UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

GENERATING K-TREES

Autor: Alejandro Vara Mira

Director: Dr. Kolja Knauer
Realitzat a: Departament de Matemàtiques i Informàtica
Barcelona, 13 de juny de 2023

Abstract

The main goal of this thesis is to accomplish the generation of all unlabeled k-trees, which can be considered as a natural extension of ordinary trees.

In pursuit of this objective, three distinct codification schemes for k-trees have been explored, each of which comprising a colored tree. Through an in-depth examination of one of these schemes, it has been demonstrated that all k-trees can be encoded. Consequently, the focus of the research has shifted towards generating all non-redundant encodings.

The algorithms developed in this study, along with the necessary code to achieve the project's objective, have been implemented in SageMath.

2020 Mathematics Subject Classification. 05A15, 05C05, 05C75

Acknowledgments

I would like to express my deepest appreciation to my mentor, Dr. Kolja Knauer, for his invaluable guidance and support throughout the course of my bachelor's thesis. Kolja's extensive knowledge, prompt responses, and unwavering patience have been instrumental in shaping the outcome of this work.

Additionally, I would like to extend my heartfelt thanks to my parents for their unwavering support, love, and encouragement throughout this endeavor and the entire bachelor's degree. Their belief in my abilities has been a constant source of motivation.

Contents

List of Figures

Chapter 1

Introduction

Prior to introducing the thesis's motivation and structure, it is necessary to establish foundational concepts in graph theory and provide essential theoretical background on k-trees.

1.1 Graph theory basis

This section provides an introduction to graph theory, covering fundamental concepts such as the definition of a graph, its various characteristics and types, the notion of graph isomorphism, and the meaning of terms like path, distance and more. Also, propositions related to the center of a tree are shown. Most of the presented theory can be found in: "Graphs" by F. J. Soria de Diego [8], "Graph Theory" by R. Diestel [1], and "Introduction to graph theory" by D.B. West [11]. Readers already familiar with these concepts are encouraged to proceed to the next section, the only thing that might be different is how we define the concept of branches.

A **simple graph** or **graph** $G = (V, E)$ is a finite set of elements $V = V(G)$, called vertices or nodes, and a set of edges $E = E(G)$ joining two different vertices. Two vertices are neighbors or adjacent if they are joined by an edge (if this is the case, we say that the vertices are incident on the edge, and conversely). If $e \in E(G)$ is incident to the vertices $a, b \in V(G)$, we write $e = ab$. A **subgraph** of G is a graph whose sets of vertices and edges belong to G.

A **labeled** graph is a graph where each vertex is assigned with a non-negative integer label. These labels serve as unique identifiers, allowing for the distinction and identification of vertices within the graph.

A **complete** graph is a graph in which every pair of vertices are joined by an

edge. We denote a complete graph with n vertices K_n.

Definition 1.1. Two graphs G and H are **isomorphic** if there exists a bijection φ between their sets of vertices that preserves the edges:

$$\exists \varphi : V(G) \to V(H) \textit{ such that, } xy \in E(G) \textit{ if and only if, } \varphi(x)\varphi(y) \in E(H) :$$

The map φ is called an **isomorphism**. If $G = H$, it is called **automorphism**.

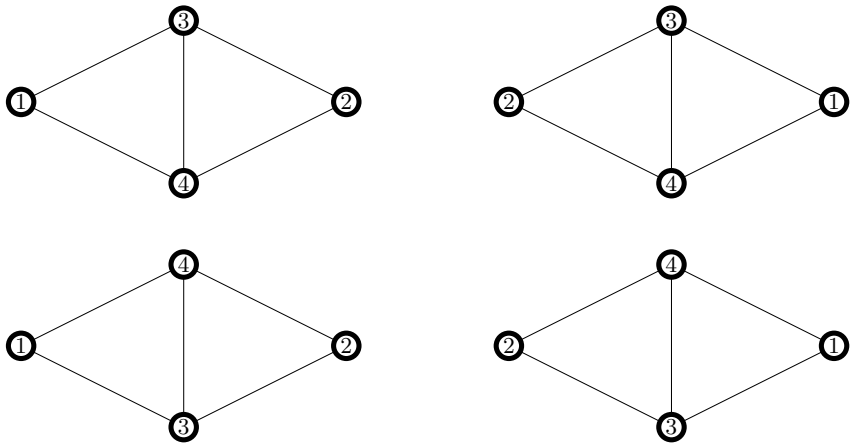


Figure 1.1: All isomorphic 2-trees on 4 vertices over the same set of nodes.

In Figure 1.1, it can be observed that by applying a permutation from Id, $(1,2)$, $(1,3)$ or $(1,2)(3,4)$ to one of the graphs, any of these isomorphic graphs can be obtained. This is because vertices 1 and 2 share the same adjacent vertices, as do vertices 3 and 4.

A **coloration** of a graph consists in the assignment of colors to its vertices. A proper coloration of a graph is achieved when two adjacent vertices do not share the same color.

A walk of length k between the vertices $v_1 \neq v_{k+1}$ of a given graph is a sequence of k edges of the form

$$v_1v_2, v_2v_3, ..., v_kv_{k+1}$$

In this case it is denoted as walk between v_1 and v_{k+1} (we also use the notation $v_1v_2...v_{k+1}$). A **path** is a walk for which all vertices are different. A graph is **connected** if there exists a path between every pair of vertices.

A closed walk of length k is a sequence of edges $\overline{v_1v_2}, \overline{v_2v_3}, ..., \overline{v_kv_1}$ (or also $v_1v_2...v_kv_1$) starting and finishing at the same vertex. A cycle C_k is a closed walk of length k having different all edges and intermediate vertices.

Let G be a connected graph and $u, v \in V$. We denote $d(u, v)$ as the **distance from u to v**, which is the length of the shortest path (called **geodesic**) joining these vertices . We call **diameter** of G the length of the longest geodesic of G, which is denoted as $diam(G)$. Given $v \in V$, the **eccentricity** of v, $\epsilon_G(v)$, is the maximum distance between v and any other vertex in V. The **center** of a connected graph G, $C(G)$, is the set of vertices of all vertices with minimum eccentricity.

A **tree**, $T = (V, E)$, is a connected graph with no cycles. In a tree, vertices that are adjacent to only another vertex are called leaves. A **rooted tree**, T_r, is a tree in which a vertex r is distinguished, this vertex is called the root of the tree. For any vertex v in a rooted tree, the **children** of v are the adjacent vertices of v that are further from the root than v. The **parent** of v, if v is not the root, is from the adjacent vertices of v, the one that is the nearest to the root.

The **depth** of a vertex in a rooted tree is the number of edges in the path from the root to that vertex. The **height** of a rooted tree is the maximum depth among all its vertices. A vertex u is considered an **ancestor** of another vertex v, if u belongs to the path from v to the root. A vertex v is a **descendent** of u, if u belongs to the path from v to the root. The **first generation** of a rooted tree, are the children of the root, or in other words, the neighbors of the root.

We define the **branches** of a rooted tree, as the set of new subtrees created when removing the root. The vertices from first generation will become the new roots of their respective subtrees. The set of branches with a height equal to the height of the tree minus one is referred to as the **longest branches**. A branch is identified by its root, we call the root its representative.

Proposition 1.2. *[C. Jordan, 1869 [5]] The center of a tree is either a vertex or a set of two adjacent vertices.*

Proof. We will prove the proposition by induction over n, the number of vertices in a tree T.

Base case: If n=1 or n=2, then the center is the entire tree.

Inductive step: Let n>2. Let T be a tree with n vertices. Assume the center of every tree with less than n vertices is a vertex or a set of two adjacent vertices. Form T′ by removing the leaves of T. Since n>2, T′ has at least a vertex.

Every vertex at maximum distance in T from a vertex $v \in V(T)$ is a leaf (otherwise, the path reaching it from v can be extended farther). Therefore, removing the leaves of T removes the vertices at maximum distance from a given interior vertex v, thereby reducing the eccentricity of v by at least 1. It does not, however,

remove the neighbors of those vertices, which are at distance $\epsilon_T(v) - 1$ from v, so it reduces the eccentricity only by 1.

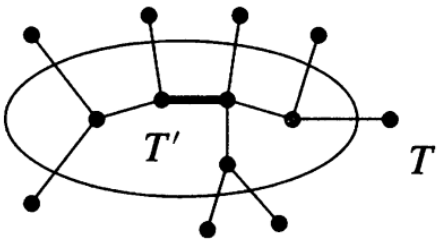


Figure 1.2: Removing the leaves from a tree T

So for any v in T′, we have $\epsilon_{T'}(v) = \epsilon_T(v) - 1$. Thus, $\epsilon_T(u) < \epsilon_T(v)$, if and only if, $\epsilon_{T'}(u) < \epsilon_{T'}(v)$ for any two vertices of T′. So the ordering of vertices by increasing eccentricity does not change when we go from T to T′. Hence the vertices of minimal eccentricity in T are the same as the vertices of minimal eccentricity in T′. This implies that the center of T′ is the same as the center of T. By induction hypothesis the center of T′ is a vertex or a set of two adjacent vertices.

□

Remark 1.3. The algorithm for determining the center of a tree involves iteratively removing the leaves of the tree until either one or two vertices remain.

Proposition 1.4. *The center of a tree is the center of its longest path.*

Proof. Given a tree T, its longest path P is between leaves, x and y, if not we could compute a longer one by adding the neighbors not in the path.

Since $C(T)$ is either a vertex or a set of two adjacent vertices, Proposition 1.2. If $C(T)$ is a vertex v, let us assume v is not in P. If $C(T)$ is a set of two adjacent vertices, let us assume one of them v is not in P.

Let d be the distance between v and the nearest vertex u from P. $\epsilon(u)$ is the distance between u and the furthest leaf x. If there is a further vertex t from u, then the path between t and y passing through u would be longer than P. Now, $\epsilon(v) > d + \epsilon(u)$, since there is only a path between v and x, otherwise we would have a cycle in the tree. Therefore, $\epsilon(v) > \epsilon(u)$ and hence v is not the center of the tree.

We have seen that the center belongs to the longest path. Additionally, for any vertex u in P, $\epsilon(u)$ is the distance between u and the furthest leaf x. Therefore, the center of the tree is the center of P.

□

Proposition 1.5. *Let T_r be a rooted tree in which the center is a single vertex r, the root. If $|V(T_r)| > 1$, then the cardinality of the set of longest branches must be greater than one.*

Proof. If a rooted tree T_r has more than one vertex and its center is one single vertex r, then the center has two or more neighbors. Therefore, T_r has more than one branch.

Now, let us assume the set of longest branches of T_r contains only one branch. From Proposition 1.4 we know that the longest path is between leaves and its center is the center of the tree. Hence, the longest path is between one of the deepest leaves from the longest branch and one from the second longest branch.

Since one is taller than the other, the height of the second longest branch must be the height of the longest one minus one, if it is smaller then the root would not be in the center. Therefore, the center of the longest path must be two vertices, since the longest path has an even number of vertices. Hence, the center must be two vertices, which is a contradiction.

So the he cardinality of the set of longest branches must be greater than one.

□

1.2 k-trees

A tree is usually defined as a connected graph with no cycles. The following inductive definition can also be used. The graph consisting on a single vertex is a tree, and any graph with n+1 vertices obtained by joining a new vertex to any vertex in a tree with n vertices is a tree.

This inductive definition suggests a generalization of the concept tree, k-trees, but first let us give a necessary definition. A **k-clique** is a complete subgraph of k vertices within a larger graph. Note, a $(k+1)$-clique contains $k+1$ k-cliques.

Definition 1.6. *The class of k-trees may be defined recursively. A graph G is a **k-tree** if either $G = K_k$ or G can be obtained from a k-tree G' by adding a vertex adjacent to a k-clique of G'. Thus a 1-tree is an ordinary tree.*

Let us take a closer look at the case of 2-trees to better understand what a k-tree is, in Figure 1.3 there is a graphical representation. The simplest 2-tree is a complete graph on 2 vertices, see (1), i.e., 2 vertices joined by an edge. If we desire to add a vertex to this 2-tree, this new vertex will join our two vertices, giving us a triangle (3-clique), see (2). This triangle has 3 different 2-cliques, each edge of the triangle.

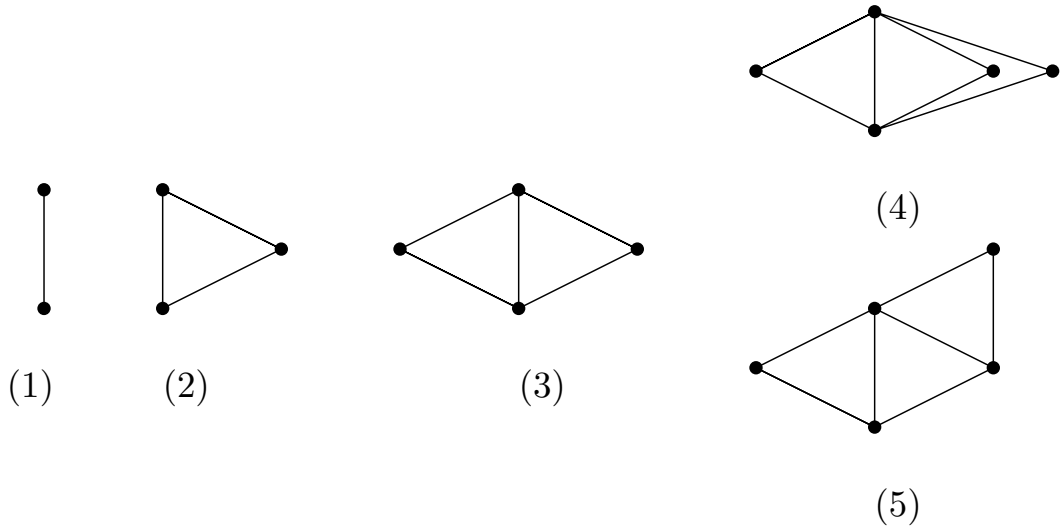


Figure 1.3: 2-tree from scratch

For adding a new vertex to the 2-tree, this time and all the following, we have to choose to which 2-clique we join the vertex, forming a new triangle. On this occasion, does not matter the 2-clique we choose, all results are isomorphic. Once added the new vertex, see (3), our 2-tree will consist of two triangles (3-cliques) joined by and edge (2-clique). Like before, if we wish to add a new vertex, we have to choose the 2-clique we want to join the vertex, and this time matters, because it can give 2-trees not isomorphic. If the vertex is joined to the 2-clique between the triangles, see (4), the 2-tree will be different than if the vertex is joined to any other 2-clique, see (5), given isomorphism. Notice that, except the case of joining the new vertex to the 2-clique between triangles, all other cases give the same 2-tree in terms of isomorphism.

To simplify notation, while working on a k-tree, a k-clique will be called **front** and a $(k+1)$-clique **hedron**. Is easy to notice that a k-tree is composed by hedrons joined by fronts. A k-tree with n hedrons has $n+k$ vertices. In the case of 2-trees, hedrons are triangles and fronts are edges.

Additionally, in a k-tree, a **simplicial vertex** refers to any vertex with degree k,

meaning it is adjacent to precisely k other vertices. Because of Definition 1.6, this k vertices form a front. In an ordinary tree, a simplicial vertex corresponds to a leaf vertex, which has a degree 1.

Remark 1.7. Any k-tree can be constructed recursively by starting from one of its fronts and joining simplicial vertices at each stage.

1.3 Motivations and Structure

The class of k-trees is a special type of graph that play a valuable role in mathematics, providing insights into graph structure and connectivity. They contribute to the development of efficient algorithms, optimization techniques, and graph decomposition methods, making them applicable to complex problems in network design, transportation systems, and supply chain management.

The main objective of this thesis is to generate all unlabeled k-trees. To achieve this goal, three distinct codification schemes for k-trees, each involving a tree with a coloration, have been explored. This work primarily focuses on one of these schemes.

The problem of counting labeled k-trees was solved approximately 50 years ago by J. W. Moon in 1969 [7]. The enumeration of unlabeled k-trees remained an unsolved problem until Gainer-Dewar resolved it in 2012 using theory of combinatorial species [3]. Subsequently, Gainer-Dewar and Ira M. Gessel presented an alternative approach in 2014 [4], resulting in a more concise description of the generating function for unlabeled k-trees. The asymptotic growth of the number of k-trees has been analyzed by Drmota and Jin using their results in [2].

In the upcoming Chapter 2, we will examine the encoding method proposed by Gainer-Dewar and Gessel for k-trees, and additionally introduce another codification scheme inspired by their work. However, our approach will eventually diverge and explore a distinct method. Specifically, we will adopt the codification scheme proposed by Kolja Knauer and Torsten Ueckerdt in 2023 [6].

Moving on to Chapter 3, we will delve into Knauer and Ueckerdt's codification scheme, providing a comprehensive explanation of the theory and algorithms involved. This will serve as a preamble for the next chapter.

In the final Chapter 4, our goal of generating all unlabeled k-trees will be achieved. By generating all different encodings, we enable the generation of all k-trees, considering their unlabeled nature. Furthermore, a specific function for computing all unlabeled k-trees has been crafted and implemented in Sagemath.

Chapter 2

Codification Schemes

In this chapter, we present and explain three distinct codification schemes for k-trees. Firstly, we explain the approach proposed by Gainer-Dewar and Gessel in [4], which is detailed in Section 2.1. Secondly, we introduce a novel codification scheme developed in this work. This scheme is presented in Section 2.2. Lastly, we explore the codification approach by Knauer and Ueckerdt in [6], which is discussed in Section 2.3.

2.1 First codification: Gainer-Dewar and Gessel

A k-tree can be thought as a made up of hedrons joined along fronts. We can color a k-tree by coloring the vertices with colors $1, 2, ..., k+1$ so that adjacent vertices are colored differently. Thus the $k+1$ vertices of any hedron are colored with all $k+1$ colors, and the k vertices of any front are colored in all but one of the colors. Each coloring of a k-tree is determined by its restriction to any one of its hedrons, see Figure 2.1.

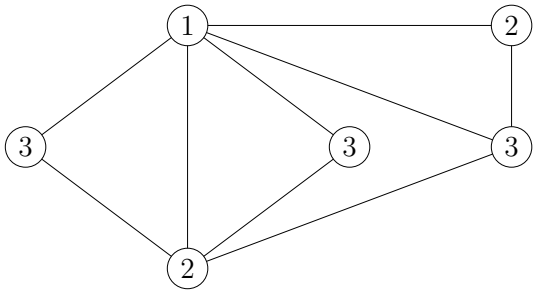


Figure 2.1: Unlabeled 2-tree colored

Consider the action of the symmetric group $S(k+1)$ on colored k-trees, achieved by permuting the colors. Through this action, *k*-trees can be characterized by their orbits within the set of colored k-trees. We will label the hedrons of *k*-trees. Specifically, a *colored hedron-labeled k-tree* consists of a colored k-tree accompanied by a bijection between its set of hedrons and a designated hedron-label set L. Notably, the only automorphism that preserves both the hedra and colors of a colored hedron-labeled k-tree is the identity automorphism, given that each vertex is uniquely determined by its color and the hedron it belongs to. This observation allows us to disregard vertex labels.

We will encode colored hedron-labeled *k*-trees by certain trees that we call *coding trees* (or *k-coding trees* if *k* needs to be specified). These trees possess two distinct types of vertices: black vertices and colored vertices. Edges connect black vertices with colored vertices, and each colored vertex is assigned one of the colors $1, 2, ..., k+1$, while remaining unlabeled in other aspects. The black vertices are labeled, and each black vertex has *k*+1 colored neighbors, one for each color, while a colored vertex may have any number of neighbors.

The process of constructing a coding tree from a colored k-tree involves coloring each front of the *k*-tree with the unique color not assigned to any of its vertices, like in Figure 2.2. This yields a coding tree with black vertices representing the hedrons of the k-tree (taking their labels) and colored vertices representing the fronts of the *k*-tree (taking their corresponding colors). A black vertex is adjacent to a colored vertex if and only if the corresponding hedron contains the corresponding front, see Figure 2.3.

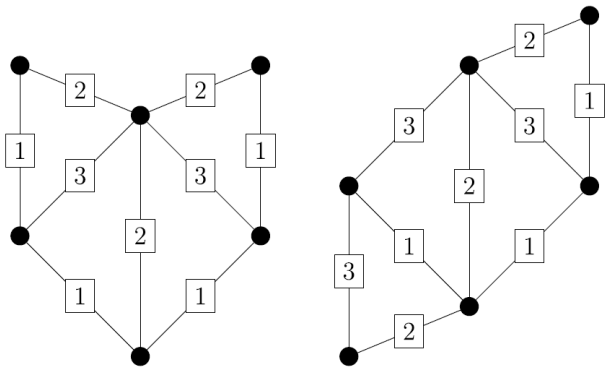


Figure 2.2: Two front-colored 2-trees (from [4])

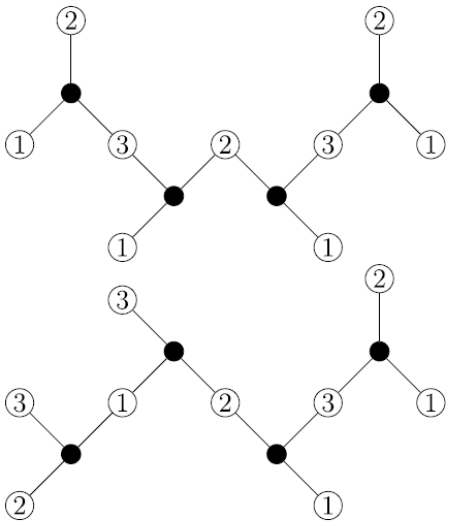


Figure 2.3: The corresponding coding trees (from [4])

It is worth noting that the encoding we employ establishes a bijection between colored hedron-labeled k-trees and k-coding trees. This claim can be substantiated with a proof akin to that found in Theorem 3.4 of [3]. So, by systematically generating and examining all distinct coding trees, we are able to generate the complete set of unlabeled k-trees.

Remark 2.1. From any coding tree, we may obtain a pruned coding tree by removing its leaves, which are all colored vertices. In a pruned k-coding tree, every leaf is black, and every black vertex has at most k+1 colored neighbors. The colors of these neighbors are distinct integers ranging from 1 to k+1.

Conversely, from a pruned k-coding tree, we can recover the original k-coding tree by adding a leaf of color i adjacent to every black vertex with no neighbor of color i, for i from 1 to k+1.

To simplify the process of generating all coding trees, alternative codification schemes have been explored. These schemes aim to provide a more accessible and practical approach for generating and representing coding trees. By investigating different codification methods, we can discover more efficient algorithms and techniques for constructing coding trees and ultimately generating the complete set of unlabeled k-trees. This exploration of alternative codification schemes enhances our understanding of the diverse representations and properties of k-trees.

2.2 Second codification: developed here

The concept of coloring the fronts of k-trees has given rise to a novel codification scheme that builds upon this idea.

Now, we will encode front-colored k-trees (previous comments about coloring serve as well here), like the examples from Figure 2.2, into *coding trees*. This time, a coding tree is a rooted tree T_r, where each vertex represents a hedron (taking its label, like before) and each edge the front connecting two hedrons (taking its corresponding color). For constructing a coding tree, a hedron h_r must be chosen in order for it to represent the root r of the coding tree, more about this later.

Once h_r is chosen, each hedron with a front in common with h_r will be represented as a child of the root in T_r. Analogously, each hedron h represented in the first generation of T_r may share additional fronts with other hedrons, the representation of those hedrons will be children of the representation of h in the coding tree. By repeating this process until no more hedrons are left, we obtain a coding tree with all hedrons represented. Consequently, we can codify any given k-tree using this approach.

Indeed, this process of constructing a coding tree aligns with the recursive definition provided in Definition 1.6 for k-trees.

Note that, the distinction of the root is necessary for giving a hierarchy of hedrons. Because more than two hedrons can share a front.

Choosing any hedron as h_r will give us a proper coding tree, i.e., one from which we could obtain the former k-tree. The problem is, changing h_r will give us different coding trees. Since, we have not being able to find a general and unambiguously way to choose h_r, we changed our focus to Knauer and Ueckerdt's codification scheme.

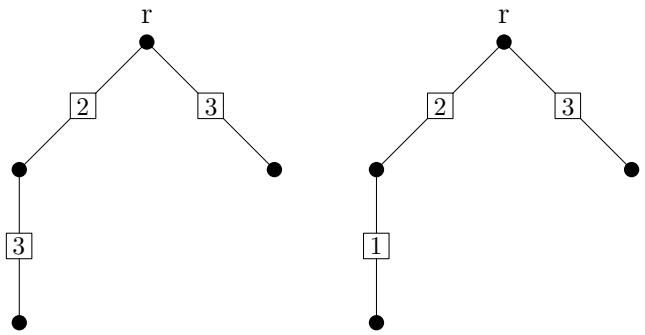


Figure 2.4: The second codification of the k-trees from Figure 2.2

In Figure 2.4, the codification of the two front-colored 2-trees from Figure 2.2 is depicted. Considering h_r from the middle hedrons, the one in the right, for both 2-trees.

2.3 Third codification: Knauer and Ueckerdt

Knauer and Ueckerdt's codification scheme solves the problem from a our previous codification. That is, finding an unambiguously root in any k-tree.

Remark 1.7 states that any k-tree, G, can be constructed recursively by starting from one of its fronts and joining simplicial vertices at each stage. Conversely, we can apply the opposite process known as the **elimination scheme**. Where at each stage, all simplicial vertices are removed from G, until only a complete graph remains.

The remaining complete graph will be represented as the root of this third codification. The reason we do not follow the same approach in the last codification is because the vertices of the coding trees in Section 2.2 represent the hedrons of the k-tree. However, in the elimination scheme, the remaining complete graph can either be a front or a hedron (see Figure 2.5), making it unsuitable to serve as the root.

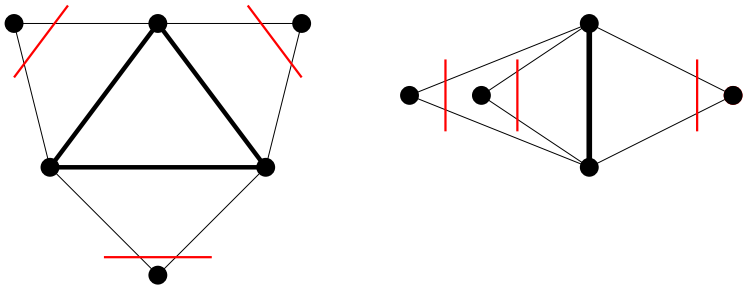


Figure 2.5: Removing last simplicial vertices from two different 2-trees

Having said this, from this point onward, we define a **coding tree** (or **k-coding tree** if k needs to be specified) as a triple (T_r, c, k). Here, T_r represents a rooted tree in which $C(T) = r$, c is a map $c : V(T_r) \setminus \{r\} \to \{1, 2, ..., k+1\}$ that assigns colors to the vertices of the tree, and k is an integer denoting the value of k for the coding tree.

Given a labeled k-tree, G, we can assign a proper coloration to G using the colors $1, 2, \ldots, k+1$. Once G is colored, we can encode it into its coding tree (T_r, c, k), which is constructed so that:

- The root r of T_r represents the remaining complete graph resulting from the elimination scheme. We denote this complete graph as H_G, which can be either a hedron or a front.
- Each vertex $v_k \in V(G) \setminus V(H_G)$ is represented as a vertex v in T_r.
- For each vertex v_k joined to a front of H_G, its representation v is a child of r in T_r. Note that, v_k forms a hedron with the front to which it is joined. Now, if a vertex u_k is joined to a front from the hedron v_k forms, the representation of u_k in T_r, u, is a child of v. Analogously for all vertices from G.
- For each $v_k \in V(G) \setminus V(H_G)$, its representation v will take the color assigned to v_k. More precisely, $c(v)$ is the color of v_k in G.

Throughout this work, we will consistently use the notation introduced here. To maintain clarity, for any vertex $v_k \in V(G) \setminus V(H_G)$, its corresponding representation in T_r is denoted as v, and vice versa. In addition, H_G is the remaining graph when applying the elimination scheme over G.

Remark 2.2. Any coloration of G is a good choice. Since as we will see in the upcoming chapter, it will result in an equivalent coding tree. That is, two **coding trees are equivalent**, if when decoding them, both return isomorphic k-trees.

Notice that, any coloration of G is the result from a permutation of colors, $\sigma : \{1, 2, ..., k+1\} \to \{1, 2, ..., k+1\}$, of our current coloration. This is because, as previously discussed, any coloration of a k-tree is determined by its restriction to anyone of its hedrons. Therefore, any alternative coloration of G will yield the same coding tree, but with the colors permuted according to the same permutation σ.

It is useful to think that r is colored with the set of colors $V(H_G)$ has assigned in G. That is, either the complete set of colors or a set of k different colors, depending on whether H_G is a hedron or a front, respectively.

This codification scheme, as compared to the later one, is more suitable as it allows us to think of the vertices from $V(T_r) \setminus \{r\}$ as simplicial vertices, joined to the fronts formed by their parents. This is because, when joining a simplicial vertex to a front, it forms a hedron with k new fronts. An example of this codification could be found in Figure 3.1

Chapter 3

Coding trees

This chapter delves into Knauer and Ueckerdt's codification scheme. Specifically:

Section 3.1 shows how we can encode any k-tree into a coding tree, by means of an algorithm.

Section 3.2 explores the process of decoding a coding tree to obtain a k-tree.

Section 3.3 examines the coherence between the algorithms presented in the previous sections. It establishes the relationship and consistency between the encoding and decoding algorithms.

Throughout this work, all algorithms presented will be implemented in SageMath [9], a powerful open-source mathematics software system. Therefore, you may come across some comments or references to SageMath in the explanations provided.

3.1 Encoding

The encoding algorithm presented in this section is based in following the elimination scheme, it will be called *coding_ktree* or simply Algorithm 1. Figure 3.1 provides a visual representation of the algorithm, which can help in understanding it.

The algorithm *coding_ktree* takes as input a labeled k-tree G and the integer k, although finding k given a k-tree is an easy task. If G is not a complete graph, k is the smallest degree of among all vertices. If G is a complete graph, the coding tree is a single vertex, the root.

We initialize the graph of our coding tree as an empty graph, T_r, and create a copy of G, H_G, where we will make transformations while we make comparisons in the original k-tree G. An empty list L is also initialized. The algorithm proceeds as follows:

Algorithm 1 coding_ktree

1. Remove all simplicial vertices from H_G, and save them in L.
2. Remove all simplicial vertices from H_G. For each removed vertex v_k, identify the set of vertices in L that are adjacent to v_k in G. Then, we join all vertices from this set to v_k in T_r, and v_k is added to L, while the vertices from the set are removed from L. If the set is empty, then the vertex is added to L directly.
3. Repeat (2) until H_G is either a front or a hedron.
4. All vertices in L are joined to r in T_r, where the label r is not in $V(G) \setminus V(H_G)$.
5. Give a coloration of G.
6. For each vertex $v_k \in V(T_r) \setminus \{r\}$, the mapping c is defined by $c(v_k)$ equals the color of v_k in G.
7. The algorithm returns (T_r, c, k)

Note that, in this case $v_k = v$, since no labels have been changed. Now, the algorithm allows us to infer the effect of each step on the graphs involved.

Step (2) establishes a parent-child relationship between vertices v_k and u_k in T_r. This relationship exists in T_r if, in G, the vertex u_k is connected to a front that is formed by the vertex v_k. More formally, when applying the elimination scheme in G, the fronts a vertex in $v_k \in V(G) \setminus V(H_G)$ forms are those removed when removing the vertex v_k, at the stage in which v_k is a simplicial vertex.

Moving on to step (4), all vertices in G that are adjacent to a front in the remaining graph H_G become children of the root vertex r in T_r. In other words, they are the elements of the first generation of T_r.

In step (5), the task of coloring a graph is usually challenging. However, when it comes to k-trees, the process becomes relatively simple. One approach is to begin by identifying a hedron and assigning a unique color to each of its vertices, ranging from 1 to $k+1$. Then, for each vertex connected to a front of the hedron, assign it the color that is not present in the front. For instance, one can start the coloring process from the hedron formed by a simplicial vertex.

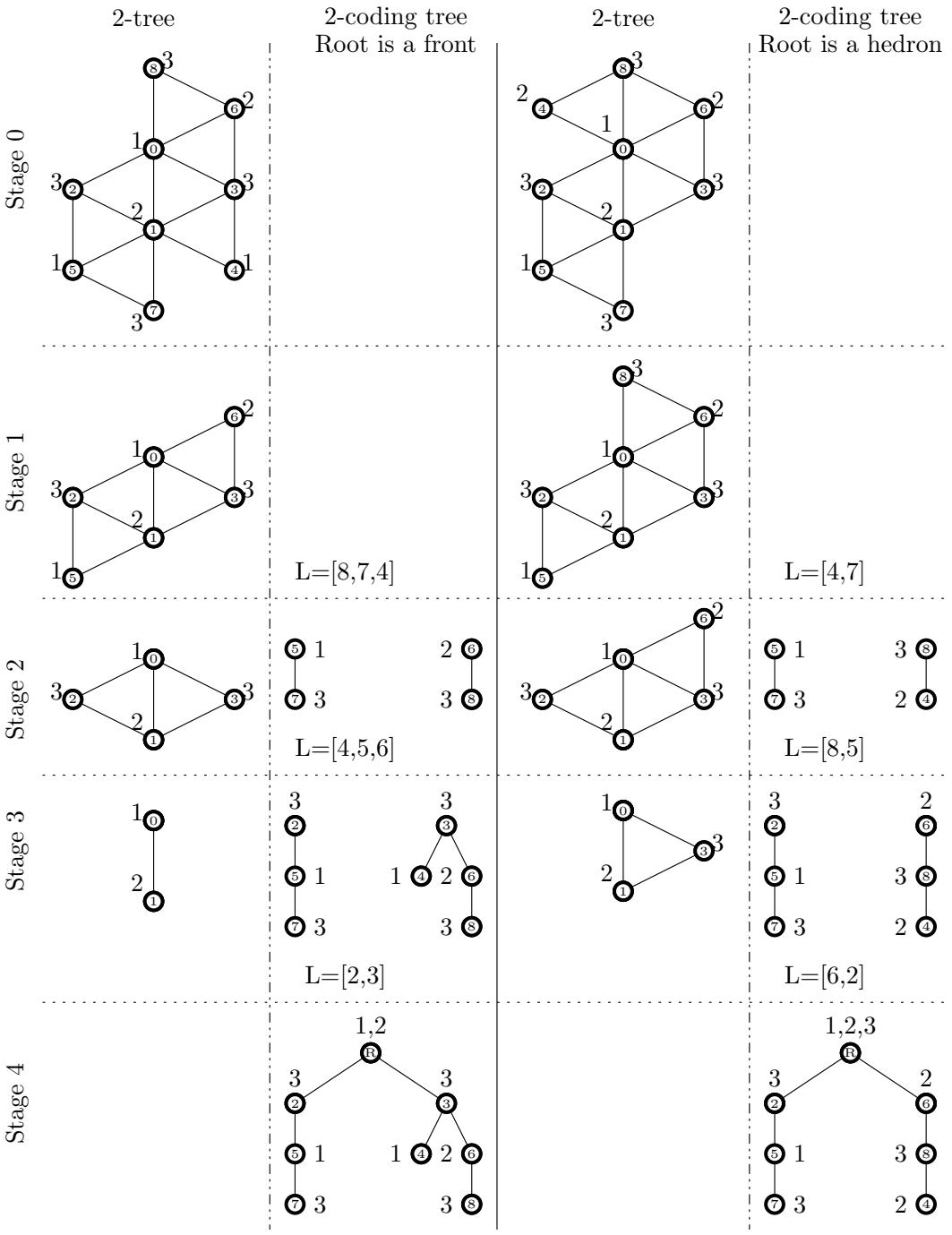


Figure 3.1: Encoding two different 2-trees

Due to the straightforward nature of coloring G, when implementing the algorithm, the coloring is computed through the *first_coloring* method from the

graph_coloring library in SageMath [9].

Once the algorithm is run, it is possible to know if the root of the coding tree represents a front or a hedron. By comparing the number of vertices between T_r and G. Let $n_T = |V(T_r)|$ and $n_G = |V(G)|$, if $n_T = n_G - k + 1$ the root represents a front, but if $n_T = n_G - k$ the root represents a hedron. Another way is by seeing if all vertices from the first generation have the same color or not, meaning the root is a front or a hedron, respectively.

Proposition 3.1. *The algorithm coding_ktrees returns a coding tree (T_r, c, k), when the input is a labeled k-tree G.*

Proof. First, let us see that T_r is a rooted tree in which $C(T_r) = r$.

In step (2), for each simplicial vertex removed, we join it to a set of vertices from L in T_r. Then we remove the set from L and add the simplicial vertex to L, ensuring that no cycles are created. In (4) we join all lasting vertices in L to the root, therefore T_r is a graph with no cycles.

In addition, it is worth noting that before beginning step (4), each vertex in T_r is descendent or the same from one of the vertices from L. Moreover, this property holds true at the start and completion of step (2) as well. Then, in step (4), all vertices from L are joined to the root, so T_r is connected.

Thus, T_r is a rooted tree, since it is a connected graph with no cycles. Additionally, $C(T_r) = r$ because the elimination scheme that we follow in G is similar to Jordan's algorithm for computing the center in a 1-tree, as stated in Remark 1.3, removing all leaves at each stage.

Also, the coloration c is a proper one. Notice that if two vertices in T_r are adjacent they are also adjacent in the G.

Therefore, (T_r, c, k) is a coding tree.

□

3.2 Decoding

For decoding a coding tree (T_r, c, k), two different algorithms have been implemented. One in the case r represents a hedron, *decoding_root_hedron* or Algorithm 2; and another for the case r represents a front, *decoding_root_front* or Algorithm 3. Both algorithms take as input from the coding tree: k, c and *children*. Here,

children : $V(T_r) \to \mathcal{P}(V(T_r))$ is a mapping that associates each vertex from T_r with the set of its children.

Before executing the decoding algorithms, it is necessary to ensure that the label of the root vertex must be 0. If this is not the case, a relabeling is required

Both algorithms traverse the coding tree by following the Depth-First Search (DFS) strategy. That is, DFS begins at the root vertex and visits its first child (the order is not important). From there, it recursively visits the first child of each subsequent vertex until it reaches a leaf node. Then it backtracks to the parent vertex and continues to explore the next unvisited child, repeating the process until all vertices have been visited.

Let us see what Algorithm 2 does.

Algorithm 2 decoding_root_hedron

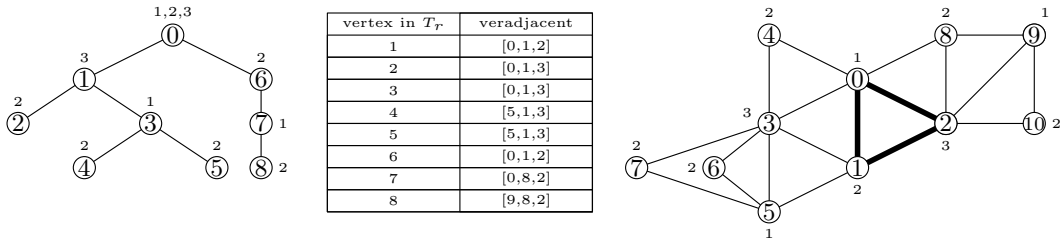
Input: k, coloration, children
$G \leftarrow CompleteGraph(k+1)$
$veradjacent \leftarrow [0, 1, ..., k]$
Function: $add\_hedrons(veradjacent, ver)$:
    if *ver* is a leaf **then**:
        $ver_k = ver + k$
        $c = coloration[ver]$
        For $i \in \{0, \ldots, k\} \setminus \{c-1\}$ **do**:
            Add the edge $(ver_k, veradjacent[i])$ to G
        return
    For v in $children[ver]$ **do**:
        $v_k = v + k$
        $c = coloration[v]$
        For $i \in \{0, \ldots, k\} \setminus \{c-1\}$ **do**:
            Add the edge $(v_k, veradjacent[i])$ to G
        $aux = veradjacent[c-1]$
        $veradjacent[c-1] = v_k$
        $add\_hedrons(veradjacent, v)$
        $veradjacent[c-1] = aux$
end
$add\_hedrons(veradjacent, 0)$
return G

First, the k-tree G is initialized as a complete graph on $k+1$ vertices, corresponding to the root in T_r. The complete graph is labeled from 0 to k, we will

take advantage of this fact. Also, the correspondence between the label of any $v \in V(T_r) \setminus \{r\}$ to a label v_k from G is given by $v_k = v + k$. To ensure that all vertices in G have different labels, we specifically requested that the root of the coding tree to be assigned with the label 0. This choice guarantees that there will be no vertex $v \in V(T_r) \setminus \{r\}$ that would correspond to $v_k = 0 + k$ in G.

Next, the list *veradjacent* is initialized as $[0, 1, ..., k]$. The list *veradjacent* will change at each stage so that when visiting a vertex $v \in T_r$, for any $i \in \{0, \ldots, k\}$, $veradjacent[i] = u_k$ where u is the nearest ancestor of v in T_r such that it has color $i + 1$. That is, when visiting a vertex $v \in T_r$, the list *veradjacent* (except $veradjacent[c(v) - 1]$) contains the elements of the front to which the corresponding v_k will be joined forming a hedron in G. The reason *veradjacent* is initialized as $[0, 1, ..., k]$ is because when initializing the G as a complete graph, we are assuming each vertex v_k of it has color $v_k + 1$.

In Figure 3.2, the decoding of a 2-coding tree is illustrated. The labels of this 2-coding tree have been set following the order the DFS strategy visits each vertex, for a better understanding. Also, a table is attached so one can see how the list *veradjacent* changes in each stage, when visiting a vertex in T_r.



vertex in T_r	veradjacent
1	[0,1,2]
2	[0,1,3]
3	[0,1,3]
4	[5,1,3]
5	[5,1,3]
6	[0,1,2]
7	[0,8,2]
8	[9,8,2]

Figure 3.2: Decoding a 2-coding tree

In the case of executing the *decoding_root_front* algorithm or Algorithm 3, an additional requirement is that $c(children(r))$ should be $k + 1$, we will take advantage of this fact. If this condition is not met, a permutation of the colors must be performed accordingly. In Theorem 3.5, we will see that this permutation yields an equivalent coding tree.

The Algorithm 3 operates in a nearly identical manner to Algorithm 2. Using the same terminology as previously mentioned, the correspondence is now expressed as $v_k = v + k - 1$. But, in this case, the k-tree G is initialized as a complete graph consisting of k vertices (a front). The local function $add\_hedrons(veradjacent, ver)$ remains unchanged, except the correspondence of labels. However, the invocation of $add\_hedrons(veradjacent, 0)$ is now substituted with the following:

Algorithm 3 decoding_root_front

.
.
.

For v in $children[0]$ **do**:
    $v_k = v + k - 1$
    $veradjacent = [0, ..., k-1]$
    For e in $veradjacent$ **do**:
        Add the edge (v_k, e) to G_f
    Append v_k to $veradjacent$
    $add\_hedrons(veradjacent, v)$

Before, we started from the root by calling $add\_hedrons(veradjacent, 0)$, since the root represented a hedron. Now we can think of the root and one of its child as a hedron and do the same thing, for all the children of the root. This is the reason why for any coloration of a coding tree in which the root represents a front, the first generation must have color $k+1$.

Proposition 3.2. *The algorithms decoding_root_hedron and decoding_root_front return a k-tree, when taking as input a coding tree.*

Proof. Initializing a k-tree as a complete graph, these algorithms are supposed to join a vertex to another k vertices that form a front in the k-tree. This vertex becomes a simplicial vertex and forms a hedron with the k vertices. So Definition 1.6 of k-tree is met.

We need to prove that for each vertex that we join to another k vertices, those k vertices form a front. This is going to be done by induction over the number of vertices in the coding tree. Since both algorithms join vertices by means of the list $veradjacent$, our inductive hypothesis is:

Let G be the graph constructed when running the decoding algorithms over the coding tree (T_r, c, k) with n_T vertices. For any $v \in V(T_r) \setminus \{r\}$, the correspondence of the first k ancestors of v such that all have different colors than v and among them, form a front in G (with the correspondence of v, a hedron in G).

Base case: If the coding tree has only one vertex, that is the root. A complete graph is returned and therefore a k-tree.

Inductive step: Adding a leaf u to the coding tree of our inductive hypothesis, being u joined to a vertex v, we have a coding tree with $n_T + 1$ vertices.

Because of the inductive hypothesis, the correspondence of v forms a hedron with the correspondence of the first k ancestors with different colors than v and among them, notice we only take into account the ancestors when joining a simplicial vertex to the k-tree (DFS). Choosing from this hedron the k vertices with different color than u give us a front. And these k vertices are the correspondence of the first k ancestors with different colors than u and among them, what we wanted to prove.

Both algorithms choose these k vertices, so we can ensure that they return a k-tree. □

3.3 Coherence

Theorem 3.3 establishes the coherence between the encoding and decoding processes by stating that the result obtained from decoding the coding tree of a k-tree is an isomorphic k-tree. This theorem assures that the essential structure and properties of the original k-tree are preserved throughout the encoding and decoding procedures.

Theorem 3.3. *Given a k-tree, the result from decoding its coding tree is an isomorphic k-tree.*

Proof. Let G be a k-tree and (T_r, c, k) the coding tree resulting from Algorithm 1. Let G' be the k-tree constructed by decoding (T_r, c, k). Note that, exists a correspondence between a vertex $v_k \in V(G) \setminus V(H_G)$ and a vertex of $v'_k \in V(G') \setminus V(H_{G'})$. In the case H_G is a hedron, Algorithm 2 is run over the coding tree and therefore the correspondence is $v'_k = v_k + k$. In the case H_G is a front, Algorithm 3 is run over the coding tree and therefore the correspondence is $v'_k = v_k + k - 1$. Let us assume there is no $v_k \in V(G) \setminus V(H_G)$ with label 0, without losing generality, in order to keep the mentioned correspondence between all vertices.

When decoding (T_r, c, k), in the stage in which we visit a vertex $v \in V(T_r) \setminus \{r\}$. The correspondence in G' of v, v'_k, is joined to the correspondence in G' of the first k ancestors of v with different colors than v and among them, which we have seen form a front. Therefore, v'_k is now a simplicial vertex of G' and G' has k new fronts, the fronts formed by v'_k. Also, these new k fronts are the only ones containing v'_k.

Now, when visiting the first child of v in T_r, u, the same process is done. Notice that v is the first ancestor of u such that $c(u) \neq c(v)$, since they are adjacent in T_r. Hence, u'_k is joined to one of the new k fronts v'_k formed, making v'_k not a simplicial vertex anymore.

Notice this is the exact opposite process we follow when encoding. That is, a vertex v_k is not a simplicial vertex until, at least, we remove the simplicial vertex u_k.

In the case v has no children, v is leaf of T_r. When finishing the decoding process, v'_k is a simplcial vertex in G', like v_k in G.

In conclusion, there is an isomorphism between $V(G) \setminus V(H_G)$ and $V(G') \setminus V(H_{G'})$, defined by the correspondence given in the first paragraph of this proof. Note that, the subgraphs H_G $H_{G'}$ are isomorphic, they are complete graphs with the same number of vertices. Hence, G and G' are isomorphic.

□

Proposition 3.4. *Given a coding tree, the result from coding its k-tree is an equivalent coding tree.*

Proof. Given a coding tree (T_r, c, k) and its k-tree G, the outcome from decoding (T_r, c, k). Coding G results in a second coding tree $(T'_{r'}, c', k)$. If we decode $(T'_{r'}, c', k)$ will result in a k-tree G' isomorphic to G, because of Proposition 3.3. Therefore, (T_r, c, k) and $(T'_{r'}, c', k)$ are equivalent. □

Theorem 3.5 provides a criterion for determining the equivalence of two coding trees. This result provides a method for identifying equivalent coding trees based on a combination of isomorphisms and color permutations.

Theorem 3.5. *Two coding trees (T_r, c, k) and $(T'_{r'}, c', k)$ are equivalent if there is an isomorphism $\varphi : V(T_r) \to V(T'_{r'})$ such that $\varphi(r) = r'$ and a permutation $\sigma : \{1, 2, ..., k+1\} \to \{1, 2, ..., k+1\}$ of the colors such that $c(v) = \sigma(c'(\varphi(v)))$ for all $v \in V(T_r)$.*

Proof. This theorem can be proved separately:

Isomorphism between rooted trees: Given two coding trees (T_r, c, k) and $(T'_{r'}, c', k)$ such that there is an isomorphism $\varphi : V(T_r) \to V(T'_{r'})$ and $\varphi(r) = r'$ and $c(v) = c'(\varphi(v))$ for all $v \in V(T_r)$. Let G be the k-tree resulting from decoding (T_r, c, k) and let G' be the k-tree resulting from decoding $(T'_{r'}, c', k)$.

Using the usual notation, let v_k and u_k be two adjacent vertices from $V(G) \setminus V(H_G)$. Since they are adjacent in G, let us assume v is the first ancestor of u in T_r with color $c(v)$ such that $c(v) \neq c(u)$, without losing generality. Consequently, $\varphi(v)$ is the first ancestor of $\varphi(u)$ in $T'_{r'}$ with color $c'(\varphi(v))$ such that $c(v) = c'(\varphi(v)) \neq c'(\varphi(u)) = c(u)$. Therefore, $\varphi(v)_k = \varphi(v_k)$ and $\varphi(u)_k = \varphi(u_k)$

are adjacent in G', since when decoding the same correspondence between the labels of the coding tree and their k-tree is given.

Hence, G and G' are isomorphic. Furthermore, is easy to see that the isomorphism $\phi : G \to G'$ is defined as:

$$\phi(v_k) = \begin{cases} \varphi(v_k), & \text{if } v_k \in V(G) \setminus V(H_G) \\ Id(v_k), & \text{if } v_k \in V(H_G) \end{cases}$$

Permutation of colors: Given two coding trees (T_r, c, k) and (T_r, c', k) such that there is a permutation $\sigma : \{1, 2, ..., k+1\} \to \{1, 2, ..., k+1\}$ of the colors such that $c(v) = \sigma(c'(v))$ for all $v \in V(T_r)$. Let G be the k-tree resulting from decoding (T_r, c, k) and let G' be the k-tree resulting from decoding (T_r, c', k).

Using the usual notation, let v_k and u_k be two adjacent vertices from $V(G) \setminus V(H_G)$. Let us assume v is the first ancestor of u in (T_r, c, k) with color $c(v)$ such that $c(v) \neq c(u)$, without losing generality. Consequently, v is the first ancestor of u in (T_r, c', k) with color $c'(v)$ such that $\sigma^{-1}(c(v)) = c'(v) \neq c'(u) = \sigma^{-1}(c(u))$. Because $\sigma^{-1}(c(v)) \neq \sigma^{-1}(c(u))$ since $c(v) \neq c(u)$, and σ^{-1} is a permutation. Therefore, v'_k and u'_k are adjacent in G', since when decoding the same correspondence between the labels of the coding tree and their k-tree is given.

Hence, G and G' are isomorphic. Furthermore, is easy to see that the isomorphism $\phi : G \to G'$ is defined as:

$$\phi(v_k) = \begin{cases} Id(v_k), & \text{if } v_k \in V(G) \setminus V(H_G) \\ \sigma(v_k + 1) - 1, & \text{if } v_k \in V(H_G) \end{cases}$$

Note that, a permutation of the colors, will only change to which front of the initial complete graph we join the correspondence of every child of the root.

In conclusion, this proves the equivalence stated in the theorem. We have also seen that the isomorphism between the two k-trees is given by the isomorphism φ and the permutation σ between their respective coding trees. □

Next, we present Proposition 3.6, which demonstrates that for any two isomorphic k-trees, their corresponding coding trees are equivalent. This result emphasizes that the specific labels assigned to the vertices of the k-tree are not crucial when encoding it into a coding tree.

Proposition 3.6. *Given two isomorphic k-trees, their coding trees are equivalent.*

Proof. The construction of the rooted tree resulting from encoding a labeled k-tree, is based solely in removing simplicial vertices and checking adjacencies. These two properties remain the same between isomorphic graphs. Therefore, when encoding two isomorphic k-trees, it will result in two isomorphic rooted trees. Moreover, the isomorphism between the rooted trees is defined as the restriction over all vertices except the root of the isomorphism between k-trees. Also, the isomorphism between the rooted trees sends one root to the other.

Now, given two isomorphic k-trees and providing a coloration of the first k-tree. Any coloration of the second is given by the isomorphism between them and a subsequent permutation of the colors. Therefore, when coding the second k-tree, it will result in an isomorphic rooted tree (seen in the first paragraph) and its coloration will be changed respecting the isomorphism and the earlier permutation. Because of Theorem 3.5 we know both coding trees are equivalent. □

Proposition 3.4 states that given a coding tree, the result from coding its k-tree is an equivalent coding tree. However, it does not guarantee that the two coding trees are isomorphic, meaning their underlying rooted trees may not be isomorphic in general. This distinction is illustrated in Figure 3.3.

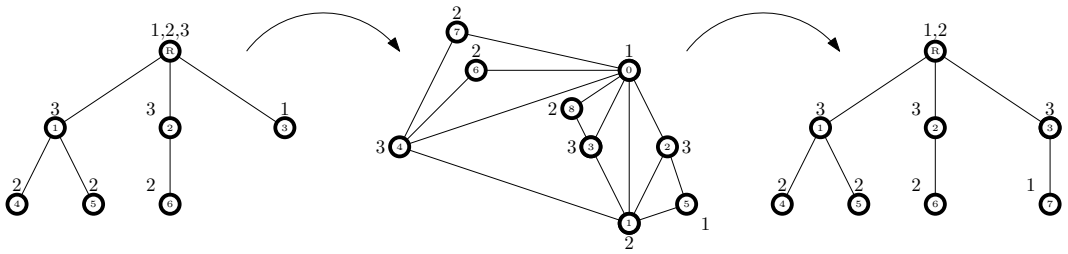


Figure 3.3: Decoding + coding

In Figure 3.3, the root of the first coding tree (T_r, c, k) represents a hedron, when decoding results in a k-tree G. But the outcome of encoding the k-tree is a coding tree $(T'_{r'}, c', k)$ in which the root represents a front. This means that the elimination scheme of G derives in the root being a front.

In (T_r, c, k), by making the root represent a front. If we add a vertex between the root and 3 with color 3, it is easy to see that this results in the same k-tree G when decoding it. Why is this the natural way of representing it when following the elimination scheme?

Proposition 3.7. *Given a coding tree (T_r, c, k), let G be the k-tree obtained from decoding (T_r, c, k), let $(T'_{r'}, c', k)$ be the resulting coding tree obtained from encoding G.*

- *If r represents a front in (T_r, c, k), then (T_r, c, k) and $(T'_{r'}, c', k)$ are isomorphic.*
- *If r represents a hedron in (T_r, c, k), then (T_r, c, k) and $(T'_{r'}, c', k)$ are isomorphic if and only if $c(v)$ is not the same for all vertices v representatives of the longest branches.*

Proof. Let us call G_1 the subgraph of G resulting from removing the simplicial vertices from G. In general, we call G_i the subgraph of G resulting from removing the simplicial vertices of G_{i-1}.

Let us also call T_{r_1} the subgraph of T_r resulting from removing the leaves from T_r In general, we call T_{r_i} the subgraph of G resulting from removing the simplicial vertices of $T_{r_{i-1}}$. Note that, $C(T_{r_i}) = r$ for all i, like we saw in the proof of Jordan's proposition, 1.2.

It is clear that the simplicial vertices of G are the correspondence of the leaves in T_r. Also, the simplicial vertices of G_1 are the correspondence of the parents of the leaves in T_r, except the case the parent is r. In other words, the simplicial vertices of G_1 are the correspondence of the leaves in T_{r_1}. Moreover, G_1 is the resulting k-tree from decoding (T_{r_1}, c, k). This property is true for any $i \in \{1, ..., l-1\}$, where l is the smallest integer such that $G_l = H_G$ is a complete graph.

Note that, $T_{r_{l-1}}$ is composed by the root and the representatives of the longest branches of G, which are two or more vertices, as stated in Proposition 1.5. Since the longest branches in G are the tallest.

We distinct two cases:

- The case r represents a front. In this case, for all $v \in V(T_{r_{l-1}}) \setminus \{r\}$, we have that $c(v) = k + 1$ as this is a requirement for decoding. The result from decoding $(T_{r_{l-1}}, c, k)$ is G_{l-1}. Notice all simplicial vertices of G_{l-1} are joined to the same front. Therefore, G_l is the front that is represented as the root r. Thus, r represents the remaining complete graph when following the elimination scheme over G, H_G. Hence, (T_r, c, k) and $(T'_{r'}, c', k)$ are isomorphic.
- On the other hand, the case$r$ represents a hedron:
  - If not all $v \in V(T_{r_{l-1}}) \setminus \{r\}$ have the same color. The result from decoding $(T_{r_{l-1}}, c, k)$ is G_{l-1}, where not all simplicial vertices in G_{l-1} are joined to the same front. Therefore, G_l is the hedron that is represented as the

root r. Thus, r represents the remaining complete graph when following the elimination scheme over G, H_G. Hence, (T_r, c, k) and $(T'_{r'}, c', k)$ are isomorphic.

– But if all $v \in V(T_{r_{l-1}}) \setminus \{r\}$ have the same color. The result from decoding $(T_{r_{l-1}}, c, k)$ is G_{l-1}, where all simplicial vertices in G_{l-1} are joined to the same front. Therefore, G_l is a front instead of the hedron that is represented as the root r. In other words, r does not represent the remaining complete graph when following the elimination scheme over G, H_G. Hence, (T_r, c, k) and $(T'_{r'}, c', k)$ are not isomorphic.

□

Chapter 4

Generating k-trees

In this chapter, our objective is to achieve the project's purpose, which is to compute all unlabeled k-trees by considering all the non-equivalent coding trees. We will make use of the theoretical knowledge we have acquired thus far to optimize this computation. Although better mentioned in Section 4.1, a comprehensive implementation that successfully achieves this objective can be accessed in the public GitHub repository [10].

In order to compute all non-equivalent coding trees, all unlabeled ordinary trees in which the center is a single vertex have been computed. This is because Theorem 3.5 states that isomorphic coding trees are equivalent. This computation has been done by means of a well-established algorithm that can be found in [12]. This has been implemented in Sagemath by the *TreeIterator* method from *trees* library [9].

For a k-tree with n vertices, its coding tree has n_T, where:

1. $n_T = n - k$ and the root represents a hedron.
2. $n_T = n - k + 1$ and the root represents a front.

After obtaining a rooted tree T_r with n_T vertices such that $C(T_r) = r$, the next step is to assign a coloration to it, based on the rules of a coding tree. That is, color all vertices except the root, so adjacent vertices have different colors. In addition, in the case the root represents a front, the elements of first generation are assigned with color $k + 1$.

However, we will refrain from computing all possible colorations, as many of them yield equivalent coding trees. This is where theoretical knowledge devel-

oped throughout Chapter 3 has been applied effectively to eliminate redundant encodings.

From all possibles colorations of the rooted tree T_r, we compute only those colorations that are unique under any permutation of colors. In other words, for any two computed colorations of T_r, there is no permutation $\sigma : 1, 2, ..., k+1 \rightarrow 1, 2, ..., k+1$ that can transform one coloration into the other. As stated in Theorem 3.5. If a coloration c of T_r is a permutation of a coloration c' of T_r, the coding trees (T_r, c, k) and (T_r, c', k) are equivalent.

In the specific scenario where the root represents a hedron, we avoid computing colorations where the representatives of the longest branches have the same color. According to Proposition 3.7, these colorations would result in k-trees where the elimination scheme yields a front as the root. Meaning, that we can compute these k-trees by decoding coding trees in which the root represents a front.

After computing the optimized set of colorations, we proceed to decode the corresponding coding trees. But we have to make sure there are still no equivalent coding trees. So we compare k-trees and see if they are isomorphic one to another.

The good news are, we do not have to compare k-trees that come from different rooted trees, since they are always non-equivalent, since all coding trees follow the elimination scheme of the k-trees we obtain when decoding them. This is because of the restrictions made in the colorations.

In the case one wants to compute all unlabeled k-trees inside a range of different values of k and number of vertices. We may take advantage of an asymptotic behavior.

Proposition 4.1. *If $k \geq n - k - 2$, all k-trees with n vertices can be generated by coloring k-coding trees with $n - k - 1$ colors.*

Proof. Let G be any k-tree with n vertices, we distinct two cases:

- The case in which the remaining complete graph when following the elimination scheme over G is a hedron. Then, we can compute G by decoding a coding tree (T_r, c, k) with $n_T = n - k$ vertices, where the root represents a hedron. Therefore, the condition $k \geq n - k - 2$ is equivalent to $k \geq n_T - 2$ and $k + 1 \geq n_T - 1$. Here, $n_T - 1$ is the number of vertices that need to be colored, since the root does not need to be colored. So the condition $k \geq n - k - 2$ states that there are the same number or more colors available than vertices to color $(n_T - 1)$. Notice $c^{-1}(\{1, 2, ..., k+1\})$ is a partition of the set $V(T_r) \setminus \{r\}$, thus it is a set of at maximum $n_T - 1$ sets. Hence, c sends a

vertex $v \in V(T_r) \setminus \{r\}$ to one color from $n_T - 1$ colors. So any coloration c is a permutation of a coloration $c' : V(T_r) \setminus \{r\} \to \{1, 2, ..., n_T - 1\}$.

- The case in which the remaining complete graph when following the elimination scheme over G is a front. Then, we can compute G by decoding a coding tree (T_r, c, k) with $n_T = n - k + 1$ vertices, where the root represents a front. Therefore, the condition $k \geq n - k - 2$ is equivalent to $k + 1 \geq n - k + 1 - 2$ and $k + 1 \geq n_T - 2$. Here, $n_T - 2$ is the number of vertices that need to be colored. Since the root does not need to be colored and the first child of the root dictates the color of the rest of elements from first generation. Note that, first generation contains more than one element, since the center is a single vertex. Like before, the condition $k \geq n - k - 2$ states that there are the same number or more colors available than vertices to color ($n_T - 2$). Notice $c^{-1}(\{1, 2, ..., k + 1\})$ is a partition of the set $V(T_r) \setminus \{r\}$, such that it is a set of at maximum $n_T - 2$ sets. Hence, c sends a vertex $v \in V(T_r) \setminus \{r\}$ to one color from $n_T - 2$ colors. So any coloration c is a permutation of a coloration $c' : V(T_r) \setminus \{r\} \to \{1, 2, ..., n_T - 1\}$.

  Hence, any coloration c is a permutation of a coloration c' defined with only $n_T - 2$ colors. Because any combination of colors in the $n_T - 1$ vertices can only use at maximum $n_T - 1$ colors.

  Now, because of Theorem 3.5, the coding trees (T_r, c, k) and (T_r, c', k) are equivalent in both cases, proving the proposition.

□

Let $k_0 = n_0 - k_0 - 2$, because of Proposition 4.1 the set of k_0-trees with n_0 vertices can be generated by decoding the set of coding trees with n_{0_T} vertices and colored with $n_0 - k_0 - 1$ colors, notice $k_0 + 1 = n_0 - k_0 - 1$. Here, n_{0_T} can be $n_0 - k_0$ or $n_0 - k_0 + 1$ depending in what the root represents, a hedron or a front, respectively.

Now, again because of Proposition 4.1, for any integer $i \geq 0$, the set of k_i-trees, where $k_i = k_0 + i$, with $n_i = n_0 + i$ vertices can be generated by decoding the set of coding trees with n_{0_T} vertices and colored with $n_0 - k_0 - 1$ colors. Since the inequality $k_i = k_0 + i \geq n_0 - k_0 - 2 = n_0 + i - k_0 - i - 2 = n_i - k_i - 2$ is satisfied and the equalities $n_i - k_i - 1 = n_0 - k_0 - 1$ and $n_{i_T} = n_{0_T}$ are also satisfied, this last equality comes from the fact that n_{i_T} can be $n_i - k_i$ or $n_i - k_i + 1$ depending in what the root represents.

Thus, by computing all non-equivalent coding trees for k_0-trees with n_0 vertices, we are also computing all non-equivalent coding trees for k_i-trees with n_i

vertices, for any integer $i \geq 0$.

4.1 Implementation

All the algorithms developed in this study have been successfully implemented in SageMath. Furthermore, a specific function for computing all unlabeled k-trees with n vertices has been meticulously crafted, leveraging the knowledge acquired throughout this research endeavor and accomplishing the thesis purpose. All the necessary auxiliary functions and dependencies have also been duly integrated into the implementation.

The complete implementation code, including the aforementioned function and its associated components, is openly available in a public GitHub repository [10]. The repository is licensed under the permissive and open-source MIT license, ensuring the freedom and accessibility of the software.

Within the repository, you will find a comprehensive Jupyter Notebook containing the entire codebase. Additionally, an HTML file is provided, which is essentially a static version of the notebook. This file might be useful for reading the code.

Chapter 5

Conclusion

The generation of all encodings has been successfully accomplished. This leads directly to the generation of all unlabeled k-trees, representing a significant milestone in this thesis. Accompanying this achievement is the implementation of the methodology in SageMath, providing a practical tool for future use.

The utilization of a well-established algorithm specifically designed for generating ordinary trees [12] has been instrumental in achieving this objective. Moreover, the application of the theoretical knowledge developed throughout this study has proven effective in eliminating redundant encodings, thereby streamlining the overall generation process.

It is important to acknowledge that while the current methodology has demonstrated its efficacy, there may still exist alternative approaches to further refine the sifting of encodings. This presents an opportunity for future research to explore and enhance the existing methodology.

Bibliography

[1] R. Diestel, *Graph theory*, Graduate Texts in Mathematics, Springer Berlin Heidelberg, 2017.

[2] Michael Drmota and Emma Yu Jin, *An asymptotic analysis of labeled and unlabeled k-trees*, Algorithmica **75** (2016), no. 4, 579–605 (English).

[3] Andrew Gainer-Dewar, Γ*-species and the enumeration of k-trees*, 2012.

[4] Andrew Gainer-Dewar and Ira M. Gessel, *Counting unlabeled k-trees*, J. Comb. Theory, Ser. A **126** (2014), 177–193 (English).

[5] Camille Jordan, *Sur les assemblages de lignes.*, Journal fur die reine und angewandte Mathematik **70** (1869), 185–190 (fre).

[6] Kolja Knauer and Torsten Ueckerdt, *Clustered independence and bounded treewidth*, arXiv:2303.13655 (2023), 1–15.

[7] J. W. Moon, *The number of labeled k-trees*, J. Comb. Theory **6** (1969), 196–199 (English).

[8] F. J. Soria de Diego, *Graphs*, 2019, Notes from the Graph course in Universitat de Barcelona.

[9] The Sage Developers, *Sagemath, the Sage Mathematics Software System (Version 8.1)*, 2017, `https://www.sagemath.org`.

[10] Alejandro Vara, *Generating k-trees*, `https://github.com/AlejandroVara/Generating_k-trees`, 2023.

[11] D.B. West, *Introduction to graph theory*, Pearson Modern Classics for Advanced Mathematics Series, Pearson, 2018.

[12] Robert Alan Wright, Bruce Richmond, Andrew Odlyzko, and Brendan D. McKay, *Constant time generation of free trees*, SIAM Journal on Computing **15** (1986), no. 2, 540–548.