500804. DOI: 10.1111/j.1467-6419.2010.00666.x. arXiv: arXiv:1008.1846v3.

[81] Zenil, Hector et al. "Two-dimensional Kolmogorov complexity and an empirical validation of the Coding theorem method by compressibility". In: *PeerJ* 2015.1 (2015), pp. 1–31. ISSN: 21678359. DOI: 10.7717/peerj-cs.23.

[82] Kulkarni, Amit and Bush, Stephen. "Detecting distributed denial-of-service attacks using Kolmogorov Complexity metrics". In: *Journal of Network and Systems Management* 14.1 (2006), pp. 69–80. ISSN: 10647570. DOI: 10.1007/s10922-005-9016-3.

[83] Li, Ming and Vitányi, Paul. *An Introduction to Kolmogorov Complexity and Its Applications*. New York, USA. ISBN 978-1-4899-8445-6. Springer-Verlag New York, 2014.

[84] Géron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. USA: O'Reilly, 2017. ISBN: 978-1-491-96229-9.

[85] Arthur, Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3 (1959), p. 210.

[86] Mitchell, T. M. *Machine Learning*. McGraw Hill, 1997.

[87] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2017. ISBN: 10 0-387-31073-8.

[88] Patterson, J. and Gibson, A. *Deep Learning*. USA: O'Reilly, 2017. ISBN: 978-1-491-91425-0.

[89] Buduma, N. and Lacascio, N. *Fundamentals of Deep Learning*. USA: O'Reilly, 2017. ISBN: 978-1-491-92561-4.

[90] Liu, Danqing. *A Practical Guide to ReLU*. 2017. URL: https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7.

[91] Walia, Anish Singh. *Activation functions and it's types-Which is better?* 2017. URL: https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f.

[92] Brownlee, Jason. *How to Choose Loss Functions When Training Deep Learning Neural Networks*. 2019. URL: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/.

[93] Brownlee, Jason. *Loss and Loss Functions for Training Deep Learning Neural Networks*. 2019. URL: https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/.

[94] Dozat, Timothy. "Incorporating Nesterov Momentum into Adam". In: *ICLR Workshop* 1 (2016), pp. 2013–2016.

[95] Grus, J. *Data Science from Scratch*. USA: O'Reilly, 2015. ISBN: 978-1-491-90142-7.

[96] Brownlee, Jason. *What is the Difference Between a Batch and an Epoch in a Neural Network?* 2018. URL: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/.

[97] Muller, A. C. and Guido, S. *Introduction to Machine Learning with Python*. USA: O'Reilly, 2017. ISBN: 978-1-449-36941-5.

[98] VanderPlas, J. *Python Data Science Handbook*. USA: O'Reilly, 2017. ISBN: 978-1-491-91205-8.