



HAL
open science

On the similarities of trees: the interest of enumeration and compression methods

Florian Ingels

► **To cite this version:**

Florian Ingels. On the similarities of trees: the interest of enumeration and compression methods. Discrete Mathematics [cs.DM]. Ecole normale supérieure de lyon - ENS LYON, 2022. English. NNT : 2022ENSL0010 . tel-03908078

HAL Id: tel-03908078

<https://theses.hal.science/tel-03908078v1>

Submitted on 20 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Numéro National de Thèse : 2022ENSL0010

THÈSE

en vue de l'obtention du grade de Docteur, délivré par
l'École Normale Supérieure de Lyon

École Doctorale n°512
École Doctorale en Informatique et Mathématiques

Discipline : Mathématiques

Soutenue publiquement le 19/09/2022, par :
Florian Ingels

**On the similarities of trees : the interest
of enumeration and compression methods**

**Sur la similarité des arbres : l'intérêt des
méthodes d'énumération et de compression**

Devant le jury composé de :

VALIENTE Gabriel Professeur, Université de Catalogne, Espagne	Rapporteur
VIALANEIX Nathalie Directrice de Recherche, INRAe, Toulouse	Rapporteure
SAGOT Marie-France Directrice de Recherche, Inria, Université Lyon 1	Présidente
STEHLÍK Matěj Professeur des Universités, Université Paris Cité	Examineur
GODIN Christophe Directeur de Recherche, Inria, ENS Lyon	Directeur de thèse
AZAÏS Romain Chargé de Recherche, Inria, ENS Lyon	Co-encadrant de thèse

À ma famille.

HIC SUNT DRACONES

Abstract

Tree data appear naturally in many scientific domains. Their intrinsically non-Euclidean nature and the combinatorial explosion phenomenon make their analysis delicate. In this thesis, we focus on three approaches to compare trees, notably through the prism of a lossless compression technique of trees into directed acyclic graphs.

First, concerning tree isomorphism, we consider an extension of the classical definition to labeled trees, which requires that trees are identical up to label rewriting. This problem is as hard as graph isomorphism, and we have developed an algorithm that drastically reduces the size of the solution search space which is then explored with a backtracking strategy.

When two trees are different, we may try to find common substructures. If this question has already been addressed for subtrees, we are interested in a larger problem, namely finding sets of subtrees appearing simultaneously. This leads us to consider forest enumeration, for which we propose a reverse search algorithm that constructs an enumeration tree whose branching factor is linear.

Finally, from a list of common substructures, one can build a convolution kernel allowing to tackle classification problems. We consider the subtree kernel from the literature, and build an algorithm that explicitly enumerates subtrees (unlike the original method). In particular, our approach allows us to parameterize the kernel more finely, significantly improving its classification abilities.

Keywords rooted trees; directed acyclic graphs; enumeration; tree isomorphism; reverse search; convolution kernels

Résumé

Les arbres sont des données qui apparaissent naturellement dans de nombreux domaines scientifiques. Leur nature intrinsèquement non euclidienne ainsi que le phénomène d'explosion combinatoire rendent leur analyse délicate. On s'intéresse dans cette thèse à trois approches permettant de comparer des arbres, sous le prisme notamment d'une technique de compression sans perte des arbres par des graphes dirigés acycliques.

D'abord, concernant l'isomorphisme d'arbres, nous considérons une extension de la définition classique aux arbres étiquetés, qui requiert que les arbres soient identiques à réécriture des étiquettes près. Ce problème est aussi dur que l'isomorphisme de graphes, et nous avons développé un algorithme qui réduit drastiquement la taille de l'espace de recherche des solutions, qui est ensuite exploré avec une stratégie de retour sur trace.

Lorsque deux arbres sont différents, on peut chercher à en trouver des sous-structures communes. Si cette question a déjà été traitée pour les sous-arbres, nous nous intéressons à un problème plus large, celui de trouver des ensembles de sous-arbres apparaissant simultanément. Cela nous amène à considérer l'énumération des forêts, pour laquelle nous proposons un algorithme de type "reverse search" qui construit un arbre d'énumération dont le facteur de branchement est linéaire.

Enfin, à partir d'une liste de sous-structures communes, on peut construire un noyau de convolution qui permet d'aborder des problèmes de classification. Nous reprenons de la littérature le noyau des sous-arbres, et construisons un algorithme qui les énumère explicitement (contrairement à la méthode originale). Notre approche permet notamment de paramétrer plus finement le noyau, améliorant significativement les capacités de classification.

Mots-clés arbres enracinés; graphes dirigés acycliques; énumération; isomorphisme d'arbres; recherche inverse; noyaux de convolution

Remerciements

À Romain Azaïs, pour à peu près tout, des débats enflammées sur les conséquences d'un résultat à l'explication des règles du tennis;

À Christophe Godin, pour les discussions (scientifiques ou non), les conseils inestimables et la confiance accordée;

À Gabriel Valiente et Nathalie Vialaneix, qui ont gentiment accepté de rapporter cette thèse et pour leurs retours précieux;

À Marie-France Sagot et Matěj Stehlík, pour avoir tout aussi gentiment accepté de participer à mon jury de thèse;

Aux membres de l'équipe Mosaic, du GN1 et du RDP, pour ces 3 années de pauses cafés, de discussions passionnantes et de soirées au foyer;

À Cécile Mercadier, sans qui rien de tout ceci ne serait arrivé;

Aux amis, pour les soirées jeux, les bons repas et tout ce qui m'a permis de souffler entre deux tentatives d'attraper une preuve qui ne voulait pas se laisser avoir;

Évidemment à mes parents et ma famille qui, sans toujours bien comprendre ce que je faisais – le savais-je moi-même ? –, m'ont toujours permis de suivre la voie qui me plaisait et m'ont soutenu sans faille;

Merci.

Preamble

My thesis takes place in the framework of the RObotics for MIcro-farms (ROMI) project¹, dedicated to promote a sustainable, local, and human-scale agriculture. ROMI is developing an affordable, multipurpose platform adapted to support organic and polyculture market-garden farms. ROMI (and therefore my work) has been financed by European Commission's Horizon 2020.

1: <https://romi-project.eu>

Before addressing the details of my work, I would like to introduce the different facets of the ROMI project, and contextualize my contribution to the project. The following partners are involved in the project:

- ▶ Institute of Advanced Architecture of Catalonia (Iaac / FabLab Barcelona / Noumena),
- ▶ Sony Computer Science Laboratories Paris,
- ▶ Inria,
- ▶ Centre national de la recherche scientifique (CNRS / ENS Lyon),
- ▶ Humboldt-Universität zu Berlin,
- ▶ Pépinières Chatelain & Chatelain Maraîchage,
- ▶ France Europe Innovation.

The ROMI project The general philosophy of the ROMI project is to propose low-cost, open source technological solutions to help micro-farms; this approach involves both hardware and software solutions.

This takes the form, for instance, of a mechanical weeding robot. Weeding is often a tedious, time-consuming and without much added value task – if one wants to avoid chemicals, one has to do it manually. The robot is mounted on wheels and, through a camera and a machine learning algorithm, is able to identify which areas are growing vegetables and which are dirt. A mechanical arm then digs up the soil, avoiding the vegetables, which prevents the proliferation of weeds.

2: <http://www.ens-lyon.fr/RDP/>

Another solution envisaged by ROMI relies on the construction of a 3D plant scanner, indoor at the moment but envisaged to be an outdoor phenotyping station in the future. This 3D scanner was developed jointly with Inria, CNRS and Sony CSL. Inria and CNRS actually interact together within the RDP laboratory², where I did my thesis; my work is hence inscribed in the context of this scanner.

3D plant scanner Phyllotaxis describes the arrangement of leaves and branches of a plant, at a macroscopic level. In other words, it is the description of the topology of the plant. Phenotyping is – in our context – the action of measuring this phyllotaxis.

Biologists routinely study a plant named *Arabidopsis Thaliana* (see Figure 1), used as a model organism – i.e., a species widely studied with the expectation that biological phenomena that can be explained on this species can be generalized to other species [1]. In such a plant, phyllotaxis is studied by measuring the angle between two consecutive branches (the divergence angle) and the distance between them along the main



Figure 1: *Arabidopsis Thaliana*, courtesy of Sana Dieudonné.

[1]: Fields et al. (2005), 'Whither model organism research?'

[2]: Besnard et al. (2014), 'Cytokinin signalling inhibitory fields provide robustness to phyllotaxis'

[3]: Guédon et al. (2013), 'Pattern identification and characterization reveal permutations of organs as a key genetically controlled property of post-meristematic phyllotaxis'



Figure 2: Phenotyping station, ©ROMI.

[4]: Chaudhury et al. (2020), 'Skeletonization of plant point cloud data using stochastic optimization framework'

[5]: Mirande et al. (2020), 'High-precision 3D segmentation of plants combining spectral clustering and quotient graph techniques: a multi-level approach'

stem (the internode length). Measuring plant phyllotaxis has contributed to significant discoveries in biology [2, 3] even though phenotyping is currently done by hand.

Being able to automate these measurements would be of great interest to biologists. More generally, having a 3D reconstruction of a plant makes it possible to imagine even more ambitious uses of the study of phyllotaxis, since we would have the complete topology of the plant. One could, for example, imagine a more systematic comparison of phyllotaxis between wild type plants and their mutants; or between plants subjected to hydric stress and others that have not, etc. The 3D scanner intends to meet this ambition of making phenotyping easier, and thus more frequent in the work of biologists. For farmers, one can imagine similar applications to compare healthy or sick plants, etc.

The 3D reconstruction pipeline that has been developed in the ROMI project is as follows. The scanner itself is a structure allowing to place a plant (typically about fifteen centimeters high), and where a robotic arm equipped with a camera will take images under different angles – see Figure 2. These photos are then processed by an algorithm that reconstructs a 3D point cloud reproducing the scanned plant.

The point cloud is then segmented, i.e. each point is given a label indicating to which individual organ it is believed to belong; and a skeleton is also calculated to represent the underlying structure. Finally, skeleton and segmentation are used to determine internode lengths and divergence angles, which provide the desired measure of phenotype.

The calculation from the point cloud of the skeleton [4] and the segmentation [5] have already been published. A diagram of the pipeline can be found in Figure 3.

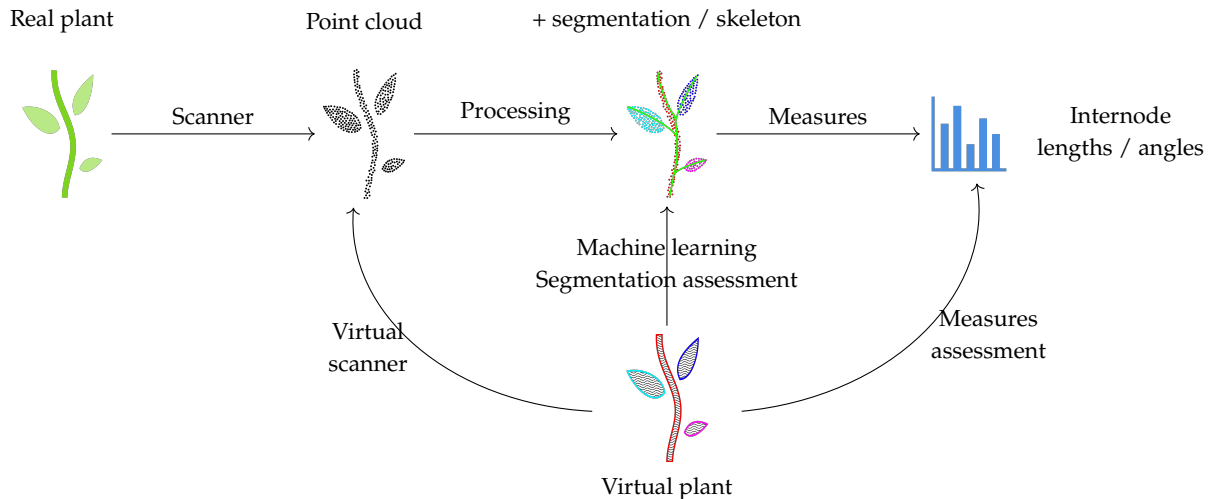


Figure 3: Reconstruction pipeline for the ROMI Scanner. The plant and histogram icon are ©Delapouite, <https://game-icons.net>.

Since the phyllotaxis of a plant can be measured by hand, pipeline effectiveness can be attested by comparing the manual measure and the one returned at the end of pipeline. Ideally, the margin of error on the latter measurements should be similar or better than the former. Another complementary way to assess the efficiency of the 3D scanner relies on

the introduction of virtual plants, which can provide numerous benefits to improve the pipeline.

Virtual plants A virtual plant, in our context, is a 3D object obtained by a generative model. In this case, the models are based on L-systems [6, 7], formal grammars widely used in virtual plant synthesis [8]. We use the LPy software [9], dedicated to the construction of such L-systems, and which also allows their 3D geometrical interpretation. To synthesize a particular plant species, it is necessary to design a specific L-system (which will give plants all the more faithful to reality as the model is detailed); the same model allows to generate different virtual plants thanks to a controllable stochasticity. For instance, Figure 4 represents a virtual *Arabidopsis* obtained with the LPy software.

Application of virtual plants When we build a virtual plant, we know exactly its topology and geometry. A number of applications of this information can be found in the context of our reconstruction pipeline.

- ▶ First, we can artificially reproduce the conditions of the real scanner (light, framing, etc.) and produce images, making it possible to create artificial scans without the need to grow actual plants. Moreover, since we know the topology and geometry of the virtual plant, we can measure the true phyllotaxis, and thus estimate the error committed at the output of the pipeline in an absolute way – and not just in comparison with the error committed manually.
- ▶ We know, typically, that one of the prerequisites of machine learning relies on annotated databases allowing to train algorithms. This kind of data is difficult to find in plants – one can however mention [10]. Virtual plants provide a good substitute, since one can create segmented point clouds by sampling the 3D mesh – and then compare with the segmentation obtained by the pipeline [11], or use it with machine learning methods [12].

Such applications are also included in the diagram of Figure 3.

Tree comparison problems Whether to validate measurements or segmentation, the core of the problem lies in the ability to compare two topologies (the structure of the plants) each provided with a geometry. This is true whether using virtual or real plants: the reconstructed structure is compared with the original structure.

Mathematically, the plant structure is called a *tree*, and each node of such a tree is provided with a piece of information (here, the geometry, the type of organ, etc.). The formal problem behind all this is therefore that of the *comparison of tree data*. That is precisely what my thesis is about.

[6]: Lindenmayer (1968), ‘Mathematical models for cellular interactions in development I. Filaments with one-sided inputs’

[7]: Lindenmayer (1971), ‘Developmental systems without cellular interactions, their languages and grammars’

[8]: Prusinkiewicz et al. (2012), *The algorithmic beauty of plants*

[9]: Boudon et al. (2012), ‘L-Py: an L-system simulation framework for modeling plant architecture development based on a dynamic language’



Figure 4: A virtual *Arabidopsis Thaliana* obtained with the LPy software; courtesy of Christophe Godin.

[10]: Dutagaci et al. (2020), ‘ROSE-X: an annotated data set for evaluation of 3D plant organ segmentation methods’

[11]: Chaudhury et al. (2021), ‘Transferring PointNet++ Segmentation from Virtual to Real Plants’

[12]: Chaudhury et al. (2020), ‘3D plant phenotyping: All you need is labelled point cloud data’

Contents

Abstract / Résumé	v
Remerciements	vii
Preamble	ix
Contents	xiii
1 Introduction	1
1.1 Motivations for the study of trees	2
1.2 Methods for comparing trees	3
1.3 Object of the thesis	5
2 Concerning trees	9
2.1 Formal definition	9
2.2 Tree isomorphisms	12
2.3 DAG compression of trees	14
2.4 DAG compression of forests	17
THE TREE CIPHERING ISOMORPHISM PROBLEM	21
3 Tree cipherings	23
3.1 Motivation	23
3.2 Formal definition	25
3.3 A new kind of DAG compression	28
4 On the construction of tree cipherings	37
4.1 Addressing the problem	37
4.2 Framework	39
4.3 The algorithm	44
4.4 Analysis of the algorithm	49
ENUMERATION TREES: FROM TREES TO FORESTS	57
5 From tree to forest enumeration	59
5.1 Enumeration problems	59
5.2 Tree enumeration	62
5.3 Forest enumeration	64
6 Enumeration of forests	71
6.1 Exhaustive enumeration of FDAGs	72
6.2 Growth of the tree	77
6.3 Variations on the enumeration tree	81
6.4 Enumeration of forests of subtrees	85

THE SUBTREE KERNEL REVISITED	91
7 The subtree kernel	93
7.1 Kernel methods	93
7.2 Theoretical study	96
8 A new framework for computing the subtree kernel	101
8.1 Framework	101
8.2 Real data analysis	105
8.3 Interest of the DAG approach	115
9 Perspectives	119
9.1 Tree isomorphisms	119
9.2 Search for frequent patterns	124
9.3 Classification of trees	127
APPENDIX	131
A Technical proofs	133
A.1 Proof of Proposition 2.7	133
A.2 Proofs of Section 3.2	134
A.3 Proof of Theorem 3.6	136
A.4 Proofs of Section 4.4	137
A.5 Proof of Proposition 7.3	144
B A bijection between FDAGs and row-Fishburn matrices	147
B.1 Equivalence between FDAGs and reduced adjacency matrices	147
B.2 Reduced adjacency matrix to incremental adjacency matrix	147
B.3 Incremental adjacency matrix to reduced adjacency matrix	148
Bibliography	151
Index of frequent notations	157

List of Figures

1	Arabidopsis Thaliana.	ix
2	ROMI Phenotyping scanner.	x
3	Reconstruction pipeline for the ROMI Scanner.	x
4	A virtual Arabidopsis.	xi
1.1	A tree.	1
1.2	Three examples of data that can be modeled by trees.	2
1.3	Some classes of graphs with increasing structural complexity.	2
1.4	An example of a tree and its DAG compression.	6
1.5	The treeex logo, drawn by yours truly.	7
2.1	A non-exhaustive bestiary of graphs.	9
2.2	A rooted tree.	10
2.3	Encoding of a tree with Knuth tuples.	11
2.4	Two isomorphic graphs.	12
2.5	Two trees, isomorphic as unordered trees, but not isomorphic as ordered trees.	12
2.6	Classes of equivalence of the nodes of a tree.	13
2.7	An unordered tree and its DAG compression.	15
2.8	DAG compression of a tree.	15
2.9	Tree reconstruction from DAG compression.	16
2.10	Step by step illustration of DAG recompression algorithm.	19
3.1	Pigpen cipher.	23
3.2	A silly ciphering problem.	24
3.3	The tree ciphering isomorphism problem.	24
3.4	Computation of the number of tree isomorphisms.	26
3.5	Binary relation on the alphabets induced by a tree isomorphism.	26
3.6	Intuition for DAG compression of labeled trees.	28
3.7	DAG compression with labels.	31
3.8	Example of tree for DAG compression with labels.	34
3.9	Iterations of COMPRESSIONWITHLABELS algorithm on an example.	34
4.1	A graph and its reduction as a labeled tree.	38
4.2	Illustration of the SPLITCHILDREN procedure.	41
4.3	Illustration of the deduction rules on collections.	43
4.4	Initialization of the algorithm.	45
4.5	After partitioning by depth.	45
4.6	After partitioning by equivalence class.	46
4.7	After partitioning by parents.	47
4.8	State of the system after converting all bags into collections.	47
4.9	End of the preprocessing phase.	48
4.10	$r(\mathbb{B}, \mathbb{C})$ at different steps of the preprocessing.	53
4.11	Proportion of mappings at the end of preprocessing.	54
4.12	Computation time (in seconds) spent in the preprocessing part and the backtracking part.	54
4.13	Time spent in backtracking as a function of the size of the search space.	55
4.14	Logratio $\log_{10}(t_{\perp}/t_{\top})$ with regard to the size and proportion of labels of trees.	56
5.1	An illustration of the reverse search method on permutations.	61
5.2	Rightmost path and expansion rule for ordered trees.	63
5.3	Two different representations of the same unordered tree.	63
5.4	Left-heavy embedding of a tree.	63

5.5	DAG compression of irredundant forests.	65
5.6	Topological orderings of a DAG.	68
5.7	Canonical ordering of a FDAG.	69
6.1	Branching rule.	73
6.2	Elongation rule.	73
6.3	Widening rule.	74
6.4	FDAG enumeration tree.	77
6.5	Number of successors of random FDAGs.	79
6.6	Computation time for the successors of random FDAGs.	80
6.7	A self-nested tree and its DAG reduction.	83
6.8	Construction of a forest of subtrees from a FDAG.	86
6.9	Enumeration tree of the subFDAGs of an FDAG.	87
6.10	Quotient $Q(D)$ according to the number of vertices of D	89
7.1	Embedding data in a feature space in a machine learning context.	93
7.2	Computation of the subtree kernel on an example.	95
7.3	Two trees T_0 and T_1 that fulfill conditions (i) and (ii).	96
7.4	Two trees T_0 and T_1 that fulfill conditions (i), (ii) and (iii).	96
8.1	Discriminance weight functions.	105
8.2	Conversion of HTML code into a tree.	107
8.3	Description of a Wikipedia data set.	108
8.4	Classification results for the 50 Wikipedia databases.	109
8.5	Visualisation of one data set of unordered trees among the Wikipedida databases.	110
8.6	Estimation of the distribution of the discriminance weight function.	111
8.7	Description of INEX data sets.	111
8.8	Classification results for INEX as ordered and unordered trees.	112
8.9	Description of the videogame sellers data set.	113
8.10	Description of the VasuSynth data set and classification results.	113
8.11	Description of the Hicks et al. data set and classification results.	114
8.12	Description of the Faure et al. data set and classification results.	115
8.13	Description of the LOGML data set and classification results.	115
8.14	Relative improvement of the discriminance weight for all data sets.	116
8.15	Computation time for several repetitions of the subtree kernel.	117
9.1	Tree cipherings with structured labels space: an example.	122
9.2	All possible non-isomorphic (up to a cipher) labeled trees of size at most 3.	125
9.3	Example of non-uniqueness of labeled DAGs compressing a given labeled tree.	125
9.4	Interest of considering forests of subtrees instead of subtrees only in kernel-related problems.	129
A.1	Backtracking tree for mapping two sets of $n = 3$ objects.	138
B.1	FDAGs as row-Fishburn matrices.	149

List of Algorithms

1	TREEISOMORPHISM	14
2	REBUILDTREE	16
3	TREECOMPRESSION	17
4	DAGRECOMPRESSION	18
5	REBUILDTREEWITHCIPHER	31

6	COMPRESSIONWITHLABELS	33
7	MAPNODES	41
8	SPLITCHILDREN	41
9	SPLITCHILDRENBISS	43
10	NEXTCANDIDATES	49
11	BACKTRACKING	49
12	REVERSESEARCH	60
13	MINIMALWORDS	67
14	ANTECEDENT	75
15	HEIRS	87
16	FREQUENTHEIRS	88
17	CLASSIFIER	97

List of Tables

1.1	Number $a(n)$ of trees with n nodes for some values of n	5
5.1	Filtering the topological orderings of a DAG.	68
6.1	Number of FDAGs accessible from D_0 in k steps in the enumeration tree.	78
8.1	Summary of the 8 data sets.	116
9.1	First values of the sequences $a(n)$ and $b(n)$ counting the number of non-isomorphic trees of size n , respectively for tree isomorphism and tree isomorphism up to a cipher.	125
9.2	Comparison between the number $g(n)$ of unlabeled connected graphs with n vertices with the number $a(n)$ of trees with n vertices for some values of n	126
A.1	Strategy for deciding the first tuple to process between (m, α) and (n, β) to minimize f	140

Introduction

1

Trees are sanctuaries. Whoever knows how to speak to them, whoever knows how to listen to them, can learn the truth.

Herman Hesse

Throughout my 3 years of PhD, when a friend would ask me what my work was about, I would often tell them, “I study trees... but not the ones that grow outside, mathematical trees.” To this cryptic answer, if the moment allowed it, I gave more details, but too often I did not have the time to continue and had to leave them with this enigma. As I begin this thesis, here comes my opportunity to take the time to explain in detail my work, and unveil the mystery.

The trees I study, although abstract, are just as rich and fertile as those in real life – and contain as many secrets. Many natural questions arise when trying to understand trees. Among them, the ones I am interested in are related to the comparison of trees. How to know when trees are similar? And if they are not, how different are they? What do they have in common?

In the same way that biologists have many techniques to compare actual trees, from their overall shape to their genome, we need dedicated methods to compare mathematical trees. My PhD thesis investigates exactly such comparison methods for trees.

In this introduction, the study of trees is motivated more specifically in [Section 1.1](#). Different methods for tree comparisons are presented in [Section 1.2](#), while [Section 1.3](#) elaborates on the object of this thesis and its outline.

Let us start, however, with a definition.

Definition 1.1 *A connected graph is a set of vertices, connected by edges, such that by walking along the edges, one can, from any vertex, reach any other vertex.*

Then, a (mathematical) tree is a connected graph that contains no cycle – where a cycle is a walk along the edges that allows to return to the starting point without retracing one’s steps; in other words, a loop.

An example of a tree is provided in [Figure 1.1](#).

- 1.1 Motivations for the study of trees 2
 - Trees are ubiquitous 2
 - Trees in graph theory 2
- 1.2 Methods for comparing trees 3
 - Tree isomorphisms 3
 - Search for frequent patterns . . 4
 - Classification of trees via a similarity function 4
- 1.3 Object of the thesis 5
 - Enumeration problems 5
 - Lossless compression of trees 5
 - Outline of the thesis 6
 - A word on implementation . . 6

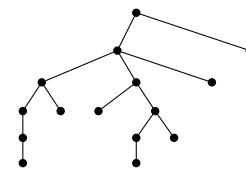
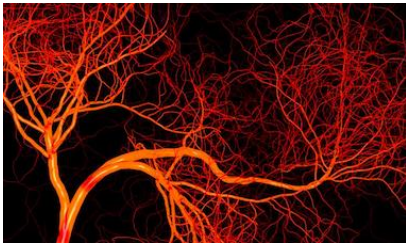


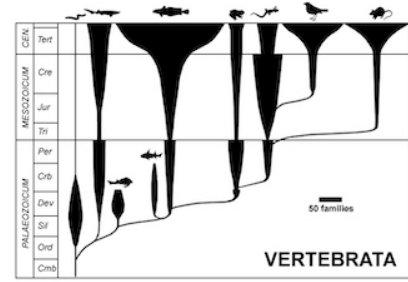
Figure 1.1: A tree.



(a) Blood vessels, ©The Franklin Institute



(b) Actual trees, ©Mustang Joe, Flickr



(c) Phylogenetic trees, ©Wikipedia Commons

Figure 1.2: Three examples of data that can be modeled by trees.

1.1 Motivations for the study of trees

Trees are ubiquitous

[13]: Le et al. (1989), ‘Tree graphs of RNA secondary structures and their comparisons’

[14]: Costa et al. (2004), ‘A Tree-Based Approach to Clustering XML Documents by Structure’

[15]: Martín-Delgado et al. (2002), ‘Density-matrix renormalization-group study of excitons in dendrimers’

[16]: Meagher (1982), ‘Geometric modeling using octree encoding’

[17]: Breiman et al. (2017), *Classification and regression trees*

[18]: Breiman (2001), ‘Random forests’

As we have seen in the preamble, trees appear naturally when we consider the structure of plants. Actually, trees are ubiquitous since they are found in a wide variety of scientific fields, from RNA secondary structures in biology [13] to XML files in computer science [14] through dendrimers in chemistry and physics [15]. Some examples of data that can be represented by trees are shown in Figure 1.2.

Trees are also widely used as a data structure to collect and store information in many algorithms; among which octrees [16] in computer vision, or even decision trees and random forests [17, 18] in machine learning.

In general, trees can model any phenomenon or object whose hierarchical structure does not contain cycles.

Trees in graph theory

In addition to arising naturally in many domains, trees are also of interest as a class of graphs of intermediate structural complexity, between paths (or linear graphs), and general graphs³, as illustrated in Figure 1.3. Indeed, many fundamental questions about graphs, difficult to address in the general case, can be solved in a much simpler or more efficient way on trees (and even more easily on paths).

This is for instance the case of the graph isomorphism problem – about which we will come back at length in the body of the document – or the graph coloring problem, solved for trees, but NP-hard in the general case [19].

It is also sometimes useful to extract a tree – referred to as a spanning tree – from a graph, for optimization purposes. For instance, it can be necessary to ensure that an algorithm, which traverses the nodes of a graph one after the other, can not loop infinitely when it encounters a cycle. Algorithms as famous as Dijkstra’s or A* make an extensive use of spanning trees. We can also mention the Spanning Tree Protocol for Ethernet networks [20]. As an example, the tree of Figure 1.3b is a spanning tree of the graph of Figure 1.3c.

3: Actually, paths are a particular instance of trees, and both are particular instances of graphs.

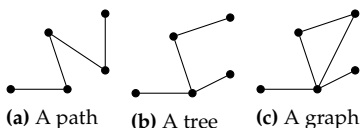


Figure 1.3: Some classes of graphs with increasing structural complexity.

[19]: Kubale (2004), *Graph colorings*

[20]: Perlman (1985), ‘An algorithm for distributed computation of a spanning tree in an extended LAN’

Finally, there is not only one type of tree that can be considered in graph theory. In this thesis, we are interested in rooted trees, i.e. trees where there exists a special node⁴, the root, which has no parent – all other nodes have a single parent. The tree of [Figure 1.1](#) is rooted (the root being on top). Among rooted trees, we distinguish ordered trees – that is, the order of the children of a node is important – from unordered trees – where the order is not important. Moreover, whether ordered or not, the nodes of a rooted tree can also carry a label, which can be of any nature. In the sequel, we will refer to rooted trees simply as “trees”.

4: We refer to nodes, in the context of trees, instead of vertices – used for graphs.

1.2 Methods for comparing trees

While there are many problems to consider concerning trees, my work is primarily concerned with the issue of *comparing* them. We assume that we have at least two trees, of the same nature⁵, and we try to quantify their similarity.

5: In the sense that they must be comparable: it is not allowed to have one ordered and the other not, etc.

Tree isomorphisms

When comparing two objects, of any nature, the most elementary operation is an equality test. It is only when they are not identical that it becomes necessary to refine the way they are compared, to identify their similarities and differences. In the context of trees, this equality test is called a tree isomorphism⁶. A tree isomorphism is a bijection on the nodes that preserves the parent/child relationships.

6: From ancient Greek, “same shape”.

In the case of unlabeled trees, ordered or not, one can determine if two trees are isomorphic in linear time in the number of nodes. The same is true for non-rooted trees. See for instance [\[21, 22\]](#).

[21]: Bonamy (2010), ‘A small report on graph and tree isomorphism’
[22]: Aho et al. (1974), ‘The design and analysis of computer algorithms’

If the trees are labeled, we can adapt the definition of isomorphism in two ways. When we map two nodes together via a bijection, we associate, at the same time, the labels of these nodes. The first definition requires that if two nodes are mapped together, then their labels must be identical – so the constructed association is the identity. This definition yields a problem that is as easy to solve as for unlabeled trees.

For the second definition, we require that a given label in the first tree is only associated to a single other label in the second tree – so that all nodes sharing a label in a tree are associated to nodes that also share a label in the other tree. In a way, the aim is to construct a bijection on the nodes that preserves a partition of the labels [\[23\]](#). With this definition, the problem becomes as difficult as the problem of graph isomorphism [\[24\]](#), which is not resolved in the general case, even if efficient algorithms exist [\[25\]](#).

[23]: Champin et al. (2003), ‘Measuring the similarity of labeled graphs’

[24]: Booth et al. (1979), *Problems polynomially equivalent to graph isomorphism*

[25]: McKay et al. (2014), ‘Practical graph isomorphism, II’

During my thesis, I addressed the problem of isomorphism on labeled trees using the latter definition, and proposed an algorithm to tackle it. This work has been presented and published at the IWOCA 2021 conference [\[26\]](#).

[26]: Ingels et al. (2021), ‘Isomorphic Unordered Labeled Trees up to Substitution Ciphering’

Search for frequent patterns

When two numbers are different, we can still compare them by observing which one is smaller and which one is larger. In the case of trees, we can not define so naturally a notion of “bigger” or “smaller”. Another way to compare numbers can be to list their divisors, and look at the ones they have in common. This approach, on the other hand, can be applied to trees – but instead of divisors, we will look for substructures in common.

This idea of “substructures” is broad and can cover many definitions. Subtrees are such an example. A subtree of a tree is the tree formed by a node and all its descendants in the starting tree. The simplest subtree is a leaf, that is, a node without children. All trees have leaves, so they all have this subtree in common – in the very same way that all numbers are divisible by 1.

[27]: Asai et al. (2003), ‘Discovering frequent substructures in large unordered trees’

[28]: Avis et al. (1996), ‘Reverse search for enumeration’

[29]: Nakano (2002), ‘Efficient generation of plane trees’

[30]: Nakano et al. (2003), ‘Efficient generation of rooted trees’

When we have a large number of trees to compare, we can also try to identify the substructures that appear most frequently. This problem has been tackled for unordered trees [27]. The method used in [27] is based on a famous enumeration technique, the reverse search method [28], and was previously introduced to enumerate ordered [29] and then unordered trees [30].

I am interested in taking this problem – enumerating trees – to a higher order, by enumerating forests – as collections of trees. Indeed, being able to identify frequent subforests present in a dataset is promising: they are richer substructures than subtrees, while containing subtrees (since a tree is a singleton forest) – so one can expect to capture more detailed information and differences not visible with the frequent subtrees alone.

[31]: Ingels et al. (2020), ‘A Reverse Search Method for the Enumeration of Unordered Forests using DAG Compression’

[32]: Ingels et al. (2022), ‘Enumeration of Irredundant Forests’

As for trees in [27, 30], one must be able to enumerate forests before enumerating subforests. So, I developed an enumeration algorithm for unordered forests, also based on the reverse search method. This work has been presented at the WEPA 2020 conference [31], and published in Theoretical Computer Science in 2022 [32].

Classification of trees via a similarity function

One way to organize and compare a large number of objects is to arrange them in families; this is called a classification problem. In the case where we already know the family of a certain number of objects, but not of all of them, we talk about supervised classification: we train an algorithm to recognize the members of the same family and to distinguish them from the other ones, and then the algorithm predicts the family of the objects for which we don’t have any information. This is a machine learning approach, exactly the same way as when an algorithm is trained to recognize pictures of dogs or cats.

A standard way to accomplish this is to construct a similarity function – which measures how alike two objects are – so that objects from the same family have a much higher similarity than objects from different families. Kernels are a powerful way to construct a similarity function, and can be used in conjunction with a support vector machine algorithm

to solve such supervised classification problems. On this topic, we refer the reader to [33].

In particular, for trees, so-called convolution kernels can be used [34, 35], whose premise is that two trees are similar if they share many substructures in common. As earlier, there are many possible definitions of “substructures”, each leading to a new kernel; many examples can be found in [36].

I am particularly interested in the subtree kernel [37], which evaluates the similarity between two trees based on their common subtrees. Whereas the original paper proposed a recursive computation algorithm, I proposed an iterative algorithm that allows a greater finesse in the choice of specific parameters. I have shown, in particular, that this finesse allows to significantly improve the performance of the kernel – i.e. its ability to discriminate families in the classification problem – on some datasets.

This work was presented at the 51st Journées de Statistiques conference in 2019 [38], and published in the Journal of Machine Learning Research in 2020 [39].

1.3 Object of the thesis

Enumeration problems

The three problems previously introduced are all, since they concern trees, eminently combinatorial problems. The difficulty lies, in fact, not in the intrinsic complexity of the structures, but rather in the exponential multiplication of possibilities as the size of the trees increases. As an illustration, one can find in Table 1.1, how the number of (unlabeled, unordered, rooted) trees increases as tree size increases, a result due to Cayley – who also coined the term “tree” [40].

The main criterion to take into account when designing algorithms to solve the above-mentioned problems is their efficiency: we want to be able to obtain an answer in a few seconds, minutes, maybe hours, but barely more. For this reason, most algorithms try to get around the difficulty by never making explicit the part where you have to explore and enumerate all possibilities.

In my thesis, I tried to take the opposite side – whenever possible – and to show that there is something to be gained by enumerating explicitly. By virtue of the above, this ambition requires even more attention to design effective algorithms. As such, parsimonious enumeration is of utmost importance – in the sense that we can not afford to enumerate superfluous objects; this includes both duplicates (a same object obtained several times) and undesirables (objects not answering the problem).

Lossless compression of trees

Lossless compression methods are a way of condensing one object into another, occupying a reduced size, so that no information is lost (as is the case with RAR compression for files), and the original object can be reconstructed identically.

[33]: Cristianini et al. (2000), *An introduction to support vector machines and other kernel-based learning methods*

[34]: Haussler (1999), *Convolution kernels on discrete structures*

[35]: Collins et al. (2001), ‘Convolution kernels for natural language’

[36]: Da San Martino (2009), ‘Kernel methods for tree structured data’

[37]: Vishwanathan et al. (2004), ‘Fast kernels for string and tree matching’

[38]: Ingels et al. (2019), ‘De l’importance de la fonction de poids dans le noyau des sous-arbres’

[39]: Azaïs et al. (2020), ‘The weight function in the subtree kernel is decisive’

n	$a(n)$
1	1
10	719
20	12,826,228
30	354,426,847,597

Table 1.1: Number $a(n)$ of (unlabeled, unordered, rooted) trees with n nodes for some values of n – sequence A000081 in *The On-Line Encyclopedia of Integer Sequences* (2022), <https://oeis.org/A000081>.

[40]: Cayley (1875), *On the Analytical Forms Called Trees: With Application to the Theory of Chemical Combinations*

[41]: Bille et al. (2015), ‘Tree compression with top trees’

[42]: Sutherland (1964), ‘Sketchpad a man-machine graphical communication system’

[43]: Hart et al. (1991), ‘Efficient antialiased rendering of 3-D linear fractals’

[44]: Buneman et al. (2003), ‘Path queries on compressed XML’

[45]: Frick et al. (2003), ‘Query evaluation on compressed trees’

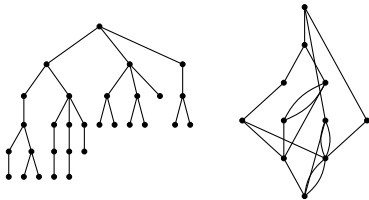


Figure 1.4: An example of a tree (left) and its DAG compression (right).

[26]: Ingels et al. (2021), ‘Isomorphic Unordered Labeled Trees up to Substitution Ciphering’

[32]: Ingels et al. (2022), ‘Enumeration of Irredundant Forests’

[39]: Azaïs et al. (2020), ‘The weight function in the subtree kernel is decisive’

Several compression methods exist for trees, among which top trees compression [41], but also Directed Acyclic Graph (DAG) compression [42–45], in which I am particularly interested.

The principle of DAG compression is to factorize the identical (thus isomorphic) subtrees and to create a graph condensing all the structure of the tree – an example is provided in Figure 1.4. Since no information is lost, having a tree or its compressed form is strictly identical; and since I am interested in how to compare trees, the question of directly comparing compressed forms arises.

In fact, most of the time, the algorithms I built use the compressed form of the trees, not the trees themselves. By proceeding in this way, it is possible to decrease the difficulty arising from the combinatorial and enumerative aspect mentioned above.

Outline of the thesis

Upcoming Chapter 2 of this thesis formally introduces the definitions of trees, tree isomorphism and DAG compression already mentioned earlier in this introduction. The rest of the thesis is divided into three parts, independent of each other, and resume the three comparison methods mentioned above.

The first part, *The Tree Ciphering Isomorphism Problem*, is concerned with the isomorphism problem on labeled trees – where the definition used, imposing that the bijection of the nodes induces a bijection on the labels, induces the greatest difficulty. The problem is formalized in Chapter 3, while an algorithm addressing it is proposed in Chapter 4. The basis of this work is derived from [26], but is here expanded with additional content.

In the context of the search for frequent patterns, the second part, *Enumeration Trees: from Trees to Forests*, focuses on what has already been done for trees in Chapter 5, and introduces the problem of enumerating forests. Chapter 6 proposes an algorithm addressing the latter, as well as its theoretical analysis. An extension is also proposed to deal with related issues, including the frequent subforests mining problem. This part is mainly reproduced from [32].

The third part, *The Subtree Kernel Revisited*, introduces supervised classification problems on trees through the prism of the subtree kernel in Chapter 7, for which a new computational framework is proposed in Chapter 8, together with analyses on real databases. The contributions in this part are derived from [39].

Finally, Chapter 9 summarizes the different contributions of my thesis and opens a discussion on the research directions they induce.

A word on implementation

As mentioned, the nature of the problems addressed in this thesis makes it necessary to develop efficient algorithms. Thus, beyond my traditional tools as a mathematician – paper, pencil, board, chalk – I also devoted an important part of my work to the implementation of many algorithms

and data structures, in order to be able to put the fruit of my efforts to the test.

Thus, all of the algorithms of my design that can be found in this thesis, as well as a large number of algorithms from the literature that will be presented, have been implemented with the goal of being as efficient as possible in mind, this constraint also feeding my theoretical design work.

All of this has been incorporated into the Python `treex` library, developed within my team, dedicated to rooted trees, and capable of handling ordered or unordered, labeled or unlabeled trees, and offering a wide variety of procedures – random generation, processing algorithms, visualization, etc [46].

Excluding documentation and unit tests, my contribution represents almost 2,000 lines of code. One can thus find a module dedicated to the computation of labeled tree isomorphisms; another for the enumeration of trees, forests and subforests; and also a module for the computation of the subtree kernel.



Figure 1.5: The `treex` logo, drawn by yours truly.

[46]: Azaïs et al. (2019), ‘`Treex`: a Python package for manipulating rooted trees’

Concerning trees

2

Trees are as close to immortality as the rest of us ever come.

Karen Joy Fowler

This chapter introduces the most important concepts that will be used in the rest of this thesis. The principal notations are also introduced here; in this regard, see also the index of frequent notations at the end of the document.

Section 2.1 introduces the necessary vocabulary of graph theory allowing then to properly define *rooted trees* which are at the heart of our concerns in this thesis. Considerations on encoding and random tree generation are also discussed.

The notion of tree isomorphism is extensively detailed in Section 2.2, both from a theoretical and algorithmic point of view. Finally, the DAG compression for trees and forests is presented, respectively, in Section 2.3 and Section 2.4. These two concepts will be at the core of the various problems discussed later.

- 2.1 Formal definition 9
 - Graph vocabulary 9
 - Rooted trees 10
 - Encoding of trees 11
 - Random trees 12
- 2.2 Tree isomorphisms 12
 - Definitions 12
 - The Aho, Hopcroft & Ullman algorithm 13
- 2.3 DAG compression of trees . 14
 - Overview 14
 - Formal definition 15
 - Construction 16
- 2.4 DAG compression of forests .17
 - Definition 17
 - DAG recompression 17
 - Connection between a forest and its compressed form 19

2.1 Formal definition

Graph vocabulary

A *graph* is a pair $G = (V, E)$ where V is a finite set of elements called *vertices*, and E is a set of paired vertices, called *edges*. We exclude here that an edge connects a vertex to itself: loops are forbidden. A *path* in a graph is a finite sequence of edges such that the ending vertex of each edge is the starting vertex of the next edge – as in Figure 2.1a.

Directed A graph is said to be *directed* if the edges have orientations, i.e. if we distinguish the edge going from u to v from the one going from v to u . In the case of a directed graph, we refer to *arcs* rather than edges. A directed graph – or *digraph* – is presented in Figure 2.1b.

Connected A graph G is said to be *connected* if, for any two vertices u and v , G contains a path that connects u and v . In the case where G is directed, it is connected if and only if its undirected version – i.e. where each directed arc is replaced by an undirected edge – is connected. In other words, a connected graph is composed of a single fragment. A disconnected graph is shown in Figure 2.1c.

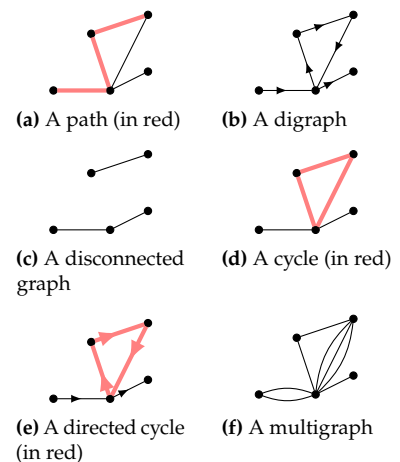


Figure 2.1: A non-exhaustive bestiary of graphs.

Acyclic A *cycle* is a path that connects a vertex with itself – see Figure 2.1d. A graph is said to be *acyclic* if it contains no cycles. A directed graph that contains no directed cycles – where an example of directed cycle can be found in Figure 2.1e – is called a *Directed Acyclic Graph* (DAG) .

Multigraph A *multigraph* is a graph in which it is permitted to have several edges or arcs connecting the same two vertices. An example is provided in Figure 2.1f.

In this thesis, all the graphs considered are connected, directed and acyclic. Some of them are, in addition, multigraphs.

Rooted trees

Definition 2.1 A rooted tree T is a connected graph with no cycle such that there exist a unique vertex $\mathcal{R}(T)$, called the root, which has no parent, and any vertex different from the root has exactly one parent.

The leaves $\mathcal{L}(T)$ are all the vertices without children. A tree is a directed graph, the arcs being directed from the root to the leaves, from parent to children. Since only rooted trees are considered in this thesis, they will be referred to simply as trees in the sequel. An example is provided in Figure 2.2.

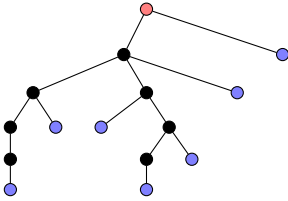


Figure 2.2: A rooted tree. The root is displayed in red, and the leaves in blue.

Topological properties A number of quantities can be calculated on a tree, among which height, depth and outdegree. The height of a node v of a tree T can be recursively defined as

$$\mathcal{H}(v) = \begin{cases} 0 & \text{if } v \in \mathcal{L}(T), \\ 1 + \max_{u \in C(v)} \mathcal{H}(u) & \text{otherwise;} \end{cases} \quad (2.1)$$

where $C(v)$ denotes the set of children of v . The height $\mathcal{H}(T)$ of the tree T is defined as the height of the root, i.e., $\mathcal{H}(T) = \mathcal{H}(\mathcal{R}(T))$. For $0 \leq h \leq \mathcal{H}(T)$, T^h denotes the set of nodes of T with height h . Note that the mapping $h \mapsto T^h$ can be constructed in linear time, by a traversal of the tree.

The depth of a node v is recursively defined as

$$\text{depth}(v) = \begin{cases} 0 & \text{if } v = \mathcal{R}(T), \\ 1 + \text{depth}(\mathcal{P}(v)) & \text{otherwise;} \end{cases} \quad (2.2)$$

where $\mathcal{P}(v)$ designates the parent of v . The depth of the tree T is defined as the maximum depth among all nodes. Notably, $\mathcal{H}(T) = \text{depth}(T)$.

Finally, the outdegree of T is the maximal branching factor that can be found in T , that is $\text{deg}(T) = \max_{v \in T} \#C(v)$. The outdegree of a node v is thus simply defined as $\text{deg}(v) = \#C(v)$.

Particular trees In this thesis, we are interested in specific classes of trees. Thus, trees can be ordered or unordered, labeled or not.

Trees are said to be *ordered* if the order in which the children of a node appear is significant. If not, trees are said to be *unordered*.

A *labeled* tree is a tree in which each node is given a label, whose nature can be arbitrary, and we denote the label of a node v by \bar{v} . The set of labels of a tree T is denoted by $\mathcal{A}(T)$ and defined as $\mathcal{A}(T) = \{\bar{v} : v \in T\}$. $\mathcal{A}(T)$ is also called the *alphabet* of T .

In the rest of the thesis, it will be specified when the considered trees are ordered or not, labeled or not. To simplify the notations, we shall indifferently denote by \mathcal{T} the set of considered trees.

Forests The literature acknowledges two definitions for a forest [47].

- ▶ An undirected, disconnected, acyclic graph. Each connected component is therefore an unrooted tree.
- ▶ A disjoint union of trees, where the trees can be of any nature.

The two definitions coincide when the trees considered are unrooted (and therefore unordered). In this thesis, we adopt the second definition.

Definition 2.2 A forest F is a finite set of trees, i.e. $F = \{T_1, \dots, T_n\}$ where $\forall i, T_i \in \mathcal{T}$.

Encoding of trees

In addition to their representation as graphs, ordered trees can be encoded in different forms, related to other fields of mathematics or computer science. Typically, an ordered tree can be encoded as a sequence – such as Prüfer sequences [48]. We are particularly interested here in Knuth tuples, that can be found in [49].

We associate to each leaf in the tree the tuple “()”. Then, by increasing height, each node receives the concatenated tuple “($t_1 \cdots t_n$)” where t_i is the tuple associated to the i -th child of that node. The tuple associated to the tree is the one of its root. An example is provided in Figure 2.3.

This technique associates a unique tuple with an ordered tree, and one can reconstruct the original tree from the tuple. Therefore, the set of ordered trees is in bijection with the set of well parenthesized expressions. Alternatively, by replacing “(” with 0 and “)” with 1, ordered trees can be interpreted as sequences or strings. For instance, the tree of Figure 2.3 is encoded with the sequence 0010010110010111.

Interpreting 0 as +1 and 1 as -1, we can read this sequence as an excursion (i.e. a walk that comes back to the origin) in \mathbb{Z} , starting at 0. This walk also draws the graph of a function, which is called the Harris path of the tree [50, 51].

In the case of unordered trees, this tuple is not unique (except in pathological cases). In the example of Figure 2.3, if we swap the nodes of depth 1 to place the leaf between its two siblings (or after them), we obtain a different tuple for the tree.

[47]: Bender et al. (2010), *Lists, decisions and graphs*

[48]: Prüfer (1918), ‘Neuer beweis eines satzes über permutationen’

[49]: Knuth (2013), *Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees—History of Combinatorial Generation*

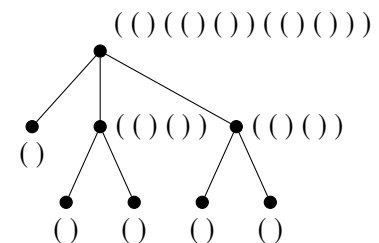


Figure 2.3: Encoding of a tree with Knuth tuples.

[50]: Aldous (1993), ‘The continuum random tree III’

[51]: Azaïs et al. (2019), ‘Inference for conditioned Galton-Watson trees from their Harris path’

Note that this method is purely topological and ignores any labels. It can be extended simply by writing the label of a node just after “(”, but before the concatenation of children tuples.

Random trees

We refer to *random graphs* when we define a probability distribution on the space of graphs, or alternatively when we describe a random process allowing to generate graphs [52]. Naturally this also applies to trees, for which there are many ways to generate random trees.

We can mention for example Galton-Watson trees [53] and conditioned Galton-Watson trees, the latter being closely related to Harris paths mentioned above [51]; or RANRUT algorithm [54, 55], which allows to construct random trees of fixed size following a uniform distribution.

In this thesis, the occurrences of random trees are exclusively *recursive random trees* [56], constructed as follows: beginning with a single node tree, we add nodes one by one, iteratively, by choosing uniformly at random a node already built and adding a leaf to it.

[52]: Bollobás (2001), *Random Graphs*

[53]: Neveu (1986), ‘Arbres et processus de Galton-Watson’

[51]: Azaïs et al. (2019), ‘Inference for conditioned Galton-Watson trees from their Harris path’

[54]: Nijenhuis et al. (2014), *Combinatorial algorithms: for computers and calculators*

[55]: Alonso et al. (1994), ‘Random Unlabelled Rooted Trees Revisited’

[56]: Zhang (2015), ‘On the number of leaves in a random recursive tree’

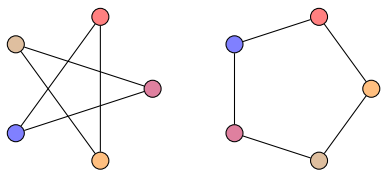


Figure 2.4: Two isomorphic graphs. An isomorphism is provided via the color of the vertices: two vertices with the same color are in bijection.

7: It is assumed that both trees are either ordered or unordered.

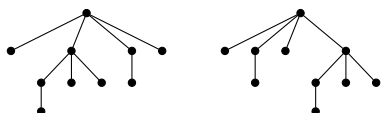


Figure 2.5: Two trees, isomorphic as unordered trees, but not isomorphic as ordered trees.

2.2 Tree isomorphisms

Definitions

A graph isomorphism between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a bijection $\phi : V_1 \rightarrow V_2$ such that $(u, v) \in E_1$ if and only if $(\phi(u), \phi(v)) \in E_2$. When there is such a bijection between two graphs, we say that they are isomorphic, and we note $G_1 \simeq G_2$ – as in Figure 2.4. The set of all isomorphisms between G_1 and G_2 is denoted by $\text{Isom}(G_1, G_2)$. Graph isomorphism is an equivalence relation on the set of graphs. Determining whether two graphs are isomorphic is an algorithmically complicated problem, on which we elaborate extensively in Chapter 3.

At this point, we shall restrict ourselves to trees. Let T_1 and T_2 be two unlabeled trees. Depending on whether the trees are ordered or not⁷, a different definition will be adopted, as follows.

Definition 2.3 A bijection $\phi : T_1 \rightarrow T_2$ is an unordered tree isomorphism if and only if, for any $u, v \in T_1$, if u is a child of v in T_1 , then $\phi(u)$ is a child of $\phi(v)$ in T_2 .

Definition 2.4 A bijection $\phi : T_1 \rightarrow T_2$ is an ordered tree isomorphism if and only if, for any $u, v \in T_1$, if u is the k^{th} child of v in T_1 , then $\phi(u)$ is the k^{th} child of $\phi(v)$ in T_2 .

An illustration is provided in Figure 2.5. As for graph isomorphism, if a tree isomorphism exists between two trees, ordered or not, they are called *isomorphic*, which is denoted by $T_1 \simeq T_2$. Tree isomorphism is an equivalence relation on the set of trees \mathcal{T} [57]. The set of all tree isomorphisms between T_1 and T_2 is, as well, denoted by $\text{Isom}(T_1, T_2)$. Its cardinality is investigated in upcoming On the number of tree isomorphisms (p. 25).

[57]: Valiente (2013), *Algorithms on trees and graphs*

In the case where the trees are labeled, we can adapt the previous definitions by imposing in addition that the labels of the mapped nodes are identical, i.e., $\bar{u} = \phi(u)$. A less restrictive definition also exists [23], which requires that the isomorphism preserves a partition of the labels. Specifically, all nodes $u \in T_1$ that share a common label $a \in \mathcal{A}(T_1)$ should only be mapped to nodes $v \in T_2$ sharing a common label $b \in \mathcal{A}(T_2)$ – thus associating in a bijective manner labels a and b . Chapter 3 and Chapter 4 explore this latter definition in detail.

Remark 2.1 The previously introduced ordered and unordered trees can actually be defined more formally via the isomorphisms defined above. The set of unordered trees is defined as the quotient set of rooted trees by the equivalence relation induced by the unordered tree isomorphisms. The same applies to ordered trees.

The concept of tree isomorphism allows us to introduce some additional notions to complement the previous section.

Subtrees For any node v of a tree T , the subtree $T[v]$ rooted in v is the tree composed of v and all its descendants $\mathcal{D}(v)$. $\mathcal{S}(T)$ denotes the set of all subtrees of T , up to isomorphism. Formally, $\mathcal{S}(T) = \{T[v] : v \in T\}/\simeq$.

Classes of equivalence We denote by $[v]$ the class of equivalence of a node $v \in T$, i.e. the set of all nodes $u \in T$ such that $T[v] \simeq T[u]$. An example is provided in Figure 2.6.

The Aho, Hopcroft & Ullman algorithm

The question remains to determine, algorithmically, whether two trees are isomorphic. The ordered case is easily solved, as per the following theorem. As stated in [21], the key is to go through both trees simultaneously, by depth-first search, and number the nodes in the order of traversal. This induces a natural bijection between the nodes, which can be easily checked as an isomorphism.

Theorem 2.1 *Ordered tree isomorphism can be decided in linear time.*

In the unordered case, the result can be obtained by the virtue of the Aho, Hopcroft & Ullman (AHU) algorithm [22], where the following result is proven.

Theorem 2.2 *Unordered tree isomorphism can be decided in linear time.*

In reality, as mentioned in [58], this complexity is only achieved by assuming that the trees can be encoded within a fixed number of bits. Without this assumption, running AHU algorithm on trees of size n is rather $O(n \log(n))$ than $O(n)$.

We present with Algorithm 1 a slightly different version of the AHU algorithm presented in [21, 22]. First, our version omits to check some stopping cases taken into account in AHU⁸, for the sake of simplicity.

[23]: Champin et al. (2003), ‘Measuring the similarity of labeled graphs’

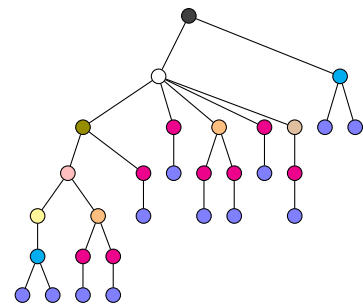


Figure 2.6: An (unordered) tree where nodes with identical equivalence class have been colored identically.

[21]: Bonamy (2010), ‘A small report on graph and tree isomorphism’

[22]: Aho et al. (1974), ‘The design and analysis of computer algorithms’

[58]: Campbell et al. (1991), ‘Tree isomorphism algorithms: Speed vs. clarity’

8: For example, if T_1 and T_2 do not have the same number of nodes, outdegree or height, they can not be isomorphic.

9: It suffices to add the label \bar{v} as the first element of the list C_v constructed in Line 8 – but after the eventual sorting step.

Algorithm 1: TREEISOMORPHISM

Input: $T_1, T_2, \star \in \{\text{ord.}, \text{unord.}\}$

Output: \top if and only if $T_1 \simeq T_2$

```

1 if  $\mathcal{H}(T_1) \neq \mathcal{H}(T_2)$  then
2   | return  $\perp$ 
3 else
4   |  $i \leftarrow 0$ 
5   | Let  $f : \emptyset \mapsto 0$ 
6   | for  $h$  from 0 to  $\mathcal{H}(T_1)$  do
7     | for  $v \in T_1^h \cup T_2^h$  do
8       |  $C_v \leftarrow [N_u : u \in C(v)]$ 
9       | if  $\star = \text{unord.}$  then
10        | Sort  $C_v$ 
11        | if  $f(C_v)$  is defined then
12          |  $N_v \leftarrow f(C_v)$ 
13        | else
14          |  $i \leftarrow i + 1$ 
15          | Define  $f(C_v) = i$ 
16          |  $N_v \leftarrow i$ 
17 | return  $N_{\mathcal{R}(T_1)} = N_{\mathcal{R}(T_2)}$ 

```

[59]: Knuth (1973), ‘The art of computer programming. Vol. 3, Sorting and Searching’

[60]: Skiena (2012), ‘Sorting and searching’

[42]: Sutherland (1964), ‘Sketchpad a man-machine graphical communication system’

[43]: Hart et al. (1991), ‘Efficient antialiased rendering of 3-D linear fractals’

[45]: Frick et al. (2003), ‘Query evaluation on compressed trees’

[44]: Buneman et al. (2003), ‘Path queries on compressed XML’


Secondly, from a theoretical point of view, our version is slightly less efficient – see upcoming [Proposition 2.3](#). Nevertheless, our version has the merit of being simpler to formalize, but above all it allows to treat both ordered and unordered trees (where AHU only addresses the latter) and can be easily adapted for labeled trees in the case where we impose equality of labels⁹.

The overall idea of the algorithm is to assign to each node v , in a bottom-up approach, a number N_v representing its equivalence class $[v]$ – starting with leaves receiving the number 0. The algorithm returns \top if and only if the roots are assigned the same number, i.e., if and only if $[\mathcal{R}(T_1)] = [\mathcal{R}(T_2)]$ – equivalent to $T_1 \simeq T_2$. We are particularly interested in this feature of assigning to each node its equivalence class, and it will be exploited in upcoming DAG compression, but also in [Chapter 4](#).

Given two trees T_1 and T_2 , assuming that $\#T_1 = \#T_2 = n$ and $\text{deg}(T_1) = \text{deg}(T_2) = d$, we have the following result.

Proposition 2.3 *Algorithm 1 has complexity:*

- ▶ $O(nd \log(d))$ for unordered trees;
- ▶ $O(nd)$ for ordered trees.

Proof. Suppose that determining whether $f(C_v)$ is defined can be made in constant time (e.g. via hash tables [59]). The double **for** loop allows to visit all the nodes of T_1 and T_2 . When visiting a node v , we scan its children and sort them only in the unordered case, for a complexity of $\text{deg}(v)$ or $\text{deg}(v) \log(\text{deg}(v))$ – since merge sort has worst time complexity in $O(k \log(k))$ [60]. Summing over all the nodes and bounding $\text{deg}(v)$ by d yields the expected result. 

2.3 DAG compression of trees

Trees can present internal repetitions in their structures. Eliminating these structural redundancies defines a reduction of the initial data that can result in a DAG. In particular, beginning with [42], DAG representations of trees are also much used in computer graphics where the process of condensing a tree into a graph is called object instancing [43]. One can also use DAG reduction of trees to simplify queries on XML documents [44, 45].

Overview

Before discussing the formal construction of DAG compression, we present here the general idea in the form of a vertex coloring procedure.

Let T be an unordered tree. Each node u of T is assigned a color $c(u)$ such that, for any pair of distinct nodes, $c(u) = c(v) \iff [u] = [v]$. This is equivalent to partitioning the nodes of T according to their equivalence class.

We then construct a directed graph D whose number of vertices is equal to the number of colors used, i.e. $\#D = \#\text{Im } c$, and whose arcs are

constructed as follows. For any two nodes $u, v \in T$, if $u \in C(v)$, we add an arc $c(v) \rightarrow c(u)$ in D . Note that this definition implies that multiple arcs are possible in D , as if there exist $u, u' \in C(v)$, so that $[u] = [u']$, then the arcs $c(v) \rightarrow c(u)$ and $c(v) \rightarrow c(u')$ are identical.

The resulting graph is a DAG, and we call it the *DAG compression* of T . The main advantage of this compression is that it is lossless, i.e. the original tree can be reconstructed identically. If T is ordered, the order in which the arcs $c(v) \rightarrow c(u)$ are placed must respect the order of appearance of u among the children of v . An illustration is provided in Figure 2.7.

Formal definition

We use here a formalism similar to the one that can be found in [61].

Let T be a unlabeled tree. Let $Q(T) = T/\simeq$ be the quotient graph of T by \simeq . Nodes of $Q(T)$ are equivalence classes of nodes of T ; an arc $a \rightarrow b$ exists in $Q(T)$ if and only if there exist $u, v \in T$ such that $[u] = a, [v] = b$, and $v \in C(u)$. Such a graph $Q(T)$ is known to be a DAG [61, Proposition 1].

In fact, in a more general way, we have the following proposition.

Proposition 2.4 *Let \sim be an equivalence relation on \mathcal{T} so that $T_1 \sim T_2 \implies \#T_1 = \#T_2$. Then, the quotient graph $Q(T) = T/\sim$ is a DAG.*

Proof. Assume there exist q_1, \dots, q_n in $Q(T)$ such that there is a cycle $q_1 \rightarrow q_2 \rightarrow \dots \rightarrow q_n \rightarrow q_1$. Since all trees represented by class q_i share the same number of nodes, let us call this number c_i . Since an arc $q_i \rightarrow q_j$ in $Q(T)$ implies that there exist $u, v \in T$ with $[u] = q_i, [v] = q_j, v \in C(u)$, then $c_j < c_i$. Therefore, $c_1 < c_2 < \dots < c_n < c_1$, which is absurd. ✍

Let s be a section of $Q(T)$, i.e. an injective function $s : Q(T) \rightarrow T$ such that for any $q \in Q(T), [s(q)] = q$. For $q \in Q(T)$, let us build the multiset $S(q) = \{[v] : v \in C(s(q))\}$. For any $b \in S(q)$, let $n(b, q)$ be the multiplicity of b in $S(q)$. This number is equal to the number of children with class b of a node with class q in T . As one arc $q \rightarrow b$ already exists in $Q(T)$, we add $n(b, q) - 1$ new arcs $q \rightarrow b$ to match the multiplicity. If T is ordered, the arcs leaving q must respect the order of appearance of children of $s(q)$ in T . This whole operation creates a new DAG¹¹ $\tilde{Q}(T)$, whose construction is illustrated on Figure 2.8. Note that $\tilde{Q}(T)$ is a multigraph.

The definition of parent and children of a node can be generalized to $\tilde{Q}(T)$ and will be denoted the same as for trees, respectively $\mathcal{P}(\cdot)$ and $\mathcal{C}(\cdot)$. Note that since there can be multiple repetitions of an arc in $\tilde{Q}(T)$, the multiplicity is accounted for in $\mathcal{P}(\cdot)$ and $\mathcal{C}(\cdot)$. The notions of height and outdegree also apply and will be denoted $\mathcal{H}(\cdot)$ and $\text{deg}(\cdot)$ as well. Similarly to trees, for a DAG D we denote by $D[v]$ the subDAG rooted in v composed of v and all its descendants in D . Unlike trees, the notion of depth as introduced in Equation 2.2 does not apply to DAGs. Take the example of Figure 2.8: on the tree, there are nodes ● with depth of 1 or 2; on the DAG these nodes are compressed together and therefore their depth is not well defined (it depends on the path one follows in the DAG from the root).

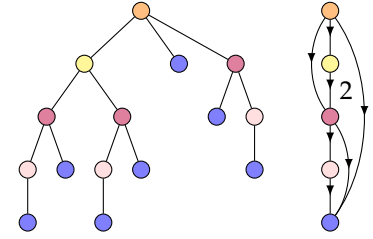


Figure 2.7: An unordered tree (left) and its DAG compression (right). Note that since the order of the arcs does not matter (T being unordered), the arc $\text{yellow} \rightarrow \text{red}$ has only been drawn once, and its multiplicity written aside.

[61]: Godin et al. (2009), ‘Quantifying the degree of self-nestedness of trees: application to the structural analysis of plants’

10: In other words, s associates to each equivalence class a representative.

11: Indeed, since we are duplicating arcs that already exist, we can not create a cycle.

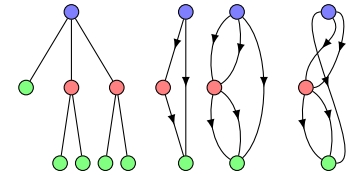


Figure 2.8: From left to right: a tree T , its quotient graph $Q(T)$, the unordered version of $\tilde{Q}(T)$ and the ordered version of $\tilde{Q}(T)$. Nodes are colored according to their class of equivalence.

We call the graph $\tilde{Q}(T)$ the *DAG compression* of T , and denote it by $\mathfrak{R}(T)$. The DAG compression of T can be seen as T deprived of its redundancies, and actually is a lossless compression of T since we have the following theorem.

Theorem 2.5 T can be reconstructed from $\mathfrak{R}(T)$ – up to tree isomorphism.

Algorithm 2: REBUILDTREE

Input: $\mathfrak{R}(T), q \in \mathfrak{R}(T)$

- 1 Let t be a single-node tree
 - 2 **for** $b \in C(q)$ **do**
 - 3 $t_b \leftarrow \text{REBUILDTREE}(\mathfrak{R}(T), b)$
 - 4 Add t_b as last subtree of t
 - 5 **return** t
-

This argument was made in [61, Proposition 4], where the proof was actually omitted but an iterative reconstruction algorithm was proposed instead. We alternatively propose with [Algorithm 2](#) a recursive reconstruction algorithm, for which we prove that it indeed reconstructs the tree as claimed. In the sequel, \mathfrak{R}^{-1} stands for the inverse operator of \mathfrak{R} , i.e. for any tree T , $\mathfrak{R}^{-1}(\mathfrak{R}(T)) \simeq T$ by the previous theorem.

Proof. Let $D = \mathfrak{R}(T)$. Using [Algorithm 2](#), let $\tau = \text{REBUILDTREE}(D, \mathcal{R}(D))$ – where $\mathcal{R}(\cdot)$ denotes the root. We aim to prove by induction that τ and T are equal, up to a tree isomorphism. Trivially, if T is a single node tree, this property holds. Otherwise, let us denote r the root of T – therefore $[r] = \mathcal{R}(D)$ – and ρ the root of τ .

We partition $C(r)$ by equivalence class – each one corresponds to a different node in D . All classes are addressed identically, so let b be one of them, and $C_b = \{u \in C(r) : [u] = b\}$. By definition, $\#C_b = n(b, \mathcal{R}(D))$; therefore, when reconstructing, we add as many subtrees to ρ .

By induction, each of these new subtrees has equivalence class b and is isomorphic to the elements of C_b . Indeed, since the order of arcs in D is not preserved in the unordered case, the topologies of T and τ are equal up to permutation of children – e.g., subtrees \bullet and \circ in [Figure 2.9](#). In the ordered case, as subtrees are inserted in last position of the children of ρ , they respect the same order as the order of the arcs, and therefore the order of children of r : the topologies are equal. 🌿

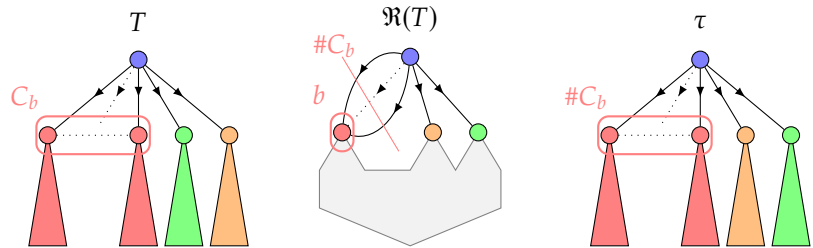


Figure 2.9: From left to right: a tree T , its unordered DAG compression $\mathfrak{R}(T)$ and its reconstruction τ .

Construction

[62]: Downey et al. (1980), ‘Variations on the common subexpression problem’

Practical algorithms to compute $\mathfrak{R}(T)$ exist in the literature, either for unordered trees [61], with complexity $\mathcal{O}(\#T \deg(T) \log(\deg(T)))$, or ordered trees [62], with complexity going from $\mathcal{O}(\#T^2)$ to $\mathcal{O}(\#T)$.

We propose with [Algorithm 3](#) an unifying algorithm for computing $\mathfrak{R}(T)$, whether should T be ordered or not. The determination on the fly of classes of equivalence of the nodes is adapted from AHU algorithm.

The key idea of the algorithm is to build a mapping f that associates to each node its class of equivalence, based on the distribution of the

equivalence classes of its children; starting from the leaves to the root. Assuming that (i) creating a new vertex is of complexity $\mathcal{O}(C_v)$ and (ii) determining whether $f(C_v)$ is defined can be made in constant time (as for [Algorithm 1](#)), the overall complexity of the algorithm is $\mathcal{O}(\#T \deg(T))$ for ordered trees and $\mathcal{O}(\#T \deg(T) \log(\deg(T)))$ for unordered trees – thus comparable to existing literature.

Remark 2.2 As for [Algorithm 1](#), [Algorithm 3](#) can take into account labeled trees (imposing label equality); once more, add the label \bar{v} as the first element of the list C_v constructed in Line 6 – but after the eventual sorting step.

2.4 DAG compression of forests

DAG compression can be extended naturally to forests, which is investigated here. This section is mostly reproduced from [\[39\]](#).

Definition

Let $F = \{T_1, \dots, T_n\}$ be a forest, as defined earlier. We construct a super-tree T_F , placing each T_i as a child of an artificial root. It suffices then to compute $\mathfrak{R}(T_F)$ with [Algorithm 3](#) – whose single root will also be artificial.

Since the root of this DAG D is artificial, we will use the notation $\mathcal{R}(D)$, to designate the children of this root – and thus the nodes of D that really correspond to the roots of the trees of the initial forest.

DAG recompression

Since DAG compression is lossless, one can imagine storing tree data in DAG form. In the case where the forest is stored as a DAG forest (i.e. each tree is compressed and stored individually), it would be superfluous to decompress all the trees to build the super tree T_F and then recompress it. Instead, we would rather build $\mathfrak{R}(T_F)$ directly from this DAG forest.

Let $F = \{T_1, \dots, T_n\}$ be the forest to be compressed, and let $F_D = \{D_1, \dots, D_n\}$ be the associated DAG forest, with $\mathfrak{R}(T_i) = D_i$. We define the degree of the forest as $\deg(F) = \max_i \deg(D_i)$. Computing $\mathfrak{R}(T_F)$ from F_D is in two steps: (i) a construct a super DAG Δ by placing each D_i as a subDAG of an artificial root – with complexity $\mathcal{O}(\deg(F) \sum_i \#D_i)$, and (ii) recompress Δ using [Algorithm 4](#). An illustration step by step of the algorithm is provided in [Figure 2.10](#).

Proposition 2.6 *Algorithm 4 correctly computes $\mathfrak{R}(T_F)$.*

Proof. Starting from the leaves, we examine all vertices of same height in Δ . Those with same children¹² are merged into a single vertex. The algorithm stops when at some height h , we can not find any vertices to be merged. Vertices that are merged in the algorithm represent isomorphic

Algorithm 3: TREECOMPRESSION

Input: T

Output: $\mathfrak{R}(T)$

```

1 Let  $Q$  be the empty graph
2  $i \leftarrow 0$ 
3 Let  $f : \emptyset \mapsto 0$ 
4 for  $h$  from 0 to  $\mathcal{H}(T)$  do
5   for  $v \in T^h$  do
6      $C_v \leftarrow \llbracket [u] : u \in C(v) \rrbracket$ 
7     if  $T$  is unordered then
8       Sort  $C_v$ 
9     if  $f(C_v)$  is defined then
10       $[v] \leftarrow f(C_v)$ 
11    else
12       $i \leftarrow i + 1$ 
13      Define  $f(C_v) = i$ 
14       $[v] \leftarrow i$ 
15      Create a new vertex  $i$  in
16       $Q$  with children  $C_v$ 
16 return  $Q$ 

```

[\[39\]](#): Azaïs et al. (2020), ‘The weight function in the subtree kernel is decisive’

12: The ordered or unordered nature of considered trees is significant at this point to determine whether the children are identical or not.


Algorithm 4: DAGRECOMPRESSION**Input:** Δ **Result:** $\mathfrak{R}(T_F)$

```

1 Construct, within one traversal of  $\Delta$ , the mapping  $h \mapsto \Delta^h$  where  $\Delta^h$ 
  is the set of vertices of  $\Delta$  at height  $h$ 
2 for  $h$  from 0 to  $\mathcal{H}(\Delta) - 1$  do
  /* The only node of height  $\mathcal{H}(\Delta)$  is the artificial root, which is therefore not
  considered. */
3   Let  $\sigma(h) \leftarrow \{f^{-1}(\{S\}) : S \in \text{Im } f, \#f^{-1}(\{S\}) \geq 2\}$  be the set of
  vertices to be merged at height  $h$ , where  $f : v \in \Delta^h \mapsto C(v)$ 
  /* It should be noticed that  $\text{Im } f$  depends on whether we consider ordered or
  unordered trees. Indeed, in the ordered case,  $\text{Im } f$  is the set of all lists of
  children; otherwise,  $\text{Im } f$  is the set of all multisets of children. */
4   if  $\sigma(h) = \emptyset$  then
5     | Exit the for loop
6   else
7     for  $M$  in  $\sigma(h)$  do
8       | Choose one element  $v_M$  in  $M$  to remain in  $\Delta$ 
9       | Denote by  $\delta_M$  the other elements of  $M$ 
10      for  $v$  in  $\Delta$  such that  $\mathcal{H}(v) > h$  do
11        | for  $u$  in  $C(v)$  such that  $\exists M \in \sigma(h), \delta_M \ni u$  do
12          | Delete  $u$  from  $C(v)$ 
13          | Add  $v_M$  in  $C(u)$ 
14      for  $M \in \sigma(h)$  do
15        | Delete  $\delta_M$  from  $\Delta$ 
16 return  $\Delta$ 

```

subtrees, so it suffices to prove that the algorithm stops at the right time. Let h be the first height for which $\sigma(h) = \emptyset$.

Suppose by contradiction that some vertices were to be merged at some height $h' > h$. They represent isomorphic subtrees, so that their respective children should also be merged together, and all of their descendants by induction. As any vertex of height $h'' + 1$ admits at least one child of height h'' , $\sigma(h)$ would not be empty, which is absurd. 

Proposition 2.7 *Algorithm 4 has complexity:*

- ▶ $O(\#\Delta \deg(F)(\log(\deg(F)) + \mathcal{H}(\Delta)))$ for unordered trees;
- ▶ $O(\#\Delta \deg(F) \mathcal{H}(\Delta))$ for ordered trees.

Proof. The proof lies in [Appendix A.1](#). 

Remark 2.3 One might also want to treat online data, but without recompressing the whole data set when adding a single entry in the forest. Let $\mathfrak{R}(T_F)$ be the already recompressed forest and D a new DAG to be introduced in the data (corresponding to a new tree T). It suffices to place D as the rightmost child of the artificial root of $\mathfrak{R}(T_F)$ then run [Algorithm 4](#) to obtain $\mathfrak{R}(T_{F \cup \{T\}})$.

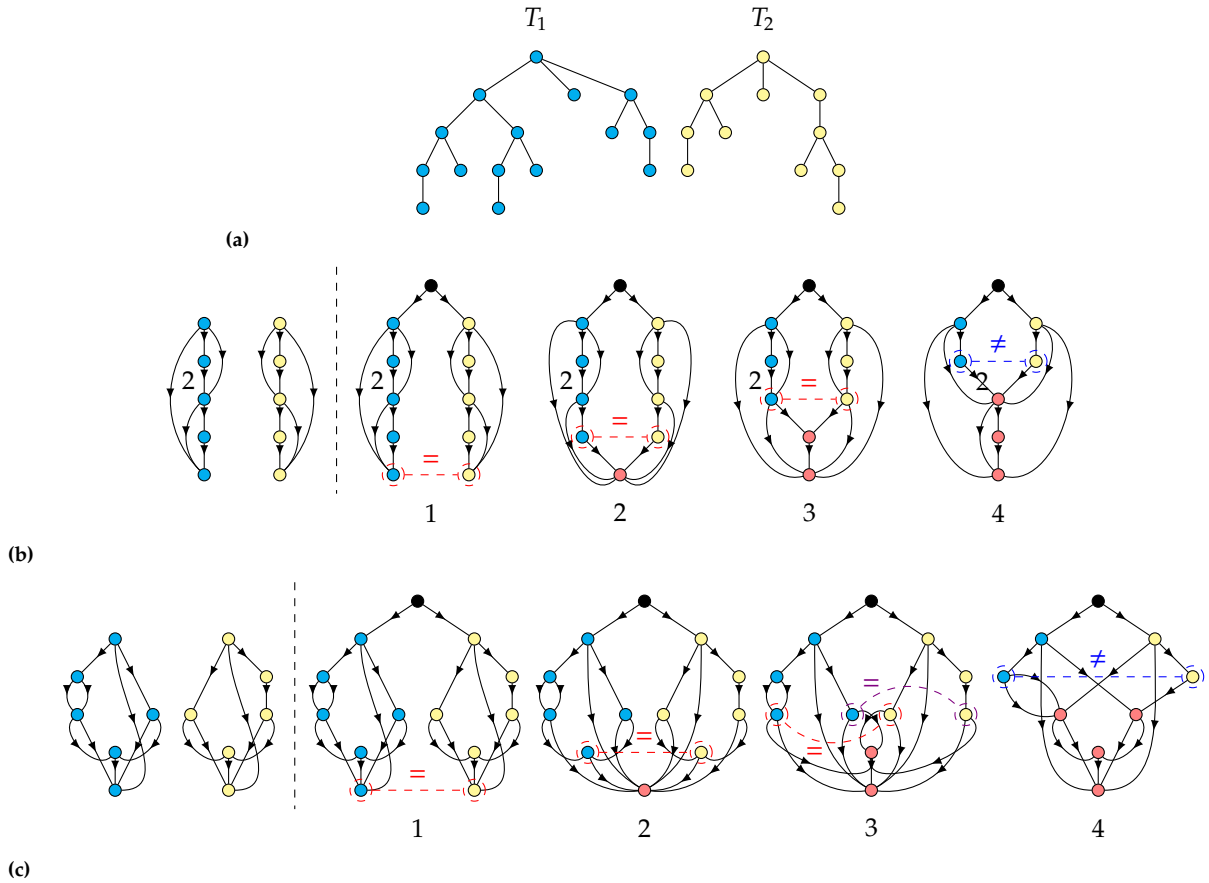


Figure 2.10: An illustration step by step of Algorithm 4 with (a) two trees T_1 and T_2 , seen either as (b) unordered or (c) ordered trees. One can observe their DAG compression (left of the dashed line) and the execution of the algorithm (right of the dashed line). At each step 1, 2 and 3, we examine vertices at height (0,1,2) and merge those which have same children. At step 4, we can not find any vertex to merge and we stop. Note that in (c) at step 3, we find two pairs of vertices to be merged: we are not restricted to one pair per height. Merged vertices are colored in red. The artificial root is colored in black.

Connection between a forest and its compressed form

Origin Whenever a DAG D compresses a forest $F = \{T_1, \dots, T_n\}$, it can be useful to retrieve, for each vertex in D , in which trees they originally appear – say, for instance, that you need to decompress only a single tree. For any vertex $v \in D$, the origin of v is defined as


$$o(v) = \{i \in \llbracket 1, n \rrbracket : \mathfrak{R}^{-1}(D[v]) \in \mathcal{S}(T_i)\}. \quad (2.3)$$

In other words, $o(v)$ represents the set of trees in F for which $\mathfrak{R}^{-1}(D[v])$ is a subtree.

The calculation of $o(\cdot)$ all comes down to how the origins are attributed to the roots. Since we control the order in which the trees are inserted under the artificial root of D , if one wants to keep the original order, it is then sufficient to place them so that the root of T_i is the i -th child of the root – and therefore this node receives origin $\{i\}$.

Once the roots have received their origin, the origin of all other vertices can be recursively calculated on the DAG as follows.

Proposition 2.8 $\forall v \notin \mathcal{R}(D), o(v) = \bigcup_{p \in \mathcal{P}(v)} o(p)$.

Proof. By assumption, the origins of the roots are correct. If $i \in o(v)$, then for all $v' \in \mathcal{D}(v)$, $i \in o(v')$ – as $\mathfrak{R}^{-1}(D[v']) \in \mathcal{S}(\mathfrak{R}^{-1}(D[v]))$. The statement follows by induction. 

Presence By construction, the DAG compression D of a tree T eliminates all repetitions of isomorphic subtrees. It can be useful to know how many copies of each of these subtrees exist in T , without having to decompress the tree.

We define the *presence* of a vertex $v \in D$, denoted by $\pi(v)$, as $\pi(v) = \#\{u \in T : [u] = v\}$. Presence can be recursively computed using the following formula

$$\pi(v) = \begin{cases} 1 & \text{if } v = \mathcal{R}(D), \\ \sum_{p \in \mathcal{P}(v)} n(p, v) \pi(p) & \text{otherwise;} \end{cases} \quad (2.4)$$

where $n(p, v)$ is the number of arcs between vertices p and v .


In the case where D compresses a forest $F = \{T_1, \dots, T_n\}$, we associate to each vertex a *presence vector* $\pi(v)$, so that each component $\pi_i(v)$ corresponds to the presence of v in T_i . The previous formula can be adapted as

$$\pi(v) = \begin{cases} (\mathbb{1}_{i \in o(v)})_{i \in \llbracket 1, n \rrbracket} & \text{if } v \in \mathcal{R}(D), \\ \sum_{p \in \mathcal{P}(v)} n(p, v) \pi(p) & \text{otherwise.} \end{cases} \quad (2.5)$$

Note that if a forest F is reduced to a singleton, i.e. $F = \{T\}$, then we retrieve [Equation 2.4](#).

Proposition 2.9 *Equation 2.4 and Equation 2.5 are correct.*

Proof. It suffices to prove that [Equation 2.5](#) is correct as per the observation. Let $v \in D$. If $v \in \mathcal{R}(D)$, then v represents the root of some tree T_i (possibly several trees if there are repetitions in the forest), and therefore $\pi_i(v) = 1$.

Otherwise, suppose by induction that $\pi_i(p)$ is correct for all $p \in \mathcal{P}(v)$ and any $i \in \llbracket 1, n \rrbracket$. Let us fix $p \in \mathcal{P}(v)$. v appears $n(p, v)$ times as a child of p , so if $\mathfrak{R}^{-1}(D[p])$ appears $\pi_i(p)$ times in T_i , then the number of occurrences of $\mathfrak{R}^{-1}(D[v])$ in T_i as a child of $\mathfrak{R}^{-1}(D[p])$ is $n(p, v) \pi_i(p)$. Summing over all $p \in \mathcal{P}(v)$ leads $\pi_i(v)$ to be correct as well. 

In the light of [Proposition 2.8](#) and [Proposition 2.9](#), the origin and presence of any vertex can be computed in linear time in one exploration of the DAG.

**THE TREE CIPHERING ISOMORPHISM
PROBLEM**

Tree cipherings

3

You know me, I think there ought to be a big old tree right there. And let's give him a friend. Everybody needs a friend.

Bob Ross

This chapter focuses on the case of labeled trees, and in particular on how to define an isomorphism that takes them into account, in a way that is less trivial than simply imposing the equality of labels.

Section 3.1 motivates this question from the point of view of a cipher problem, while making the link with graph isomorphism as defined in Chapter 2.

A more formal description can be found in Section 3.2, which defines in particular the *tree ciphering isomorphism problem*. The practical resolution of this problem will be discussed in upcoming Chapter 4. On the other hand, provided with this new isomorphism, we define in Section 3.3 a new type of DAG compression, that takes into account the labels.

Part of this chapter is reproduced from [26], but has been largely rewritten. In particular Section 3.3 is new material.

- 3.1 Motivation 23
 - A tale of ciphers 23
 - Connection with the graph isomorphism problem 25
- 3.2 Formal definition 25
 - On the number of tree isomorphisms 25
 - Tree ciphering isomorphism 26
- 3.3 A new kind of DAG compression 28
 - Intuition 28
 - Definition 29
 - Construction 32

[26]: Ingels et al. (2021), 'Isomorphic Unordered Labeled Trees up to Substitution Ciphering'

3.1 Motivation

A tale of ciphers

Substitution ciphers *Substitution ciphers* are one of the oldest encryption methods. It consists in replacing each letter of a sequence by a given sign (possibly another letter). A *simple* substitution cipher imposes that the mapping between the letters and their substituted signs is a bijection. Formally, given two alphabets \mathcal{A} and \mathcal{B} , a simple substitution cipher is a bijection $f : \mathcal{A} \rightarrow \mathcal{B}$. Given a message $w = a_1 \cdots a_n$, with $a_i \in \mathcal{A}$, the encrypted message is given by $f(w) = f(a_1) \cdots f(a_n)$.

A famous example of a simple substitution cipher is Caesar's code, where $\mathcal{A} = \mathcal{B}$, and where f is a cyclic permutation – i.e., the letters are simply shifted a few ranks in the alphabet. Another well-known example is the pigpen cipher, which places the letters of the alphabet on a grid, and replaces them by the local topology of the grid where they are placed – see Figure 3.1.

Simple substitution ciphers are easily broken, e.g. by frequency analysis [63]. Typically, the letter E is the most frequent in English, followed by T, etc [64]. The recurrence of letters in the encoded message can be analyzed to make guesses. Using statistics, we can measure the adequation between the observed frequency distribution with the expected one (when the source language is known), e.g. via χ^2 tests [65].

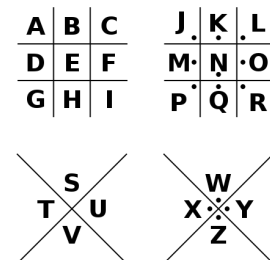


Figure 3.1: Pigpen cipher, ©Wikipedia Commons. For instance, the letter V is replaced by \wedge , U by $<$, and so on.

[63]: Gardner (1984), *Codes, ciphers and secret writing*

[64]: Mayzner et al. (1965), 'Tables of single-letter and digram frequency counts for various word-length and letter-position combinations.'

[65]: Savarese et al. (1999), 'The Caesar Cipher'

In the sequel, we consider only simple substitution ciphers, shortened to ciphers.

A silly ciphering problem Assume one has at one's disposal two messages of the same length, and one want to determine if there exists a cipher that transforms one message onto the other. This question is trivial, as the cipher is induced by the order of letters. Indeed, following the reading order, one can build the cipher letter by letter, until either (i) one gets to the end of the message, and answer Yes, or (ii) one maps two letters that break the bijection of the cipher, and the answer is No. This process actually defines an equivalence relation on messages of the same length, where two messages are equivalent if and only if the answer to the previous question is Yes. See Figure 3.2 for an illustration.

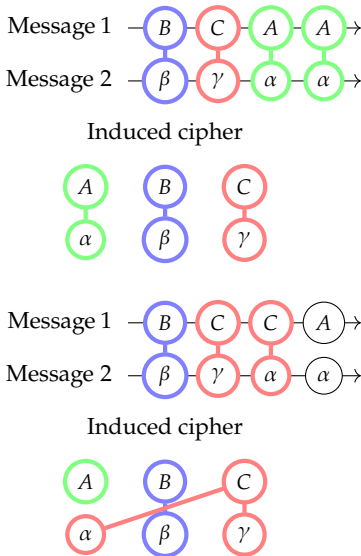


Figure 3.2: Cipher induced by the order of letters on two examples, one where the two messages are equivalent (top), and one where there are not (bottom). In the latter, the message is not parsed in its entirety since an inconsistency is detected before completion.

Same one, but with trees We are actually interested in the declination of this toy problem in the case where messages are carried, not as an ordered sequence of letters, but as labels on nodes of trees. Rather than requiring that the messages are of the same length, we impose that the trees are isomorphic. Previously, the order of the letters provided the mapping of the labels; here, this role is fulfilled by tree isomorphisms, that are usually not unique. Actually, the number of isomorphisms between two trees is given by a product of factorials (see upcoming Equation 3.1 and illustrative Figure 3.4) and thus usually extremely large.

The *tree ciphering isomorphism problem* can then be stated as follows: given two isomorphic labeled trees, is there any tree isomorphism between them that induces a cipher of the labels? This also defines an equivalence relation on labeled trees, where two trees are equivalent if and only if the answer to the previous question is Yes (see upcoming Theorem 3.1). The problem is formally defined in the next section, while an example is now provided in Figure 3.3.

Remark 3.1 The tree ciphering isomorphism problem is only really interesting to solve for unordered trees, since the problem for ordered trees is strictly equivalent to the one of messages treated before, as any tree traversal converts the tree unambiguously into a sequence, and vice-versa – as seen in Encoding of trees (p. 11).

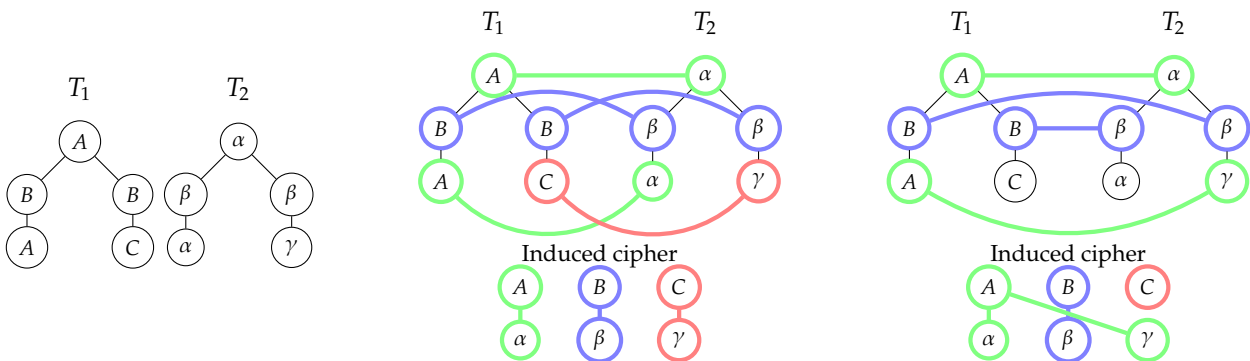


Figure 3.3: Two topologically isomorphic unordered labeled trees T_1 and T_2 (left). There exist two tree isomorphisms between T_1 and T_2 , one inducing a cipher (middle) and the other one that does not (right). In the latter, the full tree isomorphism is not parsed as an inconsistency is detected before. Overall, the two labeled trees T_1 and T_2 are equivalent since at least one tree isomorphism leads to a cipher.

Connection with the graph isomorphism problem

If the problem about sequences was trivial, and if determining whether two trees are *topologically* isomorphic can be achieved within linear time via AHU algorithm – see Section 2.2, determining if two unordered labeled trees are isomorphic under the previous definition is, on the other hand, a difficult problem. It is an instance of labeled graph isomorphism [23, 66] that was introduced under the name *marked tree isomorphism* in [24], where it has been proved *graph isomorphism complete*, i.e. as hard as graph isomorphism. The latter is still an open problem, where no proof of NP-completeness nor polynomial algorithm is known [67].

One classic family of algorithms trying to achieve graph isomorphism are *color refinement algorithms*, also known as Weisfeiler-Leman algorithms [68]. Both graphs are colored according to some rules, and the color histograms are compared afterwards: if they diverge, the graphs are not isomorphic. However, this test is incomplete in the sense that there exist non-isomorphic graphs that are not distinguished by the coloring. The distinguishability of those algorithms is constantly improved – see [69] for recent results – but does not yet answer the problem for any graph. Actually, AHU algorithm for topological tree isomorphism can be interpreted as a color refinement algorithm.

Most methods from the literature – such as color refinement – do not explicitly construct the isomorphism between the two graphs; rather, they relabel a graph so that two isomorphic graphs have the same relabelling. This option is preferred since it allows to process graph datasets, either to eliminate isomorphic elements or to identify a graph in a database. For more details on the practical resolution of graph isomorphism problems, we refer the reader to [25].

As discussed later in this chapter, in order to build a new type of DAG compression, taking into account the labels, we need to know explicitly the isomorphism between the two trees. For this reason, the method we have developed to address the (unordered) tree ciphering isomorphism problem, detailed in Chapter 4, departs from the literature and proposes an explicit construction instead of a relabelling.

3.2 Formal definition

The *tree ciphering isomorphism problem*, as motivated in the previous section, is formally introduced in this section. Nevertheless, we first present results on the number of isomorphisms between two trees.

On the number of tree isomorphisms

Ordered trees Given two ordered trees T_1 and T_2 , if $T_1 \simeq T_2$, then $\# \text{Isom}(T_1, T_2) = 1$. Indeed, let ϕ and ψ be two elements of $\text{Isom}(T_1, T_2)$. Denoting r_1 (respectively r_2) the root of T_1 (respectively T_2) and c_1, \dots, c_k the children of r_1 in this order (respectively s_1, \dots, s_k); we have $\phi(r_1) = r_2 = \psi(r_1)$ and $\phi(c_i) = s_i = \psi(c_i)$. By induction, this applies to all nodes of T_1 and T_2 and therefore $\phi = \psi$.

[66]: Zemlyachenko et al. (1985), ‘Graph isomorphism problem’

[23]: Champin et al. (2003), ‘Measuring the similarity of labeled graphs’

[24]: Booth et al. (1979), *Problems polynomially equivalent to graph isomorphism*

[67]: Schönig (1987), ‘Graph isomorphism is in the low hierarchy’

[68]: Weisfeiler et al. (1968), ‘The reduction of a graph to canonical form and the algebra which appears therein’

[69]: Grohe et al. (2021), ‘Deep Weisfeiler Leman’

[25]: McKay et al. (2014), ‘Practical graph isomorphism, II’

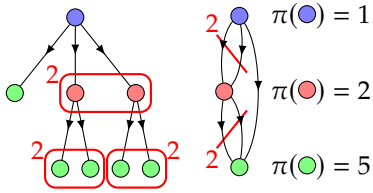


Figure 3.4: An unordered tree T (left) and its DAG compression $\mathfrak{R}(T)$ (right). The method for computing $n_{\simeq}(T)$ is highlighted in both cases: for T , nodes susceptible to be swapped are boxed together; for $\mathfrak{R}(T)$, multiplicity of arcs and presence are indicated. Eitherway, $n_{\simeq}(T) = (2!)^3 = 8$.

Unordered trees In the unordered case, any tree isomorphism $\phi : T_1 \rightarrow T_2$ maps $u \in T_1$ onto $v = \phi(u) \in T_2$ only if $[u] = [v]$ and $\phi(\mathcal{P}(u)) = \mathcal{P}(v)$. Thus, any tree isomorphic to T_1 can be obtained by swapping nodes (i) of same equivalence class and (ii) children of same node. Consequently, the number of tree isomorphisms between T_1 and T_2 depends only on the class of equivalence of T_1 (equivalently T_2), and will be denoted by $n_{\simeq}(T_1)$. For any unordered tree T , we have

$$n_{\simeq}(T) = \prod_{u \in T} \prod_{q \in \{[v] : v \in C(u)\}} (\#\{v \in C(u) : [v] = q\})!. \quad (3.1)$$

Actually, since the formula of $n_{\simeq}(T)$ implies classes of equivalence, it is natural to propose an alternative definition, computed from the DAG compression $\mathfrak{R}(T)$ of T . We have

$$n_{\simeq}(T) = \prod_{v \in \mathfrak{R}(T)} \prod_{u \in C_{\neq}(v)} (n(v, u)!)^{\pi(v)}. \quad (3.2)$$

where $n(v, u)$ is the number of arcs between vertices v and u , $\pi(\cdot)$ is the presence defined in Equation 2.4, and $C_{\neq}(v)$ is the set of *distinct* children of v , i.e. without taking into account the multiplicity.

An example is provided in Figure 3.4.

Tree ciphering isomorphism

From now on, we assume that trees are labeled, ordered or unordered. Recall that the label of node u is denoted by \bar{u} , and that we denote by $\mathcal{A}(T)$ the set of labels – the *alphabet* – of a tree T .

Let T_1 and T_2 be two topologically isomorphic labeled trees and $\phi \in \text{Isom}(T_1, T_2)$. ϕ naturally induces a binary relation R_{ϕ} over sets $\mathcal{A}(T_1)$ and $\mathcal{A}(T_2)$, defined as

$$\forall x \in \mathcal{A}(T_1), \forall y \in \mathcal{A}(T_2), \\ x R_{\phi} y \iff \exists u \in T_1, (x = \bar{u}) \wedge (y = \overline{\phi(u)}). \quad (3.3)$$

Figure 3.5 illustrates this induced binary relation on an example.

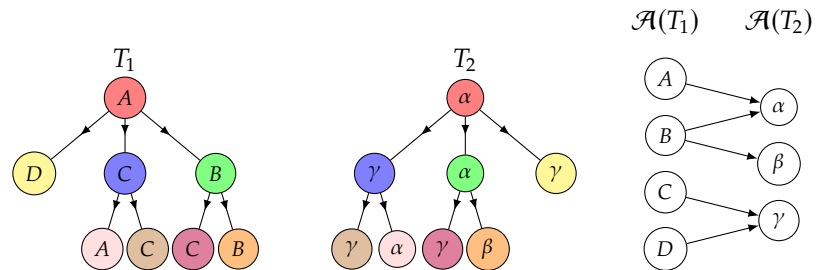


Figure 3.5: Two topologically isomorphic unordered labeled trees (left) and the induced binary relation (right). The tree isomorphism ϕ is displayed through node colors.

Such a relation R_{ϕ} is said to be a bijection if and only if for any $x \in \mathcal{A}(T_1)$, there exists a unique $y \in \mathcal{A}(T_2)$ so that $x R_{\phi} y$, and conversely if for any $y \in \mathcal{A}(T_2)$, there exists a unique $x \in \mathcal{A}(T_1)$ so that $x R_{\phi} y$. This is not the case of the relation induced by the example in Figure 3.5, since C and D are both in relation to γ , and also B is in relation to both α and β .


Whenever R_ϕ is a bijection, we define the function $f_\phi : \mathcal{A}(T_1) \rightarrow \mathcal{A}(T_2)$ by $f_\phi(x) = y \iff x R_\phi y$. This bijective function f_ϕ is called a *cipher*¹³ and verifies

$$\forall u \in T_1, f_\phi(\bar{u}) = \overline{f_\phi(u)}. \quad (3.4)$$

Definition 3.1 $\phi \in \text{Isom}(T_1, T_2)$ is said to be a tree ciphering if and only if R_ϕ is a bijection; in which case we denote $T_1 \xrightarrow{\phi} T_2$.

Let us denote by $\text{Cipher}(T_1, T_2)$ the set of tree cipherings between T_1 and T_2 . If $\text{Cipher}(T_1, T_2)$ is not empty, then we write $T_1 \sim T_2$ and say that T_1 and T_2 are *isomorphic up to a cipher*, since the following result applies.

Theorem 3.1 \sim is an equivalence relation over the set of labeled trees.

Proof. The proof can be found in [Appendix A.2](#). 

Remark 3.2 We just defined with \sim a generalisation of tree isomorphisms. Indeed, it suffices to add the same arbitrary, artificial, label to all nodes of considered trees, and then, it is equivalent for two trees to be (classically) isomorphic or isomorphic up to cipher (the cipher being the identity function in this case).

Remark 3.3 It is possible to be more restrictive on the choices of ciphers. Let (G, \circ) be a subgroup of the bijections between $\mathcal{A}(T_1)$ and $\mathcal{A}(T_2)$. Then, if we replace “ R_ϕ is a bijection” in [Definition 3.1](#) by “ $R_\phi \in G$ ”, the induced relation \sim_G is also an equivalence relation.

For instance, with $G = \{\text{id}\}$, $T_1 \sim_G T_2$ means $T_1 \simeq T_2$ together with the equality of labels. This (very) restricted definition has already been used to adapt DAG compression to labeled trees in [Section 2.3](#).

The *tree ciphering isomorphism problem* is to determine, given two labeled isomorphic trees, whether or not there exists a tree ciphering between them. The purpose of [Chapter 4](#) is precisely to build, algorithmically, a ciphering when it exists. We already mentioned that the tree ciphering isomorphism problem is trivial to solve on ordered trees. Indeed, since there is only one tree isomorphism between two ordered trees, either it is also a cipher, or not – there is no algorithmic problem to solve. On the other hand, due to [Equation 3.1](#), the number of isomorphisms between two unordered trees can be exponentially large, which makes the task of finding a cipher difficult, as investigated in [Chapter 4](#).

The sequel of this chapter is dedicated to the construction of a DAG compression based on this new equivalence relation. First, however, we introduce two additional results.

Whenever T_1 and T_2 are isomorphic up to a cipher, then so are their matching subtrees; more precisely, we have the following result.

¹³: Following the analogy from [A tale of ciphers](#) (p. 23).

Proposition 3.2 If $T_1 \sim T_2$ and $\phi \in \text{Cipher}(T_1, T_2)$, then

$$\forall u \in T_1, T_1[u] \sim T_2[\phi(u)].$$

Proof. The proof is deferred to [Appendix A.2](#). 🍃

In addition, if T_1 and T_2 are isomorphic up to a cipher, and if there also exist two subtrees in T_1 isomorphic up to a cipher, then so are their counterparts in T_2 , as per the next proposition.

Proposition 3.3 Let $T_1 \sim T_2$ and $\phi \in \text{Cipher}(T_1, T_2)$; if there exist $u, v \in T_1$ such that $T_1[u] \sim T_1[v]$, then $T_2[\phi(u)] \sim T_2[\phi(v)]$.

Proof. The proof is deferred to [Appendix A.2](#). 🍃

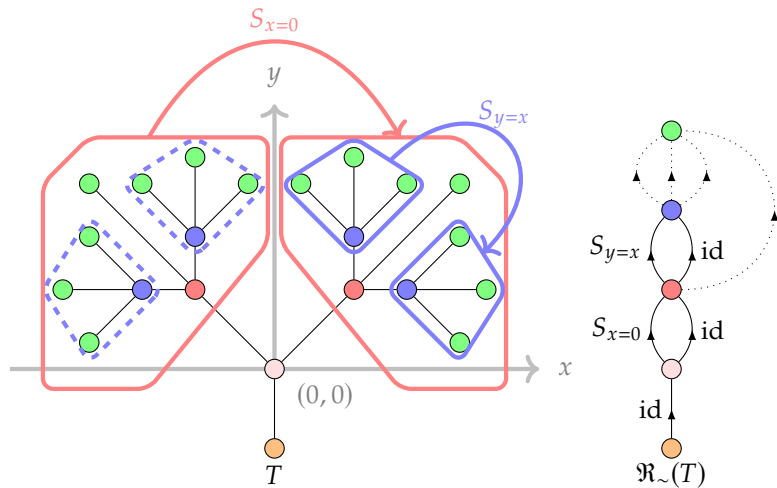
3.3 A new kind of DAG compression

As seen in [Section 2.3](#), a tree can be represented more concisely by eliminating its structural redundancies. This DAG compression is derived from tree isomorphisms introduced in [Section 2.2](#). We already mentioned the possibility of extending DAG compression to labeled trees, only if we impose label equality. Since we now have a notion of isomorphism allowing to take into account labels in a less trivial way, we propose in this section to build a new DAG compression, based on isomorphisms up to a cipher.

Intuition

Before more formal considerations, we propose here to introduce the concept of DAG compression with labels via a geometrical intuition. Consider the tree of [Figure 3.6](#), where the label on each node is the (x, y) coordinates of the node in the drawing.

Figure 3.6: A labeled tree T (left), where the labels are the (x, y) coordinates of each node; and the DAG compression $\mathfrak{R}_{\sim}(T)$ of T (right), taking into account the labels. Nodes are colored according to their equivalence class (with respect to \sim). Although not shown on the DAG, note that each node in the DAG carries a label, which are here the coordinates of the node in T chosen to represent its equivalence class. Since the transformations between leaves have not been made explicit, the arcs leading to them do not carry any information – and are not complete with respect to the DAG construction.



We can observe a number of symmetries in this tree: the red subtrees are obtained by symmetry around the y -axis, and the blue subtrees by symmetry around the axes $y = -x$ and $y = x$. In the context of ciphers, red subtrees are isomorphic, and blue ones too (so are leaves, trivially, since $\bar{u} \mapsto \bar{v}$ is always bijective).

Remark 3.4 This example also gives us an illustration of [Proposition 3.2](#) and [Proposition 3.3](#), where T_1 and T_2 are the red subtrees:

- ▶ The dashed blue subtree at the bottom left is the symmetric along the y -axis of the blue subtree at the bottom right, and they are isomorphic.
- ▶ The two dashed blue subtrees on the left are isomorphic, and so are their counterparts on the right.

Since the left red subtree is obtained by symmetry from the right red subtree, if we keep only the geometry of the right tree and the axis of symmetry, we can perfectly reconstruct the original tree. Recursively, since the top left blue subtree is obtained by symmetry from the bottom left blue subtree, retaining only the geometry of the second and the axis of symmetry also allows the reconstruction of both. Same thing for the leaves, even if the transformations are not explicit on the drawing.

As we do this, we are in fact building a DAG – also shown in [Figure 3.6](#). When we have several subtrees isomorphic up to a cipher (symmetries in our example), we retain the labels of only one (it is the representative of its equivalence class) and all the ciphers allowing to reconstruct the other elements of its equivalence class. We place those ciphers on the arcs leading to the vertex in the DAG compressing the chosen subtree, and depending on which path we follow in the DAG, we reconstruct different subtrees by composing the ciphers.

Remark 3.5 Note that one of the arcs always bears the identity cipher, which allows to reconstruct the subtree chosen as representative of its equivalence class.


For example, if we want to reconstruct the top left blue subtree of [Figure 3.6](#), we start by following the arc bearing $S_{x=0}$ (which allows us to reconstruct the left red subtree from the right one) and then the arc bearing $S_{y=x}$ (which allows us to reconstruct the top blue subtree from the bottom one).

Definition

Let T be a labeled tree, and $v \in T$. We denote by $\llbracket v \rrbracket$ the class of equivalence of v for the \sim equivalence relation, i.e. the set of all nodes $v \in T$ such that $T[u] \sim T[v]$. As in [Section 2.3](#), we define $Q(T) = T/\sim$ the quotient graph of T by \sim , which is a DAG in virtue of [Proposition 2.4](#). As for \simeq , we define a section s as an injective function $s : Q(T) \rightarrow T$ so that $\llbracket s(q) \rrbracket = q$. However, we want to find one that respects the tree structure of T .

Lemma 3.4 *There exists a section s of $Q(T)$ that verifies*

$$\forall q \in Q(T), \exists b \in \mathcal{P}(q) : s(q) \in C(s(b)). \quad (3.5)$$

Proof. Such a section can be constructed in a top-down approach. As the root $r = \mathcal{R}(T)$ is unique, $s(\llbracket r \rrbracket) = r$. Then, for any child of $\llbracket r \rrbracket$ in $Q(T)$, we know that there exists a corresponding child in $C(r)$. We can then construct s recursively by choosing among such candidates. 

To illustrate the previous proof, let us consider the example of [Figure 3.7](#). Starting from the root, $s(\bullet) = u_1$. For $s(\circ)$, we can choose between u_3 and u_4 as they are both children of u_1 . Let's say $s(\circ) = u_4$. In this case, u_5 and u_6 can no longer be chosen as representative of class \circ , for which we can choose between u_2 , u_7 and u_8 – let's say $s(\odot) = u_2$.

In the sequel, s is assumed to verify [Equation 3.5](#).

Similarly as in [Section 2.3](#), we build a new DAG $\tilde{Q}(T)$ from $Q(T)$. First we add to each vertex $q \in Q(T)$ the label $\overline{s(q)}$. Then, we add arcs exactly as for \simeq . We consider the multiset $S(q) = \{\llbracket v \rrbracket : v \in C(s(q))\}$, and denote $n(b, q)$ the multiplicity of b in $S(q)$. One arc $q \rightarrow b$ already exists in $Q(T)$, so we add $n(b, q) - 1$ news arcs $q \rightarrow b$ to match the multiplicity. If T is ordered, the arcs $q \rightarrow b$ must respect the order of apparition of children of $s(q)$ in T .

Let $u \in C(s(q))$, with $\llbracket u \rrbracket = b$. There exists $\phi \in \text{Cipher}(T[s(b)], T[u])$, by definition, so that $T[s(b)] \xrightarrow{\phi} T[u]$. For each such u – there are as many as $n(b, q)$ – we annotate the corresponding arc $q \rightarrow b$ with the cipher f_ϕ – as defined in [Equation 3.4](#). Note that we lose the dependency on ϕ since it is implicit in the graph of f_ϕ . Note also that – as claimed in [Remark 3.5](#) – for each vertex, there is always at least one entering arc annotated with id , as $T[s(b)] \xrightarrow{\text{id}} T[s(b)]$.

Remark 3.6 It is essential to note that the arcs annotated with the identity do not bear the *same* identity. On the example of [Figure 3.7](#), one of the identities represents the tree ciphering of $T[u_2]$ on itself, and thus corresponds to $\overline{u_2} \mapsto \overline{u_2}$; while the other identity represents the tree ciphering of $T[u_4]$ on itself, and thus corresponds to $\overline{u_4} \mapsto \overline{u_4}$; $\overline{u_7} \mapsto \overline{u_7}$ and $\overline{u_8} \mapsto \overline{u_8}$. Nevertheless, for the sake of simplicity, we note indifferently id in the sequel, regardless of the actual domain of definition.

Following all these steps, we get a new graph $\tilde{Q}(T)$ whose construction is illustrated in [Figure 3.7](#). We still call this graph the *DAG compression* of T but denote it $\mathfrak{R}_\sim(T)$. Note that this DAG is not unique: when we have different possible choices of representatives for a vertex, each choice leads to a DAG whose arc and vertices will carry different information. This is not a problem, however, since whatever choices are made, this compression is lossless and the original tree can be reconstructed.

Theorem 3.5 *T can be reconstructed from $\mathfrak{R}_\sim(T)$ – up to a tree isomorphism.*

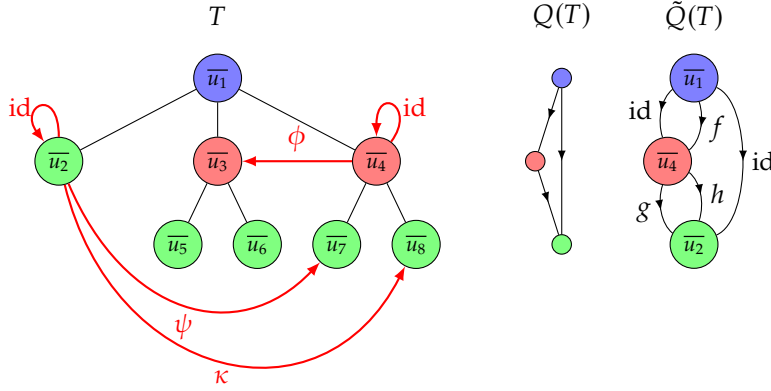


Figure 3.7: From left to right: a tree T , its quotient graph $Q(T)$ and the unordered graph $\tilde{Q}(T)$. Nodes are colored following their class of equivalence (with arbitrary labels, and with respect to \sim). The tree cipherings between observed nodes are displayed in red. We denote by $f = f_\phi$, $g = f_\psi$ and $h = f_\kappa$ the cipherings associated to each tree ciphering.

Note that we are reconstructing the tree up to a *tree isomorphism*, not a *tree ciphering*. In other words, the labels are reconstructed *exactly* (and not up to a cipher). Since tree cipherings requires topological isomorphism, we adapt [Algorithm 2](#), adding new instructions to deal with labels, and propose [Algorithm 5](#). The difference with the previous algorithm is only in the assignment of label values, therefore the topology is correctly reconstructed (up to a tree isomorphism) by virtue of [Theorem 2.5](#).

Algorithm 5: REBUILDTREEWITHCIPHER

Input: $\mathfrak{R}_\sim(T)$, $q \in \mathfrak{R}_\sim(T)$, f

- 1 Let t be a single-node tree with label $f(\bar{q})$
 - 2 **for** $b \in C(q)$ **do**
 - 3 Let g be the cipher on arc $q \rightarrow b$
 - 4 Let $t_b = \text{REBUILDTREEWITHCIPHER}(\mathfrak{R}_\sim(T), b, f \circ g)$
 /* By construction of $\mathfrak{R}_\sim(T)$, the image of g is a subset of the domain of f , so that
 $f \circ g$ is well defined. */
 - 5 Add t_b as last subtree of t
 - 6 **return** t
-

Proof. Let $D = \mathfrak{R}_\sim(T)$ and $\tau = \text{REBUILDTREEWITHCIPHER}(D, \mathcal{R}(D), \text{id})$. Since the topologies of T and τ are identical (up to a tree isomorphism), it suffices to show that the labels of T are correctly reproduced in τ .

We proceed by induction on the subtrees chosen to represent their equivalence class. The base case concerns leaves. One of the leaves $l \in \mathcal{L}(T)$ has been chosen to represent the class of leaves¹⁴ in D . At least one of the arcs leading to $\llbracket l \rrbracket$ in the DAG bears the cipher id , by construction (as the representative of a class is isomorphic to itself). The parent of $\llbracket l \rrbracket$ by this arc admits itself an entering arc bearing id , and so on up to the root of D – as stated in [Remark 3.5](#). Therefore, during the reconstruction, one of the paths passes only through arcs bearing id , and thus one of the leaves of τ admits the label \bar{l} . The base case is thus proven: the subtree chosen to represent its class (here the leaves) is well labeled in the reconstruction.


We denote by r and ρ the roots of T and τ , respectively. We partition $C(r)$ by equivalence class. All classes are addressed identically, so let b be one of them, and $C_b = \{u \in C(r) : \llbracket u \rrbracket = b\}$. By definition, $\#C_b = n(b, \mathcal{R}(D))$; therefore, when reconstructing, we add as many subtrees to ρ . We know that these subtrees C'_b are isomorphic (with respect to \simeq) to those of C_b

14: All leaves are isomorphic since $\bar{u} \mapsto \bar{v}$ is necessarily a bijection.

as the algorithm preserves the topology; we now show that they are correctly labeled.

One of the elements of C_b , $s(b)$, has been chosen as the representative of b . Since the elements of C_b are isomorphic up to a cipher, for any $u \in C_b$, there exists $\phi \in \text{Cipher}(T[s(b)], T[u])$ so that $T[s(b)] \xrightarrow{\phi} T[u]$ – whose associated cipher f_ϕ was placed on one arc between $\mathcal{R}(D)$ and b in the DAG.

By induction, we know that the subtree of C'_b reconstructed from $s(b)$ has been correctly labeled (following the arc bearing id). Therefore, the counterpart of u in C'_b , reconstructed by following the arc carrying f_ϕ , is also correctly labelled since $T[s(b)] \xrightarrow{\phi} T[u]$ – just traverse the reconstructed subtree from $s(b)$, and apply f_ϕ on each label.

Since the roots are the only nodes in their equivalence class, it follows that T and τ are identically labeled. 

Construction

To be isomorphic up to a cipher, two trees must already be isomorphic. In particular, two nodes with the same equivalence class with respect to \sim must already have the same equivalence class with respect to \simeq . Thus, to compute equivalence classes $[[\cdot]]$, one must also know the equivalence classes $[\cdot]$. We therefore propose to construct $\mathcal{R}_{\sim}(T)$ directly from $\mathcal{R}(T)$, which precisely has already associated to each node v of the tree T its equivalence class $[v]$. Since the topology is already taken into account, all that remains is to consider the labels, which is obtained by virtue of [Algorithm 6](#).

Principle The main idea of the algorithm is, in a top-down approach, to visit each node q of $\mathcal{R}(T)$ and to partition the nodes $T(q) = \{u \in T : [u] = q\}$ according to their equivalence class with respect to \sim . We construct a partition \mathfrak{P} , such that for $P \in \mathfrak{P}$, all elements of P have the same equivalence class. One representative per class is arbitrarily chosen, denoted by $s(P)$. Note that in line 8, when we check if the current node $u \in T(q)$ belongs to an already identified class, by testing whether $u \sim s(P)$, we have not yet introduced the algorithm for doing this verification. The purpose of [Chapter 4](#) is to build this algorithm; we assume for the moment its existence. Also, for nodes u not chosen to represent their class, we must remove $\mathcal{D}(u)$ from the sets $T(q')$ in the rest of the DAG. This ensures that the condition of [Equation 3.5](#) is satisfied, and thus that we can reconstruct T by virtue of [Theorem 3.5](#).

Then, in a second step, for each element P of the partition – thus for each identified equivalence class, we add a new node p in the DAG, all of which collectively replace the previous node q . Each new node p receives the label of $s(P)$. Then, for each node $u \in P$, we know by hypothesis that $u \sim s(P)$, and thus that there exists $\phi \in \text{Cipher}(T[u], T[s(P)])$ – we assume that the upcoming algorithm from [Chapter 4](#) provides such a ϕ . Since we go through the vertices of $\mathcal{R}(T)$ from top to bottom, we have already created the vertex corresponding to $[[\mathcal{P}(u)]]$ and we can therefore create an arc from it to p , bearing the cipher f_ϕ .

Algorithm 6: COMPRESSIONWITHLABELS**Input:** $D = \mathfrak{R}(T), T$ **Output:** $\mathfrak{R}_{\sim}(T)$

```

1 Construct, within one traversal of  $T$ , the mapping  $q \in D \mapsto T(q)$ ,
  where  $T(q) = \{u \in T : [u] = q\}$ 
  /* Remember that  $[ \cdot ]$  stands for the equivalence class with respect to  $\approx$ . */
2 for  $h$  from  $\mathcal{H}(D)$  to 0 do
3   for  $q \in D^h$  do
4      $\mathfrak{P} \leftarrow \{\emptyset\}$ 
5     /* We partition in  $\mathfrak{P}$  the nodes of  $T(q)$  by equivalence class. */
6     Let  $s : \emptyset \mapsto \emptyset$ 
7      $N \leftarrow \emptyset$ 
8     for  $u \in T(q)$  do
9       if  $\exists P \in \mathfrak{P} : u \sim s(P)$  then
10        Add  $u$  to  $P$ 
11         $N \leftarrow N \cup \mathcal{D}(u)$ 
12        /* By virtue of Equation 3.5, we store  $\mathcal{D}(u)$  to remove it later from
13          $T(q')$ , with  $q' \in \mathcal{D}(q)$ . */
14      else
15         $P \leftarrow \{u\}$ 
16        Define  $s(P) = u$ 
17        Add  $P$  to  $\mathfrak{P}$ 
18    for  $q' \in \mathcal{D}(q)$  do
19       $T(q') \leftarrow T(q') \setminus N$ 
20    for  $P \in \mathfrak{P}$  do
21      Create a new vertex  $p$  in  $D$  with no child
22       $\bar{p} \leftarrow s(P)$ 
23      Define  $\llbracket s(P) \rrbracket = p$ 
24      for  $u \in P$  do
25        There exists  $\phi$  so that  $T[s(P)] \xrightarrow{\phi} T[u]$ 
26        Add an arc from  $\llbracket \mathcal{P}(u) \rrbracket$  to  $p$ , bearing cipher  $f_\phi$ 
27        /* In the ordered case, take into account the position of  $u$  among
28         children of  $\mathcal{P}(u)$  when adding the new arc. */
29    Delete vertex  $q$  from  $D$ 
30 return  $D$ 

```

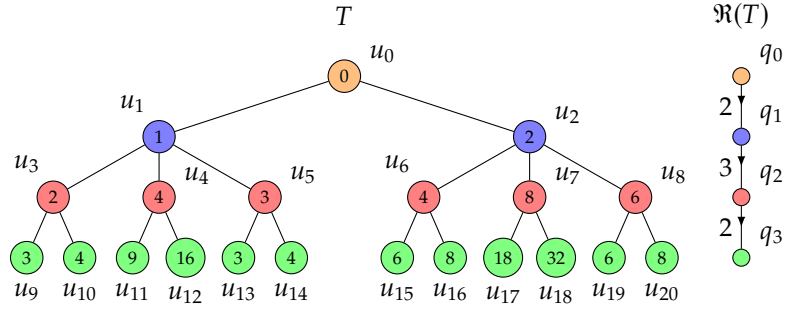
Remark 3.7 Note that the pseudocode provided does not associate to each node $u \in T_1$ its equivalence class $\llbracket u \rrbracket$. We only need it for the representative $s(P)$ of each element P of the partition \mathfrak{P} to be able to recover, later, the equivalence class of the parent of the remaining nodes. This operation is related to Equation 3.5.

Remark 3.8 When we place the ciphers on the arcs (line 22), the information we have to store is proportional to the number of different labels observed on the considered subtree. In particular, one of the ciphers is the identity.

In light of Remark 3.6, one can see the identity as an abstract function that returns whatever argument is given to it, and that, when composed with another function, returns that function. Such an approach reduces

the amount of information to be stored on the arcs.

Figure 3.8: An unordered labeled tree T (left), and its (classical) DAG compression $\mathfrak{R}(T)$ (right). Nodes with same equivalence class (with respect to \simeq) are colored accordingly.



Example of execution We now illustrate the execution of the algorithm on an example, namely the tree T of Figure 3.8. Let us first note that

- 15: Implied, restricted in its domain to the numbers observed in T .
- 16: Idem.

- ▶ $T[u_1] \sim T[u_2]$, where the cipher is the function¹⁵ $f : x \in \mathbb{R} \mapsto 2x$;
- ▶ $T[u_3] \sim T[u_4]$, where the cipher is the function¹⁶ $g : x \in \mathbb{R} \mapsto x^2$;
- ▶ $T[u_3] \approx T[u_5]$ since 3 would have two antecedents;
- ▶ $T[u_9] \sim T[u_{10}]$ and $T[u_9] \sim T[u_{14}]$, with the same cipher $h : 3 \mapsto 4$;
- ▶ $T[u_9] \sim T[u_{13}]$ with the cipher $\text{id} : 3 \mapsto 3$.

Remark 3.9 Note that the ciphers given above are not the only ones possible. For instance, the following cipher works equally well for trees

$$T[u_3] \text{ and } T[u_4]: g' = \begin{cases} 2 \mapsto 4, \\ 3 \mapsto 16, \\ 4 \mapsto 9. \end{cases}$$

The algorithm starts by considering the vertex q_0 . Since $T(q_0) = \{u_0\}$, the partition contains only one element and we create a single vertex p_0 , which has as label 0 and no children. There is nothing to remove from the sets $T(q')$. We obtain the graph described in Figure 3.9a.

Then, the vertex q_1 is treated, with $T(q_1) = \{u_1, u_2\}$. Since $T[u_1] \sim T[u_2]$ – with cipher f , the partition \mathfrak{P} contains a single element P . Arbitrarily, $s(P) = u_1$. A new vertex p_1 is created, with label 1, and two arcs connecting p_0 and p_1 are created, bearing the ciphers id (for u_1) and f (for u_2). The

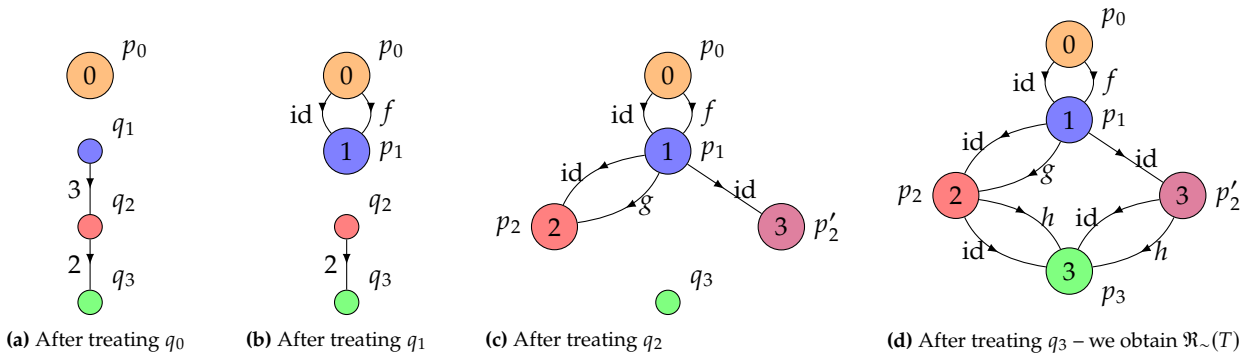


Figure 3.9: State of the graph obtained from $\mathfrak{R}(T)$ after each iteration of Algorithm 6 on the tree of Figure 3.8. Nodes are colored accordingly to their equivalence class with respect to \sim for labeled nodes, and to \simeq for unlabeled ones. The cipher on the arcs are defined as $f(x) = 2x$, $g(x) = x^2$ and $h : 3 \mapsto 4$.

nodes u_6, u_7 and u_8 are removed from $T(q_2)$, as well as nodes u_{15} to u_{20} from $T(q_3)$. The resulting graph is shown in [Figure 3.9b](#).

For vertex q_2 , with $T(q_2) = \{u_3, u_4, u_5\}$, the partition contains two elements: $P = \{u_3, u_4\}$ and $P' = \{u_5\}$. We set $s(P) = u_3$ and $s(P') = u_5$, and create two vertices p_2 and p'_2 , with respective labels 2 and 3. Both are linked to p_1 , with arcs bearing ciphers id (for u_3) and g (for u_4) for p_2 , and id (for u_5) for p'_2 . u_{11} and u_{12} are then removed from $T(q_3)$, and we get the graph of [Figure 3.9c](#).


Finally, considering vertex q_3 , with $T(q_3) = \{u_9, u_{10}, u_{13}, u_{14}\}$, the partition is reduced to a single element P containing all the nodes from $T(q_3)$. We set $s(P) = u_9$, and create vertex p_3 , linked to p_2 and p'_2 , where two arcs connect both vertex to p_3 , with ciphers id (for u_9 and u_{13}) and h (for u_{10} and u_{14}). The algorithm stops and returns [Figure 3.9d](#).

Complexity We are now interested in analyzing the complexity of [Algorithm 6](#). We recall that [Algorithm 3](#), to build $\mathfrak{R}(T)$, has complexity $O(\#T \deg(T))$ for ordered trees and $O(\#T \deg(T) \log(\deg(T)))$ for unordered trees.

Moreover, we do not (yet) have at our disposal the algorithm to check whether two trees are isomorphic up to a cipher. This algorithm will be presented in [Chapter 4](#). Since this algorithm is called in [Algorithm 6](#) to test whether subtrees of T are isomorphic, the complexity of each of these calls can be bounded by the complexity of calling it on T against itself, which we note as $C_{\sim}(T)$.

Under the assumption that the bound above is valid, we have the following theorem.

Theorem 3.6 *Algorithm 6 has complexity $O(\#T^2 C_{\sim}(T))$.*

Proof. The proof lies in [Appendix A.3](#). 

Compared to the classical DAG compression algorithm, the additional complexity can be explained by the phase where the subtrees are partitioned by equivalence class – the reader is referred to the proof for complete details.

On the construction of tree cipherings

4

Between every two pines is a doorway
to a new world.

John Muir

As already stated in [Chapter 3](#) (notably in [Remark 3.1](#)), the problem of constructing a tree ciphering between two labeled trees is only a problem when the two trees are unordered. It is therefore assumed in this chapter that the trees considered are unordered. The goal of this chapter is to build an algorithm that decides whether a tree ciphering exists between two trees, and if so, builds it.

In [Section 4.1](#), we discuss the difficulty of constructing a tree ciphering – we notably recall that the tree ciphering isomorphism problem is as difficult as graph isomorphism. We also present our strategy to address the problem.

Before introducing a two-fold algorithm – preprocessing then backtracking – to solve the problem in [Section 4.3](#), we first define in [Section 4.2](#) a number of tools and concepts used in our method. Finally, [Section 4.4](#) analyses the effectiveness of the proposed algorithm – from a theoretical point of view but also with numerical simulations.

Most of this chapter is reproduced from [\[26\]](#), but has been largely rewritten. In particular, anything related to the backtracking part of the algorithm is new material; the theoretical analysis is also more extensive than in the original paper.

4.1 Addressing the problem

The tree ciphering isomorphism problem (on unordered trees), previously introduced as “marked trees isomorphism” in [\[24\]](#), has been proven to be difficult as the following theorem stands.

Theorem 4.1 *The tree ciphering isomorphism problem is graph isomorphism complete.*

The reduction from an instance of graph isomorphism to an instance of tree ciphering isomorphism is actually linear in the size of the graph, as per the upcoming proof, reproduced from [\[24, Section 6.4\]](#). Although not necessary for the understanding of this chapter, this linear reduction is of interest and will be discussed in [Chapter 9](#).

Proof. Given a directed graph G , number its vertices from 1 to n . Build a tree T_G of height 2 such that (i) the root has label 0 and exactly n children; (ii) the children of the root are labeled from 1 to n , each corresponding

4.1 Addressing the problem	37
4.2 Framework	39
Partial bijections	39
Key idea of the algorithm	40
Bags	40
Collections	41
On the cardinality of the search space	43
4.3 The algorithm	44
Preprocessing part	45
Backtracking part	48
4.4 Analysis of the algorithm	49
Theoretical analysis	50
Experimental protocol	52
The preprocessing breaks the cardinality of the search space	52
Computation time repartition between backtracking and preprocessing	54
When trees are not isomorphic up to a cipher	55

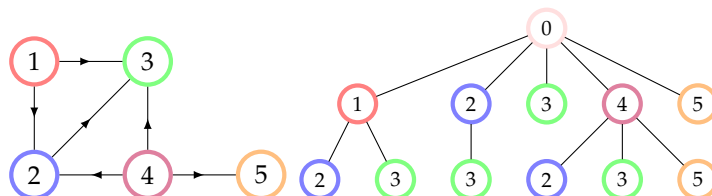
[\[26\]](#): Ingels et al. (2021), ‘Isomorphic Unordered Labeled Trees up to Substitution Ciphering’

[\[24\]](#): Booth et al. (1979), *Problems polynomially equivalent to graph isomorphism*

to a vertex of G ; and (iii) the node labeled $i \in \{1, \dots, n\}$ has a child with label j if there exists an arc $i \rightarrow j$ in G . Note that the graph can be reconstructed from the labeled tree. For any two graphs G and G' , if there exists a tree ciphering between T_G and $T_{G'}$, then G and G' are isomorphic, and reciprocally.

An example of reduction is presented in [Figure 4.1](#). 🌿

Figure 4.1: A graph (left) and its reduction as a labeled tree (right). Colors have been added for better readability.



We have seen with [Equation 3.1](#) that the number of isomorphisms between two trees is expressed as a product of factorials. Since we explore this set to find a tree ciphering, it is not difficult to explain the theoretical difficulty of the problem with the combinatorial explosion of the cardinality of the search space.

[24]: Booth et al. (1979), *Problems polynomially equivalent to graph isomorphism*

[70]: Babai (1979), ‘Monte-Carlo algorithms in graph isomorphism testing’

[71]: Klawe et al. (1982), ‘Isomorphism testing in hookup classes’

[25]: McKay et al. (2014), ‘Practical graph isomorphism, II’

[67]: Schönig (1987), ‘Graph isomorphism is in the low hierarchy’

[72]: Canzar et al. (2015), ‘On tree-constrained matchings and generalizations’

[73]: Mastrolilli et al. (2012), ‘Constrained matching problems in bipartite graphs’

Remark 4.1 In [24], the authorship of the term “marked trees” is attributed, via inner reference [31], to D.G. Corneil, M. Klawe and A. Proskurowski, in an article announced as in preparation, and entitled “Marked trees and the graph isomorphism problem”. Unfortunately, to the best of our knowledge, this paper was never published.

We also find the term “marked trees” in reference [10] of [70], referring to an oral communication of M. Klawe, entitled “Marked trees are isomorphism complete”. Unfortunately, once again, we were unable to find a written record of this communication.

To the best of our knowledge, the three authors previously mentioned have co-authored only one article [71] which does not mention the term marked trees. It would seem that besides the complexity result stated above, no research has been published on the practical resolution of the marked tree isomorphism problem, until our contribution.

As mentioned in [Connection with the graph isomorphism problem](#) (p. 25), there are efficient practical algorithms for dealing with graph isomorphism [25], although it is still unclear whether the problem is theoretically solvable in polynomial time [67]. These algorithms rely on assigning its equivalence class to a given graph – which is convenient for retrieving graphs from a database and processing graphs independently. However, none of these algorithms explicitly constructs an isomorphism between two graphs.

We saw in [Section 3.3](#) that to adapt DAG compression to labeled trees, we need to know explicitly the tree ciphering ϕ – or rather, the associated cipher f_ϕ . Consequently, the algorithms of the literature for graph isomorphism are not adapted to our need.

In the sequel of this chapter, we propose an algorithm that explicitly builds a tree ciphering. The method we develop is closer to constrained matching problems on bipartite graphs [72, 73] than to the usual techniques of graph isomorphism resolution.

In details, since we are building two bijections simultaneously – one on trees and the other on labels – that must be compatible, the general idea is to use the constraints of one to make deductions about the other, and vice versa. For instance, whenever two nodes must be mapped together, so are their labels, and therefore you can eliminate all potential tree isomorphisms that would have mapped those labels differently.

The algorithm operates in two phases. First, preprocessing is designed to find as many deductions as possible on the mappings, each of which contributes to decreasing the size of the search space for a solution. When no more deductions are possible, the preprocessing phase stops. To complete (if feasible) the two bijections, and to explore the remaining space, we then launch a backtracking phase [74]. The idea is to choose a potential mapping, perform the mapping and look recursively if we can build a solution from this choice. If not, we backtrack to that choice and make another one. If no choice leads to a solution, then we conclude that the problem has no solution.

[74]: Dechter et al. (1998), 'Backtracking algorithms for constraint satisfaction problems; a survey'

4.2 Framework

In this section, we introduce the tools and concepts that are necessary to understand the algorithm presented in the next section.

Partial bijections

The goal of the algorithm is to construct – if possible – two bijections: $\phi \in \text{Isom}(T_1, T_2)$ for the nodes, and $f : \mathcal{A}(T_1) \rightarrow \mathcal{A}(T_2)$ for the labels (so that $f = f_\phi$ with the notation of Section 3.2). They will be built incrementally, starting as empty mappings $\emptyset \mapsto \emptyset$. Therefore, we are going to increase their domain of definition and their image, as well as assessing at each update that they remain bijective.

A *partial* bijection ψ from A to B is an injective function from a subset S_ψ of A to B ¹⁷. Let $a \in A$ and $b \in B$; suppose we want to determine if the couple (a, b) is compatible with ψ – in the sense that it respects (or does not contradict) the partial bijection. First, if $a \in S_\psi$, then b must be equal to $\psi(a)$. Otherwise, if $a \notin S_\psi$, then b must not be in the image of ψ , i.e. $\forall s \in S_\psi, \psi(s) \neq b$. If those conditions are respected, then (a, b) is compatible with ψ ; furthermore, if $a \notin S_\psi$, then we can extend ψ on $S_\psi \cup \{a\}$ by defining $\psi(a) = b$ so that ψ remains a partial bijection.

17: Indeed, ψ is a bijection from S_ψ to B restricted to the image of ψ .

Formally, for any $a \in A$ and $b \in B$, with ψ a partial bijection from A to B , we define

$$\text{ExtBij}(\psi, a, b) = (a \in S_\psi \implies \psi(a) = b) \wedge (a \notin S_\psi \implies \forall s \in S_\psi, \psi(s) \neq b) \quad (4.1)$$

so that $\text{ExtBij}(a, b, \psi)$ returns \top if and only if the couple (a, b) is compatible with the partial bijection ψ . For the sake of brevity, we assume that the procedure ExtBij also updates the partial bijection in the case $a \notin S_\psi$ by defining $\psi(a) = b$ – naturally only if the procedure returned \top .

In the sequel, ExtBij is used to update both partial bijections ϕ (from T_1 to T_2) and f (from $\mathcal{A}(T_1)$ to $\mathcal{A}(T_2)$). However, if one uses a restricted

version of ciphers as presented in [Remark 3.3](#), one must design a specific version of ExtBij to update f , accounting for the desired properties.

Key idea of the algorithm

The fundamental premise of the algorithm is based on the following observation: whenever two nodes are mapped in a tree ciphering, they must share the exact same topology (in the sense that the subtrees rooted in these nodes must be topologically isomorphic), and their labels are mapped together. Namely, let $\phi \in \text{Cipher}(T_1, T_2)$, $u \in T_1$ and $v \in T_2$.

Observation 4.1 *If $\phi(u) = v$, then*

- (i) $\text{depth}(u) = \text{depth}(v)$;
- (ii) $[u] = [v]$;
- (iii) $f_\phi(\bar{u}) = \bar{v}$.

These are necessary conditions, but not sufficient. However, if we find two nodes that are the only ones to share the same values on some or all of those criteria, then these two nodes must necessarily be mapped together. For instance, the roots are by definition the only nodes in each tree to have a depth of zero and therefore must be mapped together, without even considering the other criteria.

The core of the algorithm thus consists of partitioning, successively by considering each criterion one after the other, the nodes of each tree; and as soon as we find solitary nodes, we map them together. Moreover, whenever nodes are mapped together, this has repercussions on the whole partition, as per the following observation.

Observation 4.2 *$\phi(u) = v$ also implies that*

- (iv) $\phi(\mathcal{P}(u)) = \mathcal{P}(v)$ and
- (v) $\forall w \in \mathcal{C}(u), \phi(w) \in \mathcal{C}(v)$.

The general concept of the algorithm is as follows. At a given instant, the nodes of the trees will be divided into three groups: those that have already been mapped, those that are in bags, and those that are in collections. Bags and collections are structures that will be defined in the following subsections. The nodes present in a bag all have in common some combination of the criteria defined above – and are therefore candidates to be mapped together. Collections contain sets of nodes sharing some criteria, but not yet distributed in bags.

We will define, for bags and collections, deduction rules allowing to change the distribution of nodes in these structures, notably by mapping them. For example, if a bag contains only two nodes, one from each tree, then we can map them together (like the roots in the example above).

Bags

A bag B is a couple (B_1, B_2) such that $B_1 \subseteq T_1$, $B_2 \subseteq T_2$, and $\#B_1 = \#B_2$ – this number being simply denoted by $\#B$. The initialization of bags

will be introduced later in [Section 4.3](#), so that a node can not belong simultaneously in two different bags. We denote the set of all bags by \mathbb{B} and by $\sigma : T_1 \cup T_2 \rightarrow \mathbb{B} \cup \{\emptyset\}$ the function that associate each node to the bag it belongs to – if any.

As stated earlier in [Key idea of the algorithm](#) (p. 40), whenever $\#B = 1$, the two corresponding nodes from T_1 and T_2 must be mapped together.

Deduction Rule 4.1 *As long as there exist bags $B \in \mathbb{B}$ with $B = (\{u\}, \{v\})$, call $\text{MAPNODES}(u, v, \phi, f, \sigma)$ – then remove B from \mathbb{B} .*

The procedure MAPNODES is presented in [Algorithm 7](#). If MAPNODES returns \perp , then we can conclude that $T_1 \approx T_2$. For instance, this can happen if we map two nodes whose topology is identical, but whose labels contradict those already mapped. As stated in [Observation 4.2](#), once we have mapped u and v , if their parents are not already mapped, they must be; and also, the respective children of u and v should be separated from other nodes – and recursively their children, and so on – via the SPLITCHILDREN procedure, presented in [Algorithm 8](#). Note that, in line 4, an equivalent condition would be $B_v \neq \emptyset$ ¹⁸. Since the procedure performs its recursion on the children, it eventually reaches the leaves and stops. Note that if no changes are made to a bag (because there are no children or nodes other than children – see line 4), no recursive call is made on that bag. The procedure is illustrated in [Figure 4.2](#).

Algorithm 8: SPLITCHILDREN

Input: S_u, S_v

- 1 Let $C_u = \bigcup_{w \in S_u} C(w)$ and $C_v = \bigcup_{w \in S_v} C(w)$
 - 2 **for** $B \in \mathbb{B}$ **do**
 - 3 Let $B_u = B_1 \cap C_u$ and $B_v = B_2 \cap C_v$
 - 4 **if** $B_u \neq \emptyset$ **and** $B_1 \setminus B_u \neq \emptyset$ **then**
 - 5 Add (B_u, B_v) to \mathbb{B} and $(B_1 \setminus B_u, B_2 \setminus B_v)$ to \mathbb{B}
 - 6 $\text{SPLITCHILDREN}(B_u, B_v)$
 - 7 $\text{SPLITCHILDREN}(B_1 \setminus B_u, B_2 \setminus B_v)$
 - 8 Delete B from \mathbb{B}
-

Note that since the bags are modified (either via the children or the parents) after mapping two nodes, the number of bags meeting the prerequisite of [Deduction Rule 4.1](#) can vary after each call to MAPNODES . However, the process of deduction ends at some point since we can not map more nodes than there exist in the trees.

Collections

While bags contain nodes that can be mapped together, collections contain sets of nodes that are susceptible to be put together in a bag. The need for such structures arises – as will be seen later – when labels begin to be considered.

Formally, a collection C is a couple (C_1, C_2) where $C_1 \subseteq 2^{T_1}$ and $C_2 \subseteq 2^{T_2}$ – denoting by 2^S the powerset of S . We impose the following additional constraints:

Algorithm 7: MAPNODES

Input: $u \in T_1, v \in T_2, \phi, f, \sigma$

- 1 **if** $\text{EXTBII}(f, \bar{u}, \bar{v})$ **and** $\text{EXTBII}(\phi, u, v)$ **then**
 - 2 Remove u from $\sigma(u)$
 - 3 Remove v from $\sigma(v)$
 - 4 Set $\sigma(u) = \sigma(v) = \emptyset$
 - 5 $\text{SPLITCHILDREN}(\{u\}, \{v\})$
 - 6 **if** $\phi(\mathcal{P}(u)) = \mathcal{P}(v)$ **then**
 - 7 return \top
 - 8 **else**
 - 9 return $\text{MAPNODES}(\mathcal{P}(u),$
 $\mathcal{P}(v), \phi, f, \sigma)$
 - 10
 - 11 **else**
 - 12 return \perp
-

18: If the equivalence is not true, then we can conclude that $T_1 \approx T_2$.

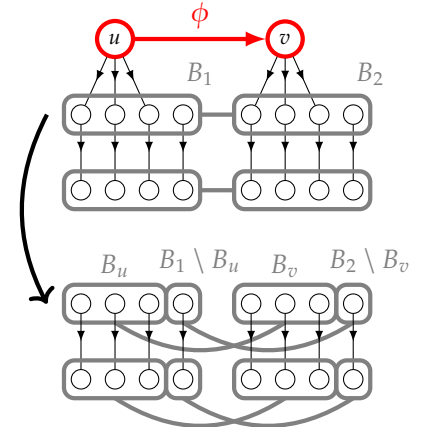


Figure 4.2: Illustration of the SPLITCHILDREN procedure. The bag formed of B_1 and B_2 (above) is split into two bags after the mapping of u and v , one composed of the children of u and v , and the other with the remaining nodes (below). Note that the latter bag is eligible for [Deduction Rule 4.1](#). Recursively, the bags of their descendants are also split.

- (i) $\forall P \in C_i, \exists a \in \mathcal{A}(T_i), \forall u \in P_i, \bar{u} = a$ – i.e., all nodes from the same set share a common label. This common label is denoted by \bar{P} .
- (ii) Denoting by $C_i(n) = \{P \in C_i : \#P = n\}$, $\#C_1(n) = \#C_2(n)$ for all $n \in \mathbb{N}$. This common cardinality is denoted by $\#C(n)$. Note that the number of n 's for which $\#C(n) > 0$ is finite.

The set of all collections is denoted by \mathbb{C} . The union of the nodes contained in the bags, in the collections and those already mapped must form a partition of the nodes; in other words, a node can belong at most to only one collection, and to only one set in it. Therefore, we redefine the function $\sigma : T_1 \cup T_2 \rightarrow \mathbb{B} \cup \mathbb{C} \cup \{\emptyset\}$ that associates each node to the bag or the collection it belongs to – if any.

The first deduction rule on collections concerns the labels themselves. If we find labels in a collection C that are already mapped in f , then we can isolate the corresponding sets from the rest of the collection. More precisely, denoting by $C_i[a] = \{P \in C_i : \bar{P} = a\}$, we introduce the following deduction rule.

Deduction Rule 4.2 *As long as there exist collections $C \in \mathbb{C}$ and labels a, b verifying $f(a) = b$, such that $C_1[a] \neq \emptyset$ and $C_1 \setminus C[a] \neq \emptyset$, add the two following collections $(C_1[a], C_2[b])$ and $(C_1 \setminus C_1[a], C_2 \setminus C_2[b])$ to \mathbb{C} – then remove C from \mathbb{C} .*

Equivalently, we could verify $C_2[b] \neq \emptyset$ and $C_2 \setminus C_2[b] \neq \emptyset$ – if this equivalence is not true, then the distribution of labels does not allow the construction of a cipher: we can conclude that $T_1 \not\approx T_2$.

The second deduction rule is complementary to the previous one. If a collection contains sets of same cardinality that all share a common label in T_1 and T_2 , then those labels must surely be mapped together.

Deduction Rule 4.3 *As long as there exist collections $C \in \mathbb{C}$, integers $n \in \mathbb{N}$ and labels a, b not already mapped in f , such that $\forall P \in C_1(n), \bar{P} = a$ and $\forall Q \in C_2(n), \bar{Q} = b$, then call $\text{ExtBij}(f, a, b)$. If it returns \perp , conclude that $T_1 \approx T_2$.*

The last deduction rule concerns the cardinalities: if $\#C(n) = 1$, then the two corresponding sets must form a bag, and their labels must be mapped together – if not already the case.

Deduction Rule 4.4 *As long as there exist collections $C \in \mathbb{C}$ and integers $n \in \mathbb{N}$ such that $C_1(n) = \{P\}$ and $C_2(n) = \{Q\}$; if $\text{ExtBij}(f, \bar{P}, \bar{Q})$, add a new bag (P, Q) to \mathbb{B} and remove P and Q from C – otherwise stop and conclude that $T_1 \approx T_2$.*

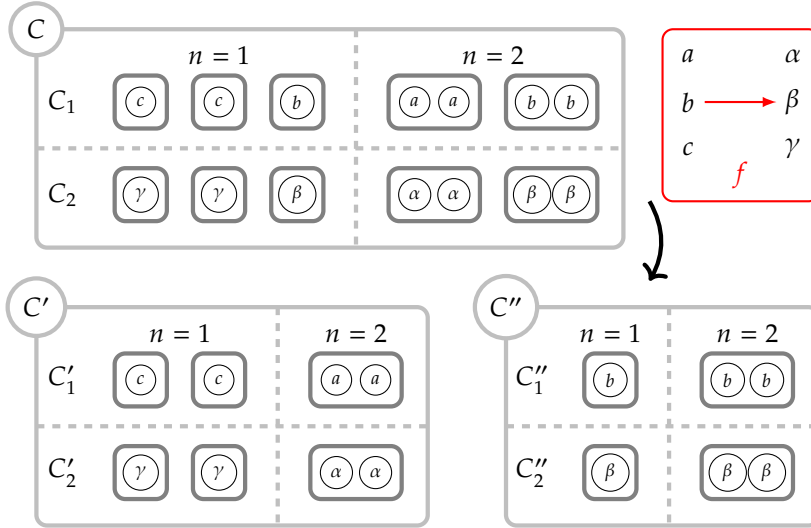


Figure 4.3: Illustration of the deduction rules on collections. Given a collection C (above, left) and the current mapping of the labels (above, right), via [Deduction Rule 4.2](#) we split C into two collections C' (below, left) and C'' (below, right). Applying [Deduction Rule 4.3](#) to C' , we map $f(a) = \alpha$ and $f(c) = \gamma$. Finally, [Deduction Rule 4.4](#) allow to create three bags with (i) the sets in $C'_2(2)$ and $C'_2(2)$; (ii) the sets in $C''_1(1)$ and $C''_2(1)$ and (iii) the sets in $C''_1(2)$ and $C''_2(2)$.

The three previous rules are illustrated in [Figure 4.3](#). Since each deduction rule modifies the collections or the label mapping, each time a deduction is made, we have to look again if one of them applies.

As in [Observation 4.2](#), once two nodes are mapped, this provides information on the mapping of their parent and children. Here is how to take into account this information when dealing with collections:

- ▶ For the parents, if they were found in a collection, they are mapped together and the other elements of their respective sets are put in a bag.
- ▶ For the children, the lines presented in [Algorithm 9](#) must be inserted at the end of [Algorithm 8](#). We send in new collections the elements whose intersection with $C(u)$ (respectively $C(v)$) and their complementary are not null. The union of all the elements thus separated is used for the recursive calls, which terminate correctly since, once again, we end up reaching the leaves.

Note that since the bags and collections are modified after this operation, we must apply the Deduction Rules to check whether new deductions can be made or not.

On the cardinality of the search space

The search space for a tree ciphering is that of tree isomorphisms, whose size is given by [Equation 3.1](#). Nevertheless, since the upcoming algorithm does not explicitly parse tree isomorphisms, but rather bijections between T_1 and T_2 , in order to directly construct an isomorphism that is already a cipher (if it is possible), we have to analyse the size of the search space differently. In fact, the whole point of bags and collections is to break the size of the bijection space – which is $(\#T_1)!$ – by partitioning the nodes increasingly finely.

First, a B bag containing nodes in T_1 and as many in T_2 allows to create exactly $(\#B)!$ mappings between them (possibly, not all mappings lead to a tree isomorphism). Then, given a collection C , since we can only put in bags sets of the same size, for each integer n , there are $(\#C(n))!$ possible

Algorithm 9: SPLITCHILDRENBIS

```

/*  $C_u$  and  $C_v$  are defined in
Algorithm 8. */
9 for  $C \in \mathbb{C}$  and  $n \in \mathbb{N}$  do
10   Let  $C'$  and  $C''$  be empty
   collections
11   Let  $S_1, S'_1$  be empty sets
12   for  $P \in C_1(n)$  do
13     Let  $P_u = P \cap C_u$ 
14     if  $P_u \neq \emptyset$  and  $P \setminus P_u \neq \emptyset$ 
       then
15       Add  $P_u$  to  $C'$ 
16       Add  $P \setminus P_u$  to  $C''$ 
17        $S_1 \leftarrow S_1 \cup P_u$ 
18        $S'_1 \leftarrow S'_1 \cup (P \setminus P_u)$ 
19       Remove  $P$  from  $C$ 
20   Let  $S_2, S'_2$  be empty sets
21   for  $Q \in C_2(n)$  do
22     Let  $Q_v = Q \cap C_v$ 
23     if  $Q_v \neq \emptyset$  and  $Q \setminus Q_v \neq \emptyset$ 
       then
24       Add  $Q_v$  to  $C'$ 
25       Add  $Q \setminus Q_v$  to  $C''$ 
26        $S_2 \leftarrow S_2 \cup Q_v$ 
27        $S'_2 \leftarrow S'_2 \cup (Q \setminus Q_v)$ 
28       Remove  $Q$  from  $C$ 
29   Add  $C'$  and  $C''$  to  $\mathbb{C}$ 
30   SPLITCHILDREN( $S_1, S_2$ )
31   SPLITCHILDREN( $S'_1, S'_2$ )

```

ways to create bags, each of them yielding in turn $n!$ possible mappings of the nodes, for a total of $(\#C(n))! \times (n!)^{\#C(n)}$ mappings. Hence, given \mathbb{B} and \mathbb{C} , the current size of the search space $N(\mathbb{B}, \mathbb{C})$ is given by

$$N(\mathbb{B}, \mathbb{C}) = \prod_{B \in \mathbb{B}} (\#B)! \prod_{C \in \mathbb{C}} \left(\prod_n (n!)^{\#C(n)} (\#C(n))! \right). \quad (4.2)$$

Therefore, subdividing bags and collections modifies the size of the search space, as follows:

- ▶ When a bag of size n is split in two due to the action of `SPLITCHILDREN`, say with $\#P_u = p$, then we go from $n!$ possibilities to $p!(n-p)!$, reducing the space by a factor of $\binom{n}{p}$.
- ▶ When `SPLITCHILDREN` is applied to a collection, several sets can be split and put together in the same collection. Suppose we split k sets P of size n into sets P_u of size p (and sets $P \setminus P_u$ of size $n-p$), where $k \leq \min(\#C(n), \deg(u))$; we go from $(\#C(n))! \times (n!)^{\#C(n)}$ possibilities to $(\#C(n)-k)! \times (n!)^{\#C(n)-k} \times k!(p!)^k \times k!((n-p)!)^k$.

The space is therefore modified by a factor of $\frac{1}{k!} \binom{\#C(n)}{k} \left(\frac{n}{p} \right)^k$.

Because of the denominator which, depending neither on p nor on n , can be arbitrarily large, it is possible, in some pathological cases, that this term is smaller than 1, and thus reflects an enlargement of the search space.

- ▶ When recursively mapping the parents of two nodes, if they were present in a bag B , we go from $(\#B)!$ to $(\#B-1)!$, reducing the space by a factor of $\#B$; if they were present in a collection C and sets of size n , then we go from $(\#C(n))!(n!)^{\#C(n)}$ possibilities to $(\#C(n)-1)!(n!)^{\#C(n)-1} \times (n-1)!$, reducing the space by a factor of $\#C(n) \times n$.
- ▶ When applying [Deduction Rule 4.2](#) and splitting a collection C into two new collections C' and C'' , for each integer n , say we have $\#C(n) = p+q$, $\#C'(n) = p$ and $\#C''(n) = q$; then we go from $(p+q)!(n!)^{p+q}$ possibilities to $p!(n!)^p \times q!(n!)^q$, reducing the space by a factor of $\binom{p+q}{q}$.
- ▶ Finally, [Deduction Rule 4.1](#), [Deduction Rule 4.3](#) and [Deduction Rule 4.4](#) do not modify the size of the search space.

As an example, the bag B of [Figure 4.2](#) represents $4! = 24$ potentials mappings, but only $3! \times 1! = 6$ after being split up; the collection C of [Figure 4.3](#) accounts for $3! \times (1!)^3 \times 2! \times (2!)^2 = 48$ potentials mappings, but only 8 after being split up.

4.3 The algorithm

The algorithm we propose to build a tree ciphering is split into two distinct phases. First, we will try to determine as many mandatory mappings as possible on the two bijections we are building, the one on the nodes and the one on the labels. Each mapping reduces the search space; thus the objective of this preprocessing phase is to break, as much as possible, the combinatorial complexity of the search space. Then, we start a backtracking procedure to complete the bijections – if possible.

Preprocessing part

Initialization Let T_1 and T_2 two labeled trees. To assess that $T_1 \simeq T_2$, we call AHU algorithm – see [The Aho, Hopcroft & Ullman algorithm](#) (p. 13) – which concludes in linear time, while assigning to the nodes of T_1 and T_2 their equivalence classes under \simeq – which we will need later in the algorithm. Let $\phi : \emptyset \mapsto \emptyset$ and $f : \emptyset \mapsto \emptyset$. We initialize $\mathbb{C} = \emptyset$ and $\mathbb{B} = \{B\}$ where $B = (\{u \in T_1\}, \{v \in T_2\})$. At this stage, the search space is exactly the bijections between the nodes of T_1 and T_2 , for a size of $(\#T_1)!$. This space is much larger than $\text{Isom}(T_1, T_2)$, and we aim to reduce its cardinality via successive partitioning in the sequel.

Throughout this section, the different steps will be illustrated on a running example, presented in [Figure 4.4](#).

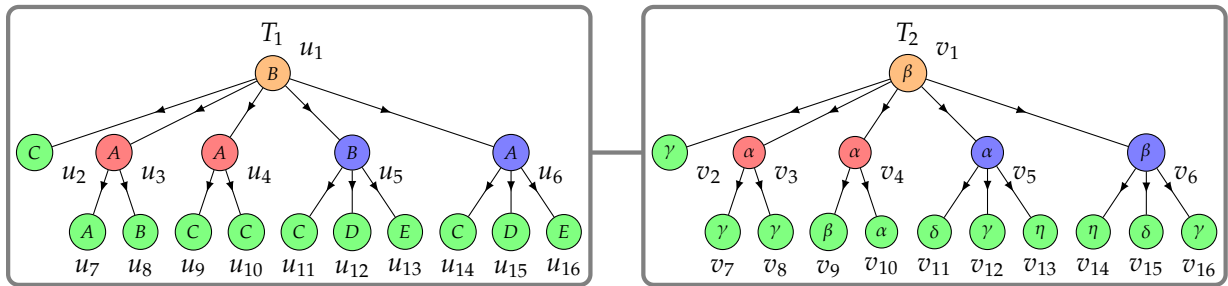


Figure 4.4: Two topologically isomorphic labeled trees T_1 (left) and T_2 (right). The color on nodes indicates the class of equivalence under \simeq . The nodes have been numbered from u_1 to u_{16} in T_1 (resp. from v_1 to v_{16} in T_2) in breadth-first search order. The size of $\text{Isom}(T_1, T_2)$ is equal to $N_{\simeq}(T_1) = (2!)^4(3!)^2 = 576$. The gray boxes indicate the only starting bag and its components; the size of the starting search space is $(\#T_1)! = 16! \approx 2.09 \times 10^{13}$.

Depth Following point (i) of [Observation 4.1](#), we know that nodes susceptible to be mapped together must share a common depth. Therefore, we partition the only bag B by depth, i.e., we define $T_i(d) = \{u \in T_i : \text{depth}(u) = d\}$, for $d \in \{0, \dots, \text{depth}(T_i)\}$, and add as many bags $(T_1(d), T_2(d))$ in \mathbb{B} – and we remove B from \mathbb{B} .

After this operation, we apply [Deduction Rule 4.1](#). Since the roots are the only nodes with zero depth, the rule will at least apply to their bags and they will be mapped together.

[Figure 4.5](#) illustrates this step on the running example.

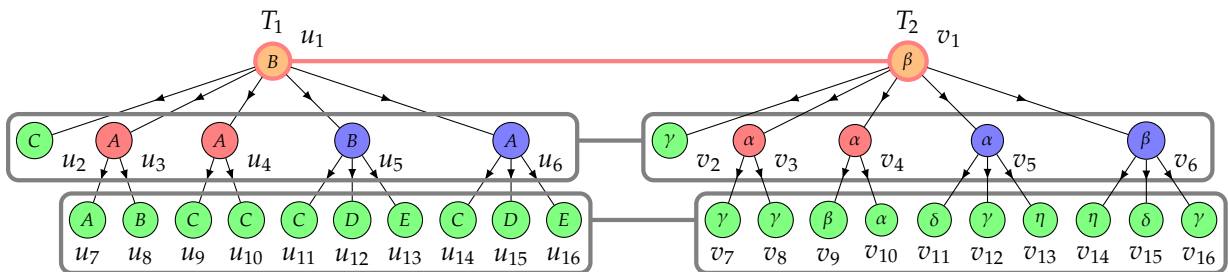


Figure 4.5: After partitioning by depth. Since u_1 and v_1 are the only nodes with depth zero, via [Deduction Rule 4.1](#), we map $\phi(u_1) = v_1$ and $f(B) = \beta$. The children of u_1 and v_1 should be set aside from the other nodes according to the `SPLITCHILDREN` procedure, but they are already alone in their bag. The size of the current space is now $5! \times 10! = 435,456,000$.

Equivalence class We now refer to point (ii) of [Observation 4.1](#), and we subdivide the bags according the equivalence class of the nodes they contain. Formally, for each bag $B = (B_1, B_2)$ in \mathbb{B} , we define $B_i(c) = \{u \in B_i : [u] = c\}$. We add a new bag $(B_1(c), B_2(c))$ to \mathbb{B} for each such c , and remove B from \mathbb{B} .

Once all bags have been partitioned, apply again [Deduction Rule 4.1](#).

[Figure 4.6](#) presents the result of this step on the running example.

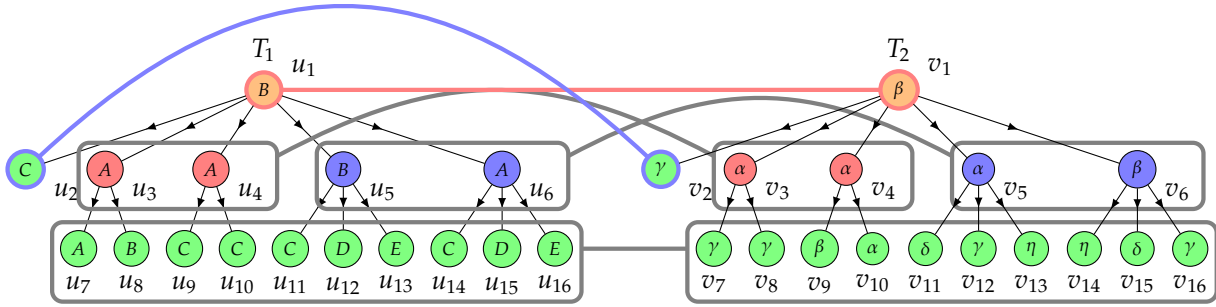


Figure 4.6: After partitioning by equivalence class. Since u_2 and v_2 are the only nodes with class $\color{green}\bullet$ and depth 1, via [Deduction Rule 4.1](#), we map $\phi(u_2) = v_2$ and $f(C) = \gamma$. Also, since u_2 and v_2 have no children and their parents are mapped together, there is nothing more to do. The size of the current space is now $(2!)^2 \times 10! = 14,515,200$.

Parents Before considering labels in the next step, we focus first on [Observation 4.2](#). To map two nodes together, their parents must also be mappable with each other. In particular, this implies that such parents belong to the same bag.

Formally, we inspect each bag $B = (B_1, B_2)$ in \mathbb{B} by increasing depth (recall that at this point, all nodes of the same bag share the same depth and the same equivalence class). We partition the nodes of B as follows: let $B_i(s) = \{u \in B_i : \sigma(\mathcal{P}(u)) = s\}$ – where σ is the function that associate, to a node, the bag it is contained in, if any. We then add a new bag $(B_1(s), B_2(s))$ to \mathbb{B} for each such s , and remove B from \mathbb{B} . Visiting the bags by increasing depth ensures that you have subdivided the bags containing the parents before turning to the children.

Once all bags have been partitioned, apply again [Deduction Rule 4.1](#).

This step is illustrated on the running example in [Figure 4.7](#).

Labels We finally consider the last point (iii) of [Observation 4.1](#). We are interested in the labels on the nodes; this time, this criterion is not purely topological and for this reason we have to use collections. Indeed, except for the labels already mapped together in the previous deductions, we do not know which labels must go together, and we can not arbitrarily map them by creating bags.

Formally, for each $B = (B_1, B_2)$ in \mathbb{B} , we define $B_i(a) = \{u \in B_i : \bar{u} = a\}$ and $\mathcal{A}(T_i)|_B = \{a \in \mathcal{A}(T_i) : B_i(a) \neq \emptyset\}$. We add a new collection $C = (\{B_1(a) : a \in \mathcal{A}(T_1)|_B\}, \{B_2(b) : b \in \mathcal{A}(T_2)|_B\})$ to \mathbb{C} , and remove B from \mathbb{B} .

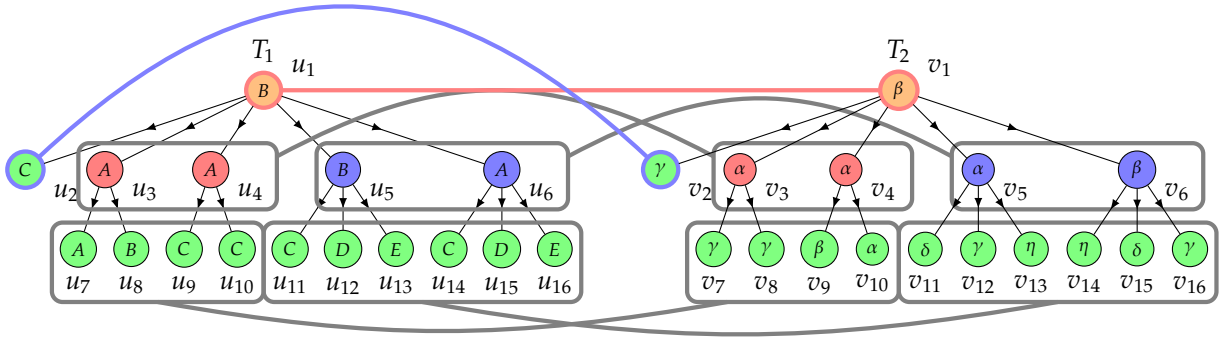


Figure 4.7: After partitioning by parents. The only modification happens when looking for nodes with depth 2; no further deduction is made. The size of the current space is now $(2!)^2 \times 4! \times 6! = 69, 120$.

Once this procedure has been applied to all bags – and therefore $B = \emptyset$ – we apply **Deduction Rule 4.2**, **Deduction Rule 4.3** and **Deduction Rule 4.4** on C . In the case bags were to be created as a result of a deduction, apply also **Deduction Rule 4.1**. We emphasize that all these rules are intertwined and of the form “as long as . . .”: each deduction made imposes to check again all the rules; the process stops only when no more deductions are made.

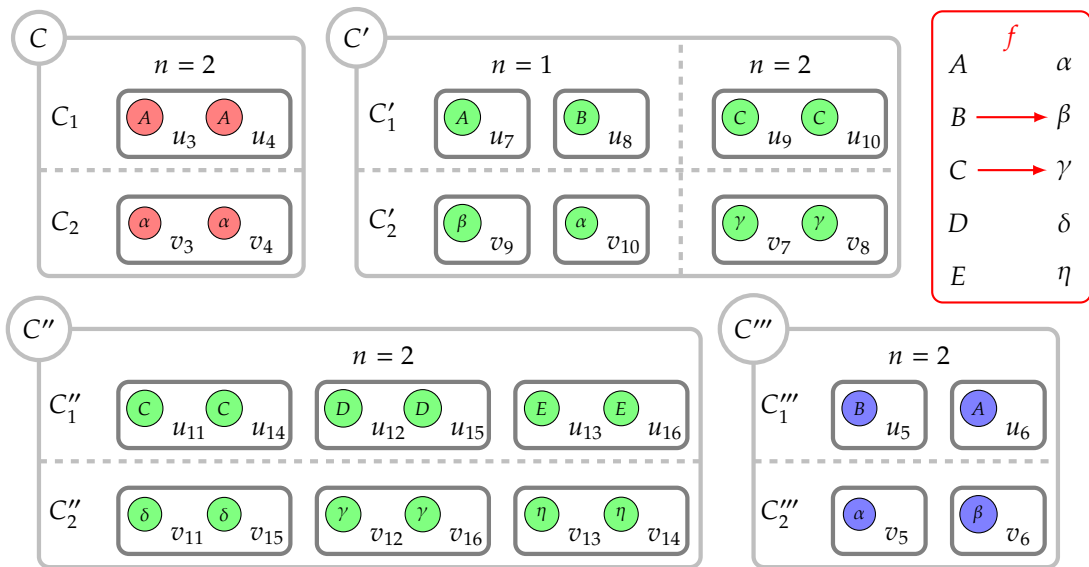


Figure 4.8: State of the system after converting all bags into collections, before any deductions, as well as the current mapping of the labels f (above, right). The size of the current space is now: $2!$ for C , $2! \times 2! = 4$ for C' , $3! \times (2!)^3 = 48$ for C'' and $2!$ for C''' ; all combined gives 768.

Figure 4.8 illustrates the state of the system after converting all bags into collections, but before applying any Deduction Rules. Applying **Deduction Rule 4.2** provokes the following changes on the running example:

- ▶ Collection C is unchanged;
- ▶ Collection C' is split into three collections: $C^{(1)} = (\{\{u_8\}\}, \{\{v_9\}\})$; $C^{(2)} = (\{\{u_9, u_{10}\}\}, \{\{v_7, v_8\}\})$ and $C^{(3)} = (\{\{u_7\}\}, \{\{v_{10}\}\})$;
- ▶ Collection C'' is split into two collections: $C^{(4)} = (\{\{u_{11}, u_{14}\}\}, \{\{v_{12}, v_{16}\}\})$ and $C^{(5)} = (\{\{u_{12}, u_{15}\}, \{u_{13}, u_{16}\}\}, \{\{v_{11}, v_{15}\}, \{v_{13}, v_{16}\}\})$;

► Collection C''' is split into two collections: $C^{(6)} = (\{\{u_5\}\}, \{\{v_6\}\})$ and $C^{(7)} = (\{\{u_6\}\}, \{\{v_5\}\})$.

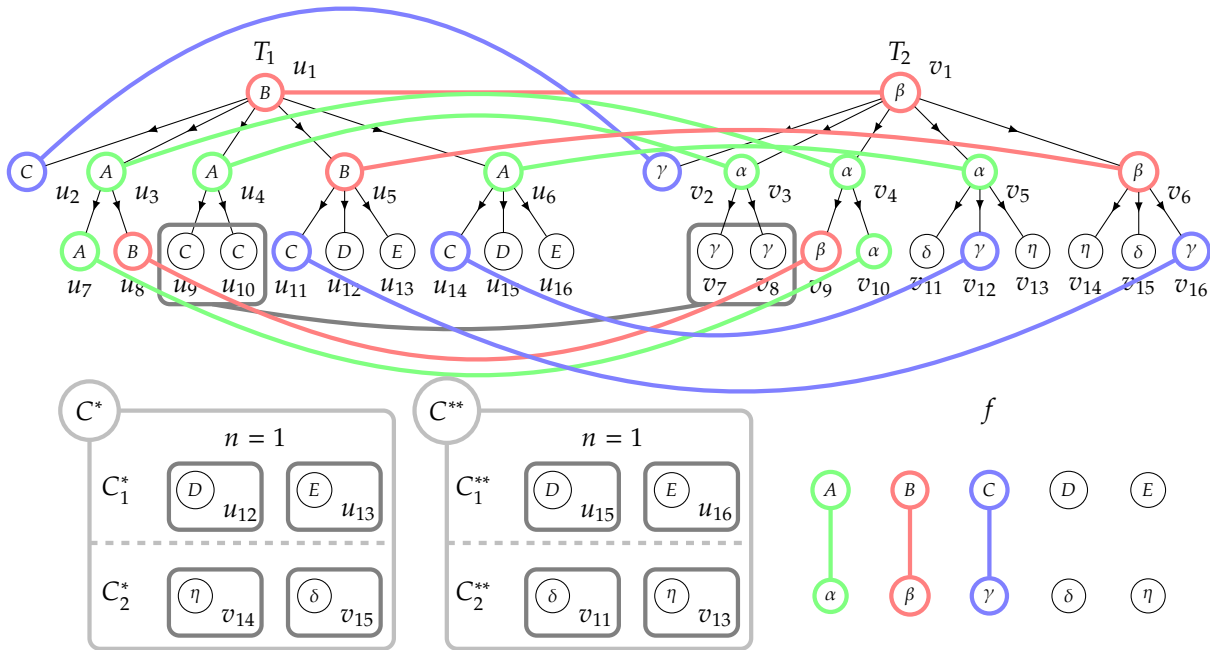


Figure 4.9: The mapping on nodes and remaining bag (above), the remaining collections (below, left) and the mapping on labels (below, right) at the end of the preprocessing phase. The collection $C^{(5)}$ has been split into two new collections C^* and C^{**} , after the mapping of (u_5, v_6) , via the `SPLITCHILDREN` procedure. The size of the final space is: $2!$ for the last bag, and $2!$ for each collections, for a total combined of 8. Note that this number is way below $N_{\approx}(T_1) = 576$, indicating that the preprocessing phase succeeded in reducing the size of the search space.

Deduction Rule 4.3 allows to map $f(A) = \alpha$ (for instance, with collection $C^{(7)}$), whereas **Deduction Rule 4.4** create several bags, most of them with one node per component, allowing new mappings (via **Deduction Rule 4.1**), as displayed in **Figure 4.9**.

Conclusion After all this steps, the preprocessing phase is finished, and the backtracking starts.

Backtracking part

The pseudocode for the backtracking part is provided in **Algorithm 11**. During the preprocessing phase, if an operation went wrong, we could immediately conclude that $T_1 \approx T_2$; here it indicates that an unfortunate choice has been made and that another one must be made, if any remains. The detection of these unsuccessful cases is done in the pseudocode via the tags **try** and **catch**¹⁹. A number of new procedures are also introduced: `SAVESTATE`, `RESTORESTATE` and `NEXTCANDIDATES`. The first two are self-explanatory, while the last one is presented in **Algorithm 10**.

The main idea behind the latter is the following: in a B bag with n nodes, we know that there are $n!$ possible mappings; however, for backtracking, the search tree only needs to branch on n children. Indeed, the first node u of B_1 will have to be mapped to one of the nodes of B_2 , and so we

19: Where **try** X ; **catch** Y executes X , and if an error is detected, executes Y instead.

consider all possible mappings between this node of B_1 and those of B_2 . The same idea also applies to collections.

Algorithm 11: BACKTRACKING

Input: $\mathbb{B}, \mathbb{C}, \phi, f, \sigma$

```

1  $\lambda, L \leftarrow \text{NEXTCANDIDATES}(\mathbb{B}, \mathbb{C})$ 
2  $S \leftarrow \text{SAVESTATE}(\mathbb{B}, \mathbb{C}, \phi, f, \sigma)$ 
3 if  $\lambda = \mathbb{B}$  then
4   for  $(u, v) \in L$  do
5     try
6       assert  $\text{MAPNODES}(u, v, \phi, f, \sigma)$ 
7       Apply Deduction Rule 4.1 to Deduction Rule 4.4
8       return  $\text{BACKTRACKING}(\mathbb{B}, \mathbb{C}, \phi, f, \sigma)$ 
9     catch
10       $\mathbb{B}, \mathbb{C}, \phi, f, \sigma \leftarrow \text{RESTORESTATE}(S)$ 
11   return  $\perp$ 
12 else if  $\lambda = \mathbb{C}$  then
13   for  $(P, Q) \in L$  do
14     try
15       assert  $\text{EXTBIJ}(f, \overline{P}, \overline{Q})$ 
16       Remove  $P$  and  $Q$  from their collection in  $\mathbb{C}$ 
17       Create a bag  $(P, Q)$  in  $\mathbb{B}$ 
18       Apply Deduction Rule 4.1 to Deduction Rule 4.4
19       return  $\text{BACKTRACKING}(\mathbb{B}, \mathbb{C}, \phi, f, \sigma)$ 
20     catch
21       $\mathbb{B}, \mathbb{C}, \phi, f, \sigma \leftarrow \text{RESTORESTATE}(S)$ 
22   return  $\perp$ 
23 else
24   return  $\top$ 

```

If we resume the example of [Figure 4.9](#), $\text{NEXTCANDIDATES}(\mathbb{B}, \mathbb{C})$ would return $\lambda = \mathbb{B}$ and $L = \{(u_9, v_7), (u_9, v_8)\}$. After mapping $\phi(u_9) = v_7$, we would deduce $\phi(u_{10}) = v_8$, and start the recursion, this time on collections. Note that all mapping choices between the remaining nodes and labels are acceptable, so the procedure will not go wrong and backtrack on this example.

4.4 Analysis of the algorithm

The interest of proceeding in two phases, compared to directly launching a backtracking procedure, is to break the cardinality of the search space to allow backtracking to be as fast as possible. From a theoretical point of view, we consider the size of the backtracking search tree, and show that our approach minimizes it. Then, on simulated data, we focus on (i) the extent to which preprocessing does break the cardinality of the search space, (ii) how the computing time is divided between preprocessing and backtracking and (iii) how the algorithm behaves when the trees are not isomorphic – versus when they are.

Algorithm 10: NEXTCANDI-

DATES

Input: \mathbb{B}, \mathbb{C}

```

1 if  $\mathbb{B} \neq \emptyset$  then
2    $\lambda \leftarrow \mathbb{B}$ 
3   Let  $B = (B_1, B_2) \in \mathbb{B}$  so that  $\#B$  is
   minimal
4   Let  $u \in B_1$ 
5    $L \leftarrow \{(u, v) : v \in B_2\}$ 
6 else if  $\mathbb{C} \neq \emptyset$  then
7    $\lambda \leftarrow \mathbb{C}$ 
8   Let  $C = (C_1, C_2) \in \mathbb{C}$  and  $n \in \mathbb{N}$ 
   so that  $n$  is maximal and
    $\#C(n) > 0$ ; in case of a tie,
   choose so that  $\#C(n)$  is minimal
9   Let  $P \in C_1(n)$ 
10   $L \leftarrow \{(P, Q) : Q \in C_2(n)\}$ 
11 else
12   $\lambda \leftarrow \emptyset$ 
13   $L \leftarrow \emptyset$ 
14 return  $\lambda, L$ 

```

Theoretical analysis

In spite of an intricate back and forth structure between nodes, bags and collections (notably through deductions rules), which makes it difficult to evaluate the complexity of our algorithm, we have a few comments to offer regarding this matter.

SPLITCHILDREN procedure Assuming $T_1 \simeq T_2$, and denoting by $n = \#T_1 = \#T_2$ and $d = \deg(T_1) = \deg(T_2)$, we have the following result.

Proposition 4.2 *The SPLITCHILDREN procedure, detailed in Algorithm 8 and Algorithm 9, has complexity $O(nd)$.*


Proof. Already, we can note that it is not necessary to scan all the bags and collections at each call: if this allows to lighten the pseudocode, the best way to do it, since we dispose of the function σ which associates to each node the bag or collection in which it is stored, is to scan only the bags and collections related to the sets C_u and C_v .

In particular, since the recursive calls are made on children, i.e. with increasing depth, the same bag or collection is processed only once throughout the calls – since the nodes have been partitioned by depth.

Splitting each bag or element of a collection into two is done in linear time – assuming that checking whether a node is present in C_u (or C_v) or not is done in constant time (e.g. with a hash table). In addition, it is necessary to pay twice the computation time of the children (one per recursive call) – this number is bounded by d per node.

We can then evaluate the total complexity of the algorithm (including recursive calls), assuming that all bags and collections are visited, to be

$$\text{in the order of } \left(\sum_{B \in \mathcal{B}} \#B + \sum_{C \in \mathcal{C}} \sum_{k \in \mathbb{N}} k \#C(k) \right) (2d + 1).$$

Since the bags and collections are composed of the nodes of T_1 and T_2 , the bracketed term can not be greater than n , concluding the proof. 


Preprocessing part Ignoring the possible recursive call to parents, the expected complexity for MAPNODES – Algorithm 7 – is roughly the same of SPLITCHILDREN. Since we can not map more nodes than there are in the trees, the number of calls to MAPNODES during preprocessing is bounded by the size of the trees, leading to a complexity for this part of $O(n^2d)$ – ignoring deductions and partitions of the nodes.

Backtracking part The underlying structure behind the backtracking step is a search tree – whose nodes we call states. In the best case, we simply traverse it from the root to a leaf; in the worst case, we explore the entirety of it. Ensuring that the tree has a minimal number of states is therefore of utmost importance to guarantee that the algorithm finishes rapidly, even in the worst case.

Theorem 4.3 *To minimize the size of the backtracking tree,*

- ▶ *one must process the bags before the collections;*
- ▶ *if there are only bags left, the bag B with the smallest cardinality $\#B$ must be processed first;*
- ▶ *if there are only collections left, the collection $C(n)$ with the largest n must be processed first; in case of a tie, process first the one with the smallest $\#C(n)$;*


where “process” $C(n)$ means to choose $P \in C_1(n)$ and $Q \in C_2(n)$ and put them in a bag.

Proof. The proof can be found in [Appendix A.4](#). 

Among all algorithms determining the order in which to process bags and collections based on their cardinality, for backtracking, [Algorithm 10](#) is optimal since it implements the strategy induced by the previous theorem.

To give an order of magnitude, we also have the following result (valid even without optimizing the processing order of the bags and collections), that illustrates the interest of preprocessing in reducing the size of the search space as much as possible.

Proposition 4.4 *The size of the backtracking tree is bounded by $2(e - 1)N(\mathbb{B}, \mathbb{C})$, with $N(\mathbb{B}, \mathbb{C})$ as defined in [Equation 4.2](#).*

Proof. The proof can also be found in [Appendix A.4](#). 

The leaves of the backtracking tree correspond to each of the possible mappings from (\mathbb{B}, \mathbb{C}) at the end of preprocessing; thus there are $N(\mathbb{B}, \mathbb{C})$ leaves. The previous result shows that the size of the tree is linear in the quantity of items that we want to enumerate. The constant, here $2(e - 1)$, expresses how parsimonious the approach is.

If we consider the enumeration tree of all permutations of $\llbracket 1, n \rrbracket$, whose number of leaves is $n!$, the size of this tree is bounded by $(e - 1)n!$ – see [\[75\]](#). If we take this constant, $e - 1$, as a baseline of what we can expect to obtain, our constant is only twice as large. Actually, [Proposition 4.4](#) being valid regardless of the order in which bags and collections are processed, the actual constant after optimization is likely to be smaller²⁰.

Note also that the complexity to explore each state depends on the state, via the possible deductions and calls to `SPLITCHILDREN` made.

In the rest of this section, we investigate the experimental behavior of our algorithm.

[75]: Knuth (2005), *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming)*

20: Note that it is the multiplicative factor before $e - 1$ that can be reduced – see the proof for complete details.

Experimental protocol

Given two labeled trees T_1 and T_2 assumed to be isomorphic up to a cipher, we expect that the complexity of establishing that $T_1 \sim T_2$ depends only on $\llbracket T_1 \rrbracket = \llbracket T_2 \rrbracket$. In particular, for a labeled tree T , the ratio $p = \frac{\#\mathcal{A}(T)}{\#T}$ is especially interesting to consider. Indeed, if $p = 1/\#T$ or $p = 1$, any tree isomorphism is also a tree ciphering – in which case the first isomorphism found is sufficient. If p takes intermediate values, the space of tree cipherings is strictly smaller than the one of tree isomorphisms – and one can not use just any one of them.

Remark 4.2 Note that we can identify if trees T_1 and T_2 are in one of these two special cases ($p = 1$ or $p = 1/\#T_i$) by a linear traversal of the nodes. Since the first isomorphism we find fits, we can easily construct one in linear time. We start by associating to each node its equivalence class – see [The Aho, Hopcroft & Ullman algorithm \(p. 13\)](#) – before mapping the roots. Then, for each pair of mapped nodes, for each equivalence class identified in their children, we map the n -th child of this class in T_1 with the n -th in T_2 , and so on recursively.

We shall see it later, but our algorithm obtains worse performances if we use it on trees in one of these two cases. So it seems better in this context to use this much simpler procedure instead.

21: We actually use $\max(\lfloor pn \rfloor, 1)$ in our implementation to avoid pathological cases.

With n and p fixed, we construct a tree T_1 with n nodes and a proportion of labels equal to p as follows. We draw uniformly at random $\lfloor pn \rfloor$ nodes²¹ and they receive the labels $1, \dots, \lfloor pn \rfloor$ in this order. The remaining nodes are randomly assigned a label between 1 and $\lfloor pn \rfloor$. For $p = 1/n$, all nodes receive instead the same label, and for $p = 1$, within a linear traversal of the trees, the i -th node visited receives label i . T_2 is then obtained as a copy of T_1 , where we shuffled the children of each node. In this case, $T_1 \sim T_2$.

To test the case $T_1 \approx T_2$, we created a copy T'_2 of T_2 , and for a randomly chosen node u in T'_2 , we replaced its label \bar{u} by another one, according to the following procedure. If each label appears only once in the tree – i.e., $p = 1$, then we select a node u at random and give it a label randomly drawn among $\mathcal{A}(T_2) \setminus \{\bar{u}\}$. Otherwise, we draw a node u at random among all the nodes whose label is present at least twice in the tree, and we give it a new label $\max(\mathcal{A}(T_2)) + 1$. In both cases, $T_1 \approx T_2'^{22}$ – notably because, trivially, $\#\mathcal{A}(T_1) \neq \#\mathcal{A}(T'_2)$.

22: Note that this change of a single label to break the isomorphism is made to make the task of detecting the non-isomorphism as difficult as possible.

We generated 100 couples of trees (T_1, T_2) and (T_1, T'_2) for each pair (n, p) , where $n \in \{25k : 2 \leq k \leq 10\}$ and $p \in \{1/n\} \cup \{k/10 : 1 \leq k \leq 10\}$ – so 9,900 trees for each case, $T_1 \sim T_2$ and $T_1 \approx T_2$.

The preprocessing breaks the cardinality of the search space

As stated in [On the cardinality of the search space \(p. 43\)](#), the size of the current search space $N(\mathbb{B}, \mathbb{C})$ is given by [Equation 4.2](#) and must be related to the size of $\text{Isom}(T_1, T_2)$, $N_{\approx}(T_1)$, given by [Equation 3.1](#). As already mentioned, the initial search space is that of bijections between the nodes of T_1 and T_2 , a far larger space than that of isomorphisms.

The goal of the different partitioning steps, presented in Section 4.3, is to break the cardinality of the space at each step, and reduce it enough so that the final size is less than $N_{\approx}(T_1)$. We can measure how far the current space is from this goal by calculating the logratio $r(\mathbb{B}, \mathbb{C})$, defined as

$$r(\mathbb{B}, \mathbb{C}) = \log_{10} \frac{N(\mathbb{B}, \mathbb{C})}{N_{\approx}(T_1)}. \quad (4.3)$$

The space is effectively reduced with respect to $\text{Isom}(T_1, T_2)$ if and only if $r(\mathbb{B}, \mathbb{C})$ is a negative number. Figure 4.10a shows the evolution of the ratio after each partitioning step. These steps appear to be sufficient to bring the ratio close to – or below 0. In more details, if we denote by $r_{\text{final}}(\mathbb{B}, \mathbb{C})$ the ratio after the Labels step of preprocessing, Figure 4.10b provide a closer look at the results. We see that excluding the cases $p = 1$ and $p = 1/n$, the space is almost systematically reduced (except for a few pathological cases when p is close to 1), with a linear tendency that seems to emerge for values of $p < 0.5$ – meaning that the search space is exponentially more reduced for larger trees.

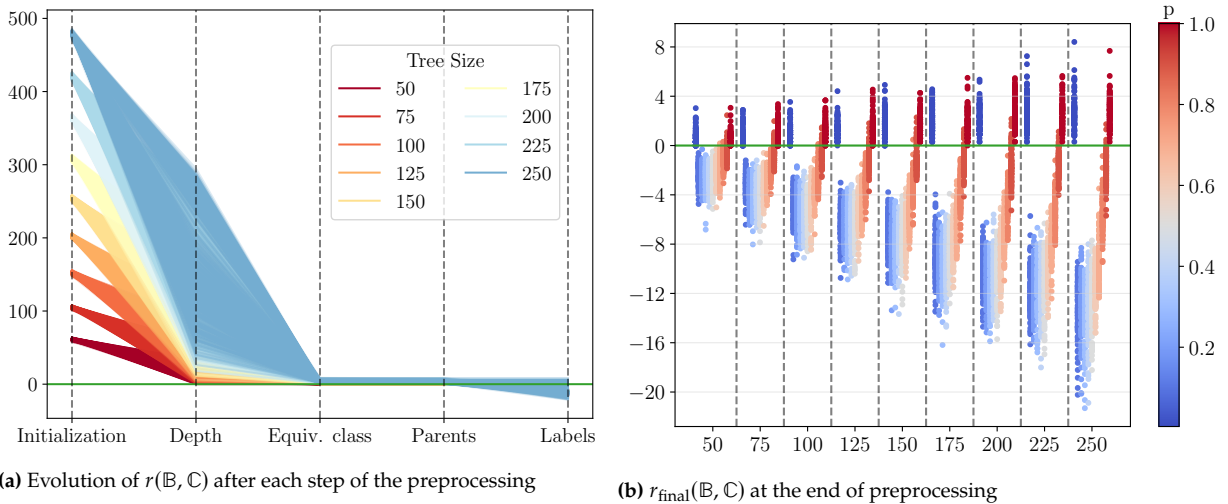


Figure 4.10: $r(\mathbb{B}, \mathbb{C})$ at different steps of the preprocessing, with regard to the size and proportion of labels of trees, with $T_1 \sim T_2$.

When $p = 1/n$, the unique label is mapped as early as the roots, and thus future deductions rely only on the topology of the tree. The collections step does not bring any additional deduction – and we can see on Figure 4.10a that it is crucial to effectively reduce the search space. Similarly, when $p = 1$, except for mandatory mappings from the topology, the collections step produces few deductions since all labels are different; moreover those collections remain and contribute for much to the size of the space.

In Figure 4.11, one can see the proportion of nodes and labels mapped at the end of the preprocessing. We observe the same trends as before, especially for the $p < 0.5$ cases. Note that in a number of cases, the proportion of mapped nodes reaches 1; in other words, preprocessing alone was sufficient to find a tree ciphering – which explain some of the low values of r_{final} in Figure 4.10b.

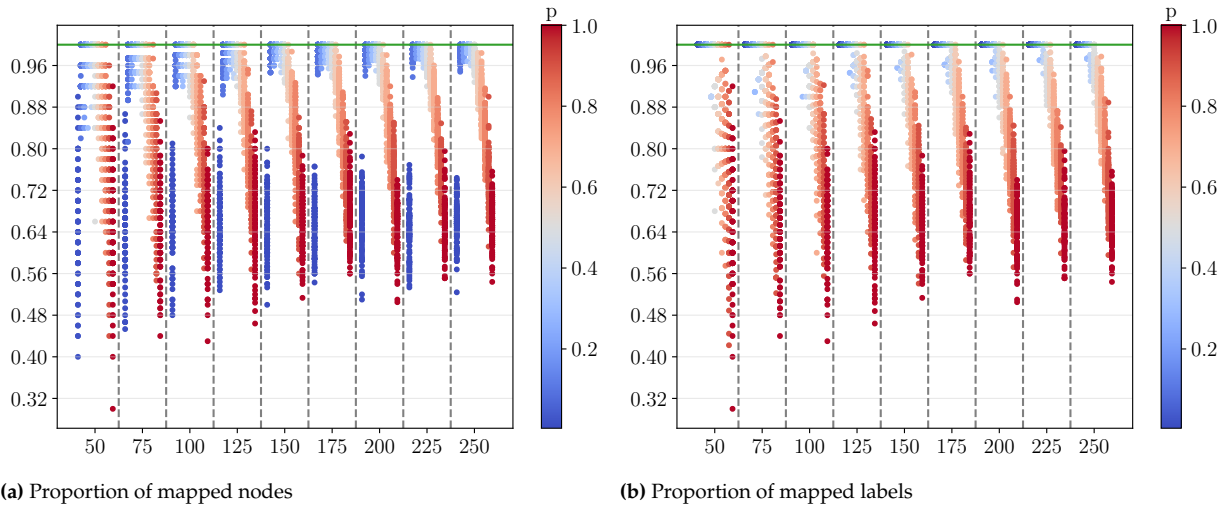


Figure 4.11: Proportion of mappings at the end of preprocessing, with regard to the size and proportion of labels of trees, with $T_1 \sim T_2$.

Computation time repartition between backtracking and preprocessing

The goal of preprocessing, by breaking as much as possible the cardinality of the search space, is to give a chance to the backtracking to converge in reasonable time. The results for $T_1 \sim T_2$ are gathered in Figure 4.12.

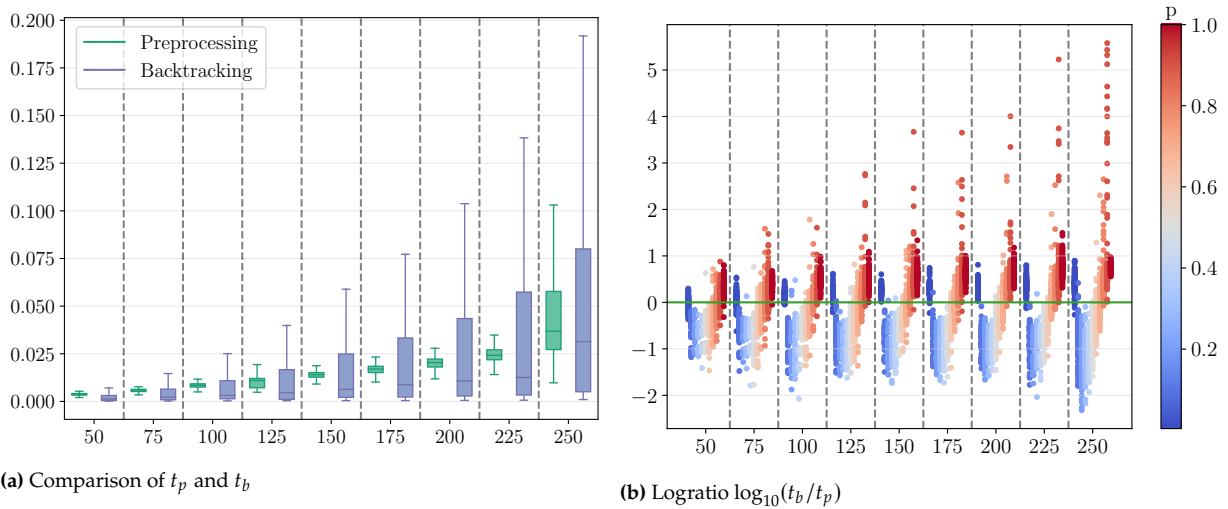


Figure 4.12: Computation time spent in the preprocessing part t_p and the backtracking part t_b , in the case where $T_1 \sim T_2$. In Figure 4.12a, the outliers are not shown for the sake of readability. On the other hand, they are present on Figure 4.12b.

As can be seen on Figure 4.12a – where outliers are not showed, the time t_p spent in preprocessing seems to grows linearly with tree size, with little variability, while the time t_b spent in backtracking seems much more volatile. We did not find any significant influence of the p parameter on the time spent in preprocessing.

Nevertheless, a closer look – see Figure 4.12b – shows that backtracking is systematically faster than preprocessing when p takes intermediate values, and slower for small or large values of p . Outliers, with backtracking times several orders of magnitude higher than preprocessing time,

only occur for large values of p (but strictly < 1). This phenomenon can be explained as follows. Let us say that all the labels are different, except for one that is present in two copies (this is one of the worst possible cases). If this duplicate is not detected until too late in the backtracking, it is easy to imagine that the algorithm will take much longer to complete – since it will have to backtrack a long way to correct its mistake.

The cases $p < 0.5$ that reach very low ratios can be explained in light of [Figure 4.11a](#), where we observe that almost all nodes are mapped at the end of preprocessing. The time spent backtracking is then very short (or even negligible, if there are no nodes left to map) – as seen on [Figure 4.12a](#), where the bottom whisker of the boxplot touches 0.

The cases $p = 1$ and $p = 1/n$ are surprising since these are cases where any tree ciphering works, i.e. the algorithm is not expected to backtrack. On the other hand, these are also the cases where the search space is the largest when the backtracking part starts. This explanation seems sufficient by virtue of [Figure 4.13](#), which shows (at least in the $p = 1$ and $p = 1/n$ cases) that the time spent backtracking is related to the size of the space at the start of the backtracking, in the following manner:

$$\log_{10}(t_b) \approx 0.72 \log_{10}(N(\mathbb{B}, \mathbb{C}))^{0.41} - 3.09;$$

where the values of the parameters have been estimated using least squares method. It seems that t_b tends, on average, to grow sublinearly as a function of $N(\mathbb{B}, \mathbb{C})$ – to relate with [Proposition 4.4](#).

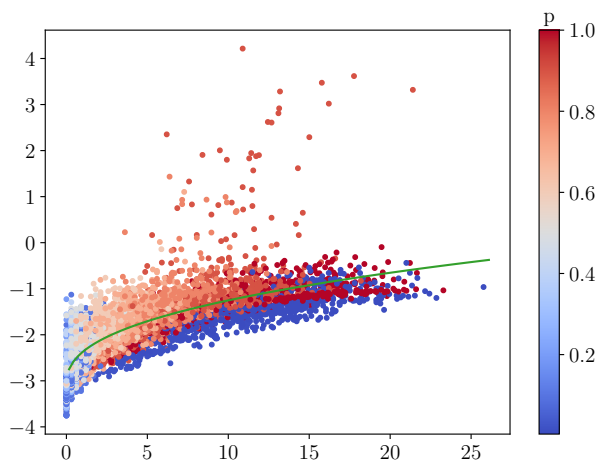


Figure 4.13: $\log_{10}(t_b)$ as a function of $\log_{10}(N(\mathbb{B}, \mathbb{C}))$ – evaluated at the beginning of backtracking. The trend is interpolated by the green curve, whose equation is given by $y = 0.72x^{0.41} - 3.09$.

When trees are not isomorphic up to a cipher

In the case where the considered trees T_1 and T_2 are not isomorphic, one of the following two outcomes is expected:

- ▶ Either we detect during the preprocessing that $T_1 \approx T_2$, in which case we expect to obtain a faster answer than when $T_1 \sim T_2$;
- ▶ either the answer is obtained during backtracking, in which case we expect the backtracking tree to be explored more extensively than in case $T_1 \sim T_2$, and the answer will be slower to obtain.

23: Note that we use here smaller trees than before, with $n \in \{10k : 2 \leq k \leq 10\}$, due to computation time constraints – in light of [Figure 4.14b](#).

With couples (T_1, T_2) and (T_1, T'_2) constructed as in [Experimental protocol \(p. 52\)](#)²³, we compared the time t_\top to decide $T_1 \sim T_2$ with the time t_\perp to decide $T_1 \sim T'_2$. We actually plot, in [Figure 4.14](#), $\log_{10} \frac{t_\perp}{t_\top}$, so that t_\perp is greater than t_\top if and only if this number is positive.

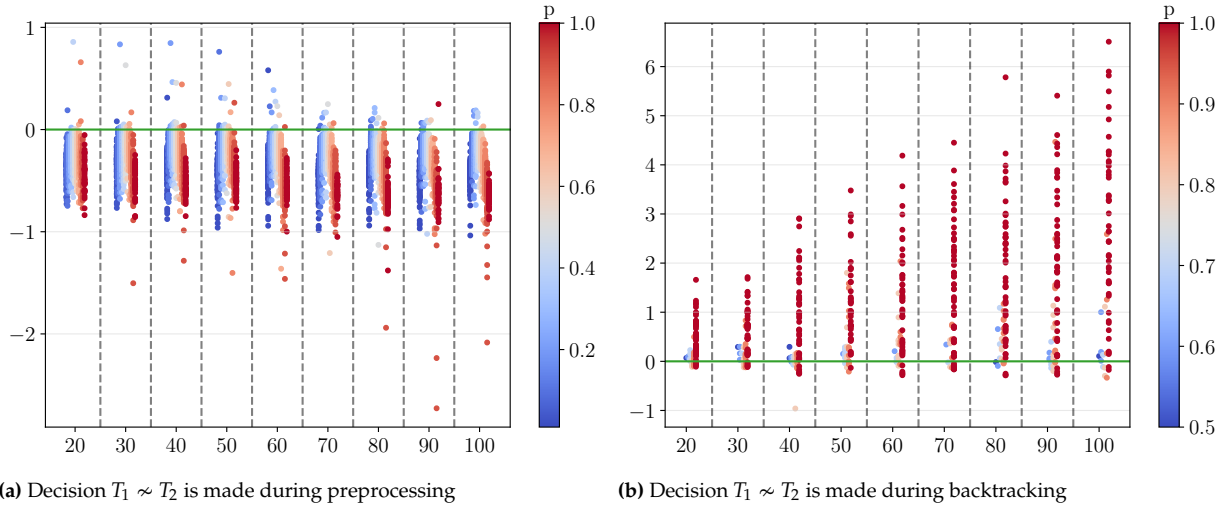


Figure 4.14: Logratio $\log_{10}(t_\perp/t_\top)$ with regard to the size and proportion of labels of trees.

In [Figure 4.14a](#), we see all the instances where we could determine $T_1 \sim T'_2$ during the preprocessing. In most cases, and seemingly independently of the parameters s and p , we were able to obtain an answer more quickly than for deciding $T_1 \sim T_2$. These instances represent 93% of all the instances tested.

The remaining 7% of instances are shown in [Figure 4.14b](#), where the decision $T_1 \sim T'_2$ was obtained during the second phase of the algorithm, i.e. the backtracking. We notice in this case that the decision is generally slower to obtain than $T_1 \sim T_2$, and particularly when $p = 1$ ²⁴, where the decision can be several orders of magnitude slower.

24: Recall, however, that in T'_2 , all the labels are different, except for one which is present in double. The proportion p shown in the figure is that of T_1 before modification.

ENUMERATION TREES: FROM TREES TO FORESTS

From tree to forest enumeration

5

A single tree is a forest of one.

Don Rittner

In this chapter, we focus on enumeration problems related to trees. First, we present in [Section 5.1](#) a generic method to enumerate structured objects, called reverse search. In particular, this method is an effective tool to solve an important data mining problem, named frequent pattern mining. The goal is to find shared denominators between data in the form of common substructures that appear with a certain minimum frequency among the items in a database.

In a second step, we look at the results of the literature. [Section 5.2](#) presents the existing enumeration algorithms for unlabeled trees, ordered or not. In particular, the concept of canonical representation of an unordered tree is introduced, consistent with the idea of parsimonious enumeration. Indeed, we want to enumerate each tree only once; but since the order of the children does not matter in unordered trees, we have to make sure that we do not obtain the same tree in duplicate, where nodes have been swapped. The strategy is therefore to impose a systematic order on the children of a node, i.e. to order the unordered trees via a systematic method, and then to enumerate only the canonically ordered trees.

Finally, in [Section 5.3](#), we introduce the problem we are interested in; the enumeration of irredundant forests, i.e. forests without repetition. The trees in these forests are unordered, and so is the forest itself. The purpose of this section is therefore to construct a canonical representation of these forests, in a similar way as trees seen previously. We show in [Theorem 5.2](#) that irredundant forests are in bijection with a certain class of DAGs – for which we construct a canonical form. Their enumeration will then be discussed in upcoming [Chapter 6](#).

Parts of this chapter are reproduced from [\[32\]](#), notably [Section 5.3](#).

5.1 Enumeration problems . . .	59
Reverse search technique . .	60
Frequent pattern mining . . .	61
5.2 Tree enumeration	62
Ordered trees	63
Unordered trees	63
5.3 Forest enumeration	64
Irredundant forests	65
A detour through formal lan-	
guages	66
Canonical FDAGs	67

5.1 Enumeration problems

Enumeration problems are recurrent in many fields, notably combinatorial optimization and data mining. They involve the exhaustive listing of a subset of the elements of a search set (possibly all of them), e.g. graphs, trees or vertices of a simplex. Given the possibly high combinatorial nature of these elements, it is essential to adopt clever exploration strategies as opposed to brute-force enumeration, typically to avoid areas of the search set not belonging to the objective subset. Another extremely important point is to perform parsimonious enumeration, in the sense that each element of the objective subset is seen only once.

[32]: Ingels et al. (2022), ‘Enumeration of Irredundant Forests’

[76]: Land et al. (2010), ‘An automatic method for solving discrete programming problems’

[77]: Yan et al. (2002), ‘gspan: Graph-based substructure pattern mining’

[28]: Avis et al. (1996), ‘Reverse search for enumeration’

[78]: Yamazaki et al. (2020), ‘Enumeration of nonisomorphic interval graphs and nonisomorphic permutation graphs’

[79]: Nowozin (2009), ‘Learning with structured data: applications to computer vision.’

One proven way of proceeding is to provide the search set with an enumeration tree structure; starting from the root, the branches of the tree are explored recursively, eliminating those that do not address the problem, and visiting each element only once. Based on this principle, we can notably mention the well-known “branch and bound” method in combinatorial optimization [76] and the `GSPAN` algorithm for frequent subgraph mining in data mining [77]. Another of these methods is the so-called reverse search technique, which requires that the search set has a partial order structure, and which has solved a large number of enumeration problems since its introduction [28] until recently [78].

Reverse search technique

In this thesis, we restrict ourselves to reverse search methods. While first introduced by Avis and Fukuda [28], the formalism adopted here is derived from more recent work [79].

Let (\mathbb{S}, \subseteq) be a partially ordered set, with a unique least element $\emptyset \in \mathbb{S}$, such that $\forall s \in \mathbb{S}, \emptyset \subseteq s$. Let $g : \mathbb{S} \rightarrow \{\top, \perp\}$ be a *property*, satisfying anti-monotonicity

$$\forall s, t \in \mathbb{S} : (s \subseteq t) \wedge g(t) \implies g(s). \quad (5.1)$$

The *enumeration problem* for the property g is the problem of listing all elements of $E_{\mathbb{S}}(g) = \{s \in \mathbb{S} : g(s) = \top\}$. An enumeration algorithm is an algorithm that returns $E_{\mathbb{S}}(g)$.

The reverse search technique relies on inverting a *reduction rule*, defined as follows.

Definition 5.1 $f : \mathbb{S} \setminus \emptyset \rightarrow \mathbb{S}$ is a reduction rule if and only if

- ▶ $\forall s \in \mathbb{S} \setminus \emptyset, f(s) \subseteq s$ and $\nexists u \in \mathbb{S} \setminus \{s, f(s)\}, (f(s) \subseteq u) \wedge (u \subseteq s)$;
- ▶ $\forall s \in \mathbb{S} \setminus \emptyset, \exists k \in \mathbb{N}^*, f^k(s) = \emptyset$.

In other words, $f(s)$ reduces s in a minimal way – by the first property – and reaches the “smallest” element of \mathbb{S} , \emptyset , in a finite number of steps. Then, the *expansion rule* is defined as $f^{-1}(t) = \{s \in \mathbb{S} : f(s) = t\}$. f defines an enumeration tree rooted in \emptyset , and repeated calls to f^{-1} can therefore enumerate all the elements of \mathbb{S} .

The reverse search algorithm is shown in [Algorithm 12](#). $E_{\mathbb{S}}(g)$ can be obtained from the call of `REVERSESEARCH` $((\mathbb{S}, \subseteq), f^{-1}, g, \emptyset)$. As g is anti-monotone, if $g(s) = \perp$, then all elements $s \subseteq t$ also have $g(t) = \perp$, and therefore pruning the enumeration tree in s does not miss any element of $E_{\mathbb{S}}(g)$.

When successfully designed, a reverse search technique should yield polynomial output delay [28, 80], i.e., the time between the output of one element and the next is bounded by a polynomial function in the size of the input.

Remark 5.1 It would have been possible to define directly the set \mathbb{S}

Algorithm 12: REVERSESEARCH

Input: $(\mathbb{S}, \subseteq), f^{-1}, g, s_0 \in \mathbb{S}$ – such that $g(s_0) = \top$

```

1 output  $s_0$ 
2 for  $t \in \{s \in f^{-1}(s_0) \mid g(s) = \top\}$  do
3   REVERSESEARCH $((\mathbb{S}, \subseteq), f^{-1}, g, t)$ 

```

[80]: Johnson et al. (1988), ‘On generating all maximal independent sets’

[28]: Avis et al. (1996), ‘Reverse search for enumeration’

as the set of elements verifying the property g . Separating the two induces that the reduction rule f formally depends only on \mathbb{S} , and not on g . This allows, once f is constructed once and for all, to filter \mathbb{S} according to various properties g without much additional work – provided they indeed verify Equation 5.1. In particular, this is useful in the case where g depends on a tunable parameter – as in the problem introduced below.

Enumeration of permutations We now illustrate the reverse search process on an example. Suppose we want to enumerate the permutations of $\llbracket 1, n \rrbracket$. Note first that any permutation π can be written as a n -digit number²⁵; for example the number 321 is read as $\pi(1) = 3$, $\pi(2) = 2$ and $\pi(3) = 1$. We denote by $N(\pi)$ this number.

As permutations can be expressed as a composition of transpositions (i.e. a swap of two elements), we can define the following order relation on permutations: $\pi \subset \pi' \iff N(\pi) < N(\pi')$, and π, π' only differ by a transposition.

Partially ordered sets are traditionally represented by Hasse diagrams [81, 82], i.e. DAGs where an arc $x \rightarrow y$ reflects $y \subset x$ – see Figure 5.1a for $n = 3$.

Given a permutation $\pi = p_1 \cdots p_n$, we look for the first index i such that $p_i > p_{i+1}$; we construct $f(\pi)$ by swapping p_i and p_{i+1} ; this is our reduction rule [83]. We briefly confirm that it meets the requirements of Definition 5.1. Naturally, $f(\pi) \subset \pi$. Since two permutations are related if they differ only by one transposition, we can not find a third permutation π' such that $f(\pi) \subset \pi' \subset \pi$. Then, since $f(\pi) < \pi$ by construction, a finite number of swaps allows to reach the identity permutation – the root of the enumeration. The enumeration tree obtained is presented, in the case $n = 3$, on Figure 5.1b.

The reduction rule still needs to be inverted to complete the reverse search scheme. The main difficulty is to find the right pairs of indices to swap (for example, we can not reconstruct 321 from 123). We refer the reader to [83] for complete details.

It is important to note that the main issue with reverse search methods is to find a reduction rule that is “properly” invertible – in other words, the construction of the expansion rule is a key factor of the technique.

Frequent pattern mining

With the ever-increasing accumulation of data, especially structured data, being able to extract information efficiently is a real challenge. The *frequent pattern mining problem* consists in finding patterns (typically, substructures for structured data) that appear with a certain minimum frequency in the items of a database.

A typical example might be the design of new drugs, where molecules are seen as graphs, with each vertex corresponding to an atom. The expectation is that molecules with similar behavior to treat certain diseases will be made up of similar chemical compounds. Therefore, it becomes crucial to identify these common chemical compounds [84].

25: If $n > 9$, one must choose a suitable base to represent the numbers.

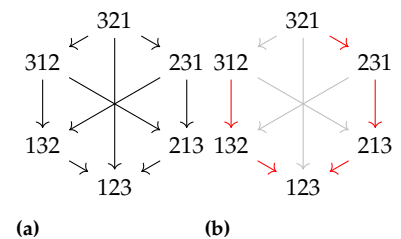


Figure 5.1: An illustration of the reverse search method on permutations. (a) depicts the relations between permutations with $n = 3$; (b) shows the enumeration tree.

[81]: Di Battista et al. (1988), ‘Algorithms for plane representations of acyclic digraphs’

[82]: Freese (2004), ‘Automated lattice drawing’

[83]: Yamanaka (2016), ‘Permutation Enumeration’

[84]: Deshpande et al. (2005), ‘Mining Chemical Compounds’

[77]: Yan et al. (2002), ‘gspan: Graph-based substructure pattern mining’

26: Note that both the elements of \mathcal{X} and the substructures investigated belong to the same space \mathbb{S} . Think of graphs and subgraphs, strings and substrings, etc. There is a kind of recursivity in the considered objects.

[79]: Nowozin (2009), ‘Learning with structured data: applications to computer vision.’

The gSPAN algorithm, dedicated to frequent subgraph mining, is notably applied to chemical data in the original paper [77] – and uses a reverse search approach.

Formally, the frequent pattern mining problem can be formulated as follows: from a dataset $\mathcal{X} = \{s_1, \dots, s_n\}$ with $s_i \in \mathbb{S}$, and a fixed threshold σ , find all elements²⁶ $s \in \mathbb{S}$ that satisfy $\text{freq}(s, \mathcal{X}) \geq \sigma$, where $\text{freq}(s, \mathcal{X})$ is a function that counts the frequency of appearance of s in the dataset \mathcal{X} . Typically,

$$\text{freq}(s, \mathcal{X}) = \frac{1}{n} \sum_{i=1}^n \mathbb{1}_{s \subseteq s_i}. \quad (5.2)$$

Another option may be to consider the multiplicity of each substructure, replacing $\mathbb{1}_{s \subseteq s_i}$ by $N_s(s_i)$, where $N_s(t)$ counts how many times s appears in t .

One way to address frequent pattern mining is precisely based on a reverse search approach. Indeed, if we already have a way to enumerate the elements of \mathbb{S} , it is sufficient to filter the enumeration according to the following property $g(s, \mathcal{X}, \sigma) = (\text{freq}(s, \mathcal{X}) \geq \sigma)$, which is anti-monotone by virtue of the following lemma – the proof can be found in [79].

Lemma 5.1 $\forall s, t \in \mathbb{S}, s \subseteq t \implies \text{freq}(s, \mathcal{X}) \geq \text{freq}(t, \mathcal{X})$.

We emphasize here that each possible definition of “ s appears in X ” leads to a different data mining problem. The choice of this definition is therefore of prime importance. In particular, this choice should induce a way of calculating $\text{freq}(s, \mathcal{X})$ that reflects the specificity of the reduction rule f , so that $\{s \in f^{-1}(s_0) : g(s) = \top\}$ can be constructed directly, instead of first generating $f^{-1}(s_0)$ and then filtering according to the value of g . Indeed, if g is too restrictive, and $f^{-1}(s_0)$ too large, one would have to enumerate objects that are not relevant to the enumeration problem, which is not desirable.

We showed that the problem of frequent pattern mining can be solved as soon as we dispose of a reverse search enumeration method of the patterns of interest. We are now interested in specific cases; and we start with trees.

5.2 Tree enumeration

This section has a twofold interest: first, it presents existing results in the literature concerning the enumeration of unlabeled trees, ordered or not. Then, through these examples, it introduces a number of practical considerations about the design of reverse search algorithms, which will be exploited to design our forest enumeration algorithm – see [Section 5.3](#) and [Chapter 6](#).

Ordered trees

The enumeration of ordered trees has been proposed in [29], where they are referred to as plane trees. The reverse search procedure they use is defined as follows.

Given an ordered tree T and a node $v \in T$, they call *rightmost child* the last element of $C(v)$. The *rightmost path* of T , denoted $RP(T)$, is the sequence starting with $\mathcal{R}(T)$, its rightmost child, their rightmost child, and recursively until a leaf is reached. The reduction rule f for ordered trees is such that $f(T)$ is the tree T deprived from the leaf of $RP(T)$.

The expansion rule $f^{-1}(T)$ constructs all trees such that, if the last element from their rightmost path is removed, we get T . These trees are exactly those obtained by adding a node as the rightmost child of each of the nodes composing $RP(T)$. An example is shown in Figure 5.2.

On the enumeration of ordered trees, let us also mention this recent work [85], which does not use a reverse search approach.

Unordered trees

As can be seen in Figure 5.3, a single unordered tree can have several representations. The additional difficulty that arises when going from ordered to unordered is exactly this: how to avoid enumerating the same tree under different representations? Considering that the number of representations is expressed as a product of factorials (since permutations of sibling nodes with different equivalence class are involved²⁷), addressing this issue is critical.

This question has been answered in [30], where the idea is to define the so-called canonical form of an unordered tree – i.e. a systematic way of representating trees – and then to enumerate only canonical trees. Note that since the set we aim to enumerate is that of canonical trees, it is imperative that the reduction rule (and thus, the expansion rule) operates on canonical trees. In particular, special care must be taken to ensure that a reduced (or expanded) tree is still canonical after operation. Below is the method used in [30].

Given a tree T , number its nodes v_1, \dots, v_n by depth-first search. The *depth sequence* of T is defined as the sequence $L(T) = (\text{depth}(v_1), \dots, \text{depth}(v_n))$. Given two trees T_1 and T_2 , the depth sequence $L(T_1)$ is said to be *heavier* than $L(T_2)$ if and only if $L(T_1) >_{\text{lex}} L(T_2)$, where $>_{\text{lex}}$ is the lexicographic order²⁸. The *left-heavy embedding* of T is the tree \hat{T} such that $T \simeq \hat{T}$, and $L(\hat{T})$ is the heaviest depth sequence among all trees isomorphic to T . An example is provided in Figure 5.4.

The canonical form of a tree T is defined in [30] as its left-heavy embedding \hat{T} . Remember that the interest of this form is to fix the representation; only the canonical trees are then enumerated. The reduction rule employed is the same as the one previously used for ordered trees in [29]: remove the leaf from the rightmost path of \hat{T} . Removing this leaf implies that the depth sequence of the new tree $f(\hat{T})$ is the same as \hat{T} , deprived of its last element. Since $L(\hat{T})$ is maximal, so is $L(f(\hat{T}))$; therefore $f(\hat{T}) = \widehat{f(\hat{T})}$ and the resulting tree is still canonical.

[29]: Nakano (2002), ‘Efficient generation of plane trees’

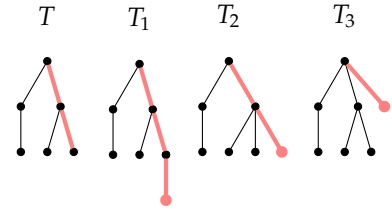


Figure 5.2: A tree T (left) and its 3 successors T_1, T_2 and T_3 obtained via the expansion rule for ordered trees. The rightmost paths are showed in red; and the new nodes in red as well.

[85]: Parque et al. (2021), ‘An Efficient Scheme for the Generation of Ordered Trees in Constant Amortized Time’

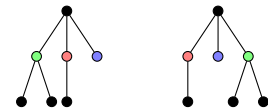


Figure 5.3: Two different representations of the same unordered tree. There are as many as possible permutations of (●, ●, ●), i.e. 6.

27: Not to be mistaken with the number of isomorphisms which involves the permutations of sibling nodes with the same equivalence class; see [On the number of tree isomorphisms](#) (p. 25).

[30]: Nakano et al. (2003), ‘Efficient generation of rooted trees’

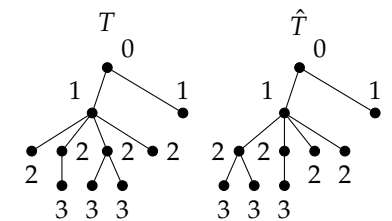


Figure 5.4: A tree T (left) and its left-heavy embedding \hat{T} (right). Depths are indicated besides each node. We have $L(T) = (0, 1, 2, 2, 3, 2, 3, 3, 2, 1)$ and $L(\hat{T}) = (0, 1, 2, 3, 3, 2, 3, 2, 2, 1)$.

28: With $a_1 \dots a_k >_{\text{lex}} b_1 \dots b_k$ if and only if either $a_1 > b_1$ or $a_1 = b_1$ and $a_2 \dots a_k >_{\text{lex}} b_2 \dots b_k$. See also [A detour through formal languages](#) (p. 66).

[29]: Nakano (2002), ‘Efficient generation of plane trees’

[80]: Johnson et al. (1988), ‘On generating all maximal independent sets’

[27]: Asai et al. (2003), ‘Discovering frequent substructures in large unordered trees’

On the other hand, and without going into details, the expansion rule differs from that of ordered trees. If we add a node along the rightmost path, not all choices generate a tree maximizing its depth sequence – and thus some are not canonical. This issue is managed in [30], and the final algorithm they propose yields polynomial delay [80]. Precisely, each element is enumerated in $O(1)$ time, making the enumeration very efficient. Finally, it should be noted that since these algorithms explicitly build their enumeration tree, each element is only enumerated incrementally, i.e. the algorithm does not output entire trees but only the difference from the previous tree.

The algorithm for enumerating unordered trees is notably reused in [27] to address the problem of frequent pattern mining on labeled unordered trees – where tree isomorphisms require equality of the labels.

5.3 Forest enumeration

The enumeration of unordered trees was discussed in the previous section. On the other hand, the DAG compression presented in [Section 2.3](#) can be seen as a method for enumerating subtrees. Our ambition here is to take these two problems of enumeration – trees and subtrees – to a higher order, i.e. to enumerate sets of trees instead of singletons – forests.

On the other hand, forests, as introduced in [Definition 2.2](#), can have repetitions – it is possible for one tree to be a subtree of another (this includes being a copy of another tree). In a parsimonious enumeration approach, this is problematic. This is also an issue in the context of frequent pattern mining: if one looks at the frequency of appearance of a given subtree (assumed to belong to a forest of subtrees), one does not want to see this subtree appearing several times.

This is why we restrict our enumeration to *irredundant* forests, i.e. forests that contains no repetition, in the sense that no tree is a subtree of another. This condition is not restrictive since one can always introduce repetition afterwards.

We are therefore interested in the problem of enumerating irredundant forests of unordered trees, and then, given a tree or forest, to enumerate all its “subforests” – as forests of subtrees. The latter has already been discussed in the literature, but without consideration on isomorphism [86]. We emphasize that we aim to enumerate these various items – forests and subforests – up to isomorphism.

[86]: Schwikowski et al. (2002), ‘On enumerating all minimal solutions of feedback problems’

Such an ambition immediately raises a number of obstacles. First of all, the trees are indeed unordered, but so are the sets of trees. For the former the literature has introduced the notion of the canonical form of a tree, as seen previously. The enumeration therefore focuses only on these canonical trees. Unfortunately, if it is possible to order a set of vertices, there is no total order on the set of trees, to the best of our knowledge. In addition, the condition of non-repetition filters the set of forests in a non-trivial way, making, a priori, the enumeration problem trickier.

In the following, we formally introduce the problem and the measures adopted to address these difficulties.

Irredundant forests

We start by defining irredundant forests.

Definition 5.2 A set $\{T_1, \dots, T_n\}$ of trees is an irredundant forest if and only if

$$\forall i \neq j, T_i \notin \mathcal{S}(T_j). \quad (5.3)$$

In other words, no tree in the forest can be a subtree of another (a fortiori, no two trees can be identical either). We denote by \mathcal{F} the set of all irredundant forests.

Our goal is to provide a reverse search method that outputs \mathcal{F} . As already stated, this goal raises two major difficulties: firstly, the twofold unordered nature of forests (the set of trees and the trees themselves), and secondly, the non-trivial condition of non-repetition. Concerning the latter, we already know a method to eliminate repetitions in a tree or a forest: DAG reduction. By resorting to this technique, we can handle the former as well.

We saw in Section 2.4 that we could compress a forest into a DAG without loss by placing all the trees of the forest under an artificial root. Actually, for irredundant forests, we do not require this artificial root and instead proceed as follows.

Let $F = \{T_1, \dots, T_n\}$ be an irredundant forest. We construct its DAG compression $\mathfrak{R}(F)$ by using a simple adaptation of Algorithm 3: on Line 4, replace $\mathcal{H}(T)$ by $\max_i \mathcal{H}(T_i)$; and on Line 5, $v \in T^h$ by $v \in \cup_i T_i^h$, with the convention that $T^h = \emptyset$ if $h > \mathcal{H}(T)$. DAG compression of irredundant forests is illustrated on Figure 5.5.

If ever F was not an irredundant forest, say if for example $T_1 \in \mathcal{S}(T_2)$, then $\mathfrak{R}(T_1)$ would be a subDAG of $\mathfrak{R}(T_2)$; and therefore the number of roots in $\mathfrak{R}(F)$ would be strictly less than $\#F$. Since such a situation can not occur, there are exactly as many roots in $\mathfrak{R}(F)$ as there are elements in F : no information is lost. In other words, F can be reconstructed from $\mathfrak{R}(F)$ and DAG compression is also lossless in this case. Since D compresses several trees, D has several roots, and the notation $\mathcal{R}(D)$ therefore refers to the set of roots of D .

In the sequel, DAGs compressing irredundant forests are called *FDAGs*, to distinguish them from general directed acyclic graphs. Since DAG compression is lossless, and since an irredundant forest can be reconstructed from its DAG reduction, we have the following result.

Theorem 5.2 There is a bijection between the set of FDAGs and the set of irredundant forests.

Therefore, the problem of enumerating irredundant forests is equivalent to the problem of enumerating FDAGs. This compression of irredundant forests into FDAGs has the merit of transforming set of trees into unique objects, which makes it possible, if able to design a canonical representation – like the trees in [27, 30], to get rid of the twofold unordered nature of forests, as claimed earlier. Indeed, any ordering of the vertices of the

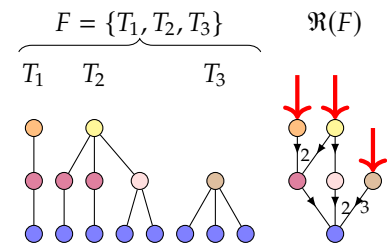


Figure 5.5: An unordered irredundant forest F (left) and its DAG compression $\mathfrak{R}(F)$ (right). Nodes are colored according to their class of equivalence. The roots of the DAG (indicated by red arrows) correspond exactly to the roots of the trees in F . For the sake of clarity, arcs of multiplicity greater than one are drawn only once and their multiplicity is written next to the arc.

[30]: Nakano et al. (2003), ‘Efficient generation of rooted trees’

[27]: Asai et al. (2003), ‘Discovering frequent substructures in large unordered trees’

DAG induces an order on the roots of the DAG, and therefore on the elements of the forest, as well on the vertices of the trees themselves.

The sequel of this chapter is dedicated to defining such canonical representation. But before getting there, we need to introduce some results on formal languages which will be necessary for the sequel of this chapter and [Chapter 6](#).

A detour through formal languages

Let \mathbb{A} be a totally ordered finite set, called alphabet, whose elements are called letters. A word is a finite sequence of letters of \mathbb{A} . The length of a word w is equal to its number of letters and is denoted by $\#w$. There is a unique word with no letter called the empty word and denoted by ϵ . The set of all words is denoted by \mathbb{A}^* . Words can be concatenated to create a new word whose length is the sum of the lengths of the original words; ϵ is the neutral element of this concatenation operation.

The lexicographic order over \mathbb{A}^* , denoted by $<_{\text{lex}}$, is defined as follows. Let $w_1 = a_0 \cdots a_p$ and $w_2 = b_0 \cdots b_q$ be two words, with $a_i, b_j \in \mathbb{A}$. If $\#w_1 = \#w_2$, then $w_1 <_{\text{lex}} w_2$ if and only if $\exists k \in \llbracket 0, p \rrbracket, a_i = b_i \forall i < k$ and $a_k < b_k$. Otherwise, let $m = \min(p, q)$; $w_1 <_{\text{lex}} w_2$ if and only if either (i) $a_0 \cdots a_m <_{\text{lex}} b_0 \cdots b_m$ or (ii) $a_0 \cdots a_m =_{\text{lex}} b_0 \cdots b_m$ and $m < q$ – that is, $p < q$. Note that, by convention, $\epsilon <_{\text{lex}} w$ for any word w .

Let $w \in \mathbb{A}^*$. We define the suffix-cut operator $SC(w)$, which removes the last letter of w :

$$SC(w) = \begin{cases} w' & \text{if } w = w'a \text{ with } a \in \mathbb{A} \text{ and } w' \in \mathbb{A}^*, \\ \epsilon & \text{otherwise.} \end{cases} \quad (5.4)$$

A *language* is a set of words satisfying some construction rules. We introduce hereafter two languages that will be useful in the sequel of this chapter and the next.

Definition 5.3 *The language of decreasing words is defined as*

$$\Lambda = \{w = a_0 \cdots a_m \in \mathbb{A}^* : a_i \geq_{\text{lex}} a_{i+1} \forall i \in \llbracket 0, m-1 \rrbracket\}.$$

Definition 5.4 *Let $\bar{w} \in \Lambda$. The language of decreasing words bounded by \bar{w} is defined as*

$$\Lambda^{\bar{w}} = \{w \in \Lambda : w >_{\text{lex}} \bar{w}\}.$$

Any word $w \in \Lambda^{\bar{w}}$ is said to be minimal if and only if $w \in \Lambda^{\bar{w}}$ but $SC(w) \notin \Lambda^{\bar{w}}$.

As an example, if $\mathbb{A} = \{0, 1, 2, 3\}$, then $\bar{w} = 211 \in \Lambda$, whereas $121 \notin \Lambda$. In addition, $\Lambda^{\bar{w}}$ contains words such as 31, 22, 21110, etc. 22 is a minimal word of $\Lambda^{\bar{w}}$ as $22 >_{\text{lex}} 211$ but $SC(22) = 2 <_{\text{lex}} 211$.

Our focus is now on the construction of the minimal words of $\Lambda^{\bar{w}}$. Let $\bar{w} = a_0 \cdots a_p$ and $w = b_0 \cdots b_q \in \Lambda^{\bar{w}}$. Taking into account that $w >_{\text{lex}} \bar{w}$ and that they both are decreasing words, there are only two possibles cases

- ▶ w and \bar{w} share a common prefix $a_0 \cdots a_m$ – i.e. $w = a_0 \cdots a_m b_{m+1} \cdots b_q$, and the word $a_0 \cdots a_m b_{m+1}$ is minimal by applying successive suffix-cut operations;
- ▶ w and \bar{w} do not share a common prefix. Necessarily $b_0 >_{\text{lex.}} a_0$, and then the word b_0 is minimal by applying several suffix-cut operations.

From the above, we deduce a method for constructing all minimal words of $\Lambda^{\bar{w}}$. First, we partition \mathbb{A} into disjoint – potentially empty – subsets:

$$\begin{aligned}\mathbb{A}^0 &= \{a \in \mathbb{A} : a >_{\text{lex.}} a_0\}, \\ \mathbb{A}^i &= \{a \in \mathbb{A} : a_{i-1} \geq_{\text{lex.}} a >_{\text{lex.}} a_i\} \quad 1 \leq i \leq p, \\ \mathbb{A}^{p+1} &= \{a \in \mathbb{A} : a_p \geq_{\text{lex.}} a\}.\end{aligned}$$

It then follows that – empty \mathbb{A}^i 's not being considered,

- ▶ $\forall b \in \mathbb{A}^0$, the word b is minimal,
- ▶ $\forall b \in \mathbb{A}^i$ with $i \in \{1, \dots, p\}$, the word $a_0 \cdots a_{i-1}b$ is minimal,
- ▶ $\forall b \in \mathbb{A}^{p+1}$, the word $\bar{w}b$ is minimal.

As we partitioned \mathbb{A} , we have proved the following proposition.

Proposition 5.3 *The number of minimal words of $\Lambda^{\bar{w}}$ is exactly $\#\mathbb{A}$.*

As a follow-up of the example some lines ago, with $\mathbb{A} = \{0, 1, 2, 3\}$ and $\bar{w} = 211$, we apply the proposed method to find the minimal elements of $\Lambda^{\bar{w}}$. We partition \mathbb{A} into: $\mathbb{A}^0 = \{3\}$, $\mathbb{A}^1 = \{2\}$, $\mathbb{A}^2 = \emptyset$, $\mathbb{A}^3 = \{0, 1\}$. The four minimal words are therefore 3, 22, 2111 and 2110.

Although the previous result is completely general, if we require that $\mathbb{A} = \{0, \dots, n\}$, then the partition method described above can be rewritten into **Algorithm 13**. While this is not included in the pseudocode provided, note that the algorithm should return an empty list if $a_0 > n$, as in this case there would be no minimal word to look for.

Algorithm 13: MINIMALWORDS

Input: $\bar{w} = a_0 \cdots a_p$, $\mathbb{A} = \{0, \dots, n\}$

Output: All minimal words of $\Lambda^{\bar{w}}$

```

1 Set  $L$  to the empty list
2 if  $a_0 < n$  then
3   for  $i \in \{a_0 + 1, \dots, n\}$  do
4     Add the word  $i$  to  $L$ 
5 for  $k \in \{1, \dots, p\}$  do
6   if  $a_k < a_{k-1}$  then
7     for  $i \in \{a_k + 1, \dots, a_{k-1}\}$  do
8       Add the word
9          $a_0 \cdots a_{k-1}i$  to  $L$ 
9 for  $i \in \{0, \dots, a_p\}$  do
10  Add the word  $a_0 \cdots a_p i$  to  $L$ 
11 return  $L$ 

```

Canonical FDAGs

FDAGs are unordered objects, like the trees they compress, and therefore their enumeration requires to reflect this nature. In practice, finding a systematic way to order them makes it possible to design a simpler reduction rule, as done for unordered trees [30], ignoring the combinatorics of permutations. The purpose of this subsection is to provide a unique way to order FDAGs. We show that such an order exists in **Theorem 5.4**, unambiguously characterizing FDAGs. The approach chosen is based on the notion of topological ordering.

As we shall see, the topological orderings of a DAG are generally not unique. We will constrain them with carefully chosen conditions, so that there is only one topological order verifying said conditions: this one will induce our canonical form.

[87]: Kahn (1962), ‘Topological sorting of large networks’

[28]: Avis et al. (1996), ‘Reverse search for enumeration’

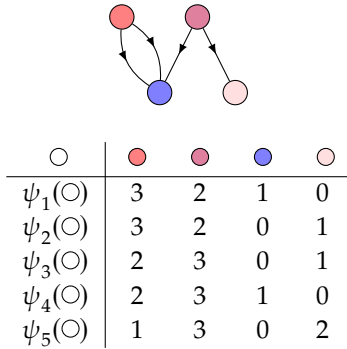


Figure 5.6: The DAG on the top admits five topological orderings, which are shown in the table.

Topological ordering Let D be a directed graph, where multiple arcs are allowed. A topological ordering on D is an ordering of the vertices of D such that for every arc $u \rightarrow v$, u comes after v in the ordering. Formally, $\psi : D \rightarrow \llbracket 0, \#D - 1 \rrbracket$ is a topological ordering if and only if ψ is bijective and $\psi(u) > \psi(v)$ for all $u, v \in D$ such that there exists at least one arc $u \rightarrow v$ in D . A well known result establishes that D is a DAG if and only if it admits a topological ordering [87]. Nonetheless, when a topological ordering exists, it is in general not unique – see Figure 5.6, which will serve as a followed example throughout this section. A reverse search enumeration of topological orderings of a given DAG can actually be found in [28].

Constrained topological ordering We aim to reduce the number of possible topological orderings of a DAG by constraining them. Let D be a DAG and ψ a topological ordering. Taking advantage of the vertical hierarchy of DAG, our first constraint is

$$\forall (u, v) \in D^2, \mathcal{H}(u) > \mathcal{H}(v) \implies \psi(u) > \psi(v). \tag{5.5}$$

Applying Equation 5.5 to the topological orderings presented in Figure 5.6, ψ_5 must be removed, as $\psi_5(\circ) > \psi_5(\bullet)$ but $\mathcal{H}(\bullet) > \mathcal{H}(\circ)$.

For any vertex v , and any $u \in C(v)$, by definition, $\mathcal{H}(v) > \mathcal{H}(u)$. Therefore, there can be no arcs between vertices at same height. Any arbitrary order on them leads to a different topological ordering. The next constraint we propose relies on the lexicographic order;

$$\forall (u, v) \in D^2, (\mathcal{H}(u) = \mathcal{H}(v)) \wedge (C_\psi(u) >_{\text{lex}} C_\psi(v)) \implies \psi(u) > \psi(v), \tag{5.6}$$

where $C_\psi(v)$ is the sequence $(\psi(w) : w \in C(v))$ sorted by decreasing order with respect to the lexicographic order. In other words, $C_\psi(v)$ is a decreasing word – see Definition 5.3 – on the alphabet $\mathbb{A} = \llbracket 0, \#D - 1 \rrbracket$. Table 5.1 illustrates the behavior of Equation 5.6 on the followed example of Figure 5.6.

○	●	●	(5.6)
$C_{\psi_1}(\circ)$	11	10	✓
$C_{\psi_2}(\circ)$	00	10	✗
$C_{\psi_3}(\circ)$	00	10	✓
$C_{\psi_4}(\circ)$	11	10	✗

Table 5.1: Application of Equation 5.6 to the remaining topological orderings of Figure 5.6 that satisfy Equation 5.5. As $C_\psi(\bullet) = C_\psi(\circ)$, we only need to consider vertices \bullet and \bullet . As $\psi_i(\bullet) > \psi_i(\bullet) \iff i \in \{1, 2\}$, the only orderings that are kept are ψ_1 and ψ_3 .

The combination of those two constraints imposes uniqueness in all cases except when there exist $u, v \in D^2$ such that $C_\psi(u) = C_\psi(v)$ and $u \neq v$. It should be clear that if we impose the condition of upcoming Equation 5.7, such a pathological case can not occur.

$$\forall (u, v) \in D^2, u \neq v \implies C(u) \neq C(v). \tag{5.7}$$

Upcoming Theorem 5.4 establishes that a DAG compresses an irredundant forest if and only if the topological order constrained by Equation 5.5 and Equation 5.6 is unique. In other words, an unambiguous characterization of FDAGs is exhibited.

Theorem 5.4 *The following statements are equivalent:*

- (i) D fulfills Equation 5.7,
- (ii) there exists a unique topological ordering ψ of D that satisfies both Equation 5.5 and Equation 5.6,

(iii) there exists a unique irredundant forest $F \in \mathcal{F}$ – cf. Equation 5.3 – such that $D = \mathfrak{R}(F)$,

where \mathfrak{R} is the DAG reduction operation defined in Section 2.3.

Proof. (i) \iff (ii) follows from the above discussion. (iii) \implies (i) follows from the definition of \mathfrak{R} . Indeed, if there were two distinct vertices $(u, v) \in D^2$ with the same multiset of children, they would have been compressed as a unique vertex in the reduction. We now prove that (i) \implies (iii).

In the first place, if D fulfills Equation 5.7, then D must admit a unique leaf, denoted by $\mathcal{L}(D)$. Indeed, if there were two leaves l_1 and l_2 , we would have $\mathcal{H}(l_1) = \mathcal{H}(l_2) = 0$ but also $C(l_1) = C(l_2) = \emptyset$, which would violate Equation 5.7. Let r_1, \dots, r_k be the vertices in D that have no parent. We define D_1, \dots, D_k as the subDAG rooted respectively in r_1, \dots, r_k . Then, we define $T_i = \mathfrak{R}^{-1}(D_i)$ and $F = \{T_1, \dots, T_k\}$. The T_i 's are well defined as all vertices in D (consequently in D_i) have a different multiset of children, and therefore compress distinct subtrees – i.e. F fulfills Equation 5.3, therefore $F \in \mathcal{F}$. Moreover, $D = \mathfrak{R}(F)$. 🍃

In Chapter 6, we shall only consider FDAGs. Consequently, from Theorem 5.4, they admit a unique topological ordering ψ satisfying both Equation 5.5 and Equation 5.6, called *canonical ordering*. Thus, for any FDAG D , the associated canonical ordering ψ will be implicitly defined. The vertices will be numbered accordingly to their ordering, i.e. $D = (v_0, \dots, v_n)$ with $\psi(v_i) = i$. Finally, as a consequence of Equation 5.5 and Equation 5.6, note that D can be partitioned in subsets of vertices with same height, each of them containing only consecutive numbered vertices. Figure 5.7 provides an example of a FDAG and its canonical ordering.

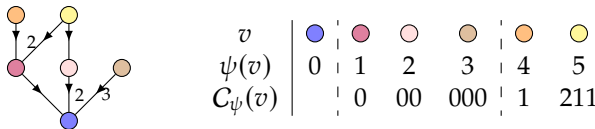


Figure 5.7: A FDAG D (left) and its canonical ordering ψ (right). Vertices that are at the same height are enclosed in the table between the dashed lines.

Enumeration of forests

I'm always astonished by a forest. It makes me realise that the fantasy of nature is much larger than my own fantasy. I still have things to learn.

Gunter Grass

This chapter is dedicated to the problem of enumerating unordered irredundant forests, as defined in [Definition 5.2](#). By virtue of [Theorem 5.2](#), it is equivalent to enumerate FDAGs, i.e. DAGs compressing (without loss) irredundant forests. Furthermore, it has been shown that FDAGs can be ordered in a canonical way, through [Theorem 5.4](#).

Consequently, the majority of this chapter is dedicated to the enumeration of *canonical FDAGs*. In particular, we set up a reverse search procedure – presented in [Section 5.1](#). As we have already seen, two points must be taken into consideration when building a reverse search method:

- ▶ the difficulty lies mainly in the expansion rule (i.e. the inverse of the reduction rule);
- ▶ the rules (reduction and expansion) must respect the canonicity of the enumerated items.

With regard to the first point, we intend in this chapter to build the reverse search procedure in *reverse*. That is, we construct three expansion rules in [Section 6.1](#), which, given a canonical FDAG, construct a new FDAG (also canonical, according to the second point). We then show that any canonical FDAG has a unique antecedent by one of these three rules. This allows us to construct the reduction rule (which associates a unique antecedent to a FDAG), and to complete the reverse search scheme.

This defines an enumeration tree of canonical FDAGs, whose properties are studied in [Section 6.2](#). We show that as the depth increases, the number of nodes in the enumeration tree at this depth grows exponentially. However, a given FDAG in the tree has a number of successors linear in the size of the FDAG; these successors can be computed in linear or quadratic time (depending on the chosen implementation). The growth of the tree is therefore, from this point of view, controlled. This allows us to conclude by showing that our algorithm runs with a polynomial delay, which is expected from a reverse search method.

The sequel of the chapter is devoted, first, to exploring three variants of our algorithm in [Section 6.3](#), allowing in particular (i) to enumerate classical forests, with redundancy; (ii) self-nested trees, a particular class of trees where all subtrees of the same height are isomorphic; and finally (iii) to constrain enumeration by imposing a certain number of criteria (degree, height, maximum number of nodes) on the enumerated FDAGs.

6.1 Exhaustive enumeration of FDAGs	72
Expansion rules	72
Analysis of the rules	74
Enumeration tree	76
6.2 Growth of the tree	77
Asymptotic growth	78
Branching factor	78
Polynomial delay	81
6.3 Variations on the enumeration tree	81
Extension to forests with repetitions	81
Enumeration of self-nested trees	83
Constraining the enumeration	84
6.4 Enumeration of forests of subtrees	85
Forests of subtrees	85
Frequent subFDAG mining problem	87

Finally, a particular instance of the frequent pattern mining problem – the frequent *subforest* mining problem is considered in Section 6.4, where the notion of subforest is defined first, and an enumeration algorithm is provided afterwards (largely based on the previous algorithm for enumerating FDAGs).

Most of this chapter makes a consistent use of previously introduced concepts, in particular those presented in A detour through formal languages (p. 66) and Canonical FDAGs (p. 67). The entire chapter is reproduced from [32], with the notable exception of Enumeration of self-nested trees (p. 83), which is new material.

[32]: Ingels et al. (2022), ‘Enumeration of Irredundant Forests’

6.1 Exhaustive enumeration of FDAGs

Reverse search techniques imply finding reduction rules, and then invert them. Equally, we will define instead three expansion rules, whose inverse will be reduction rules. An expansion rule takes a FDAG and creates a new FDAG, that is “expanded” in the sense of having either more vertices or more arcs. We show then that these rules preserve the canonicity of FDAGs, and that any FDAG admits a unique antecedent by one of the three rules in Proposition 6.3. The mapping of a FDAG to its unique antecedent is exactly our reduction rule, whose inverse are the three expansion rules. From this, we can derive an enumeration tree of canonical FDAGs, as claimed.

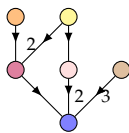
Expansion rules

We begin with a preliminary definition.

Definition 6.1 Let D be a FDAG, with $D = (v_0, \dots, v_n)$. We define the two following alphabets

$$\begin{aligned} \mathbb{A}_= &= \{\psi(v) : v \in D, \mathcal{H}(v) = \mathcal{H}(v_n)\} = \{p + 1, \dots, n\}, \\ \mathbb{A}_< &= \{\psi(v) : v \in D, \mathcal{H}(v) < \mathcal{H}(v_n)\} = \{0, \dots, p\}, \end{aligned}$$

where $p \in \llbracket 0, n - 1 \rrbracket$, $\psi(\cdot)$ is the canonical ordering of D , and $\mathcal{H}(\cdot)$ is the height as defined in Equation 2.1.



v	●	●	○	●	●	●
$\psi(v)$	0	1	2	3	4	5
$C_\psi(v)$		0	00	000	1	211

Figure 5.7: A FDAG D (top) and its canonical ordering ψ (bottom).

In other words, $\mathbb{A}_=$ contains the indices of all vertices that have the same height as the vertex with the highest index according to ψ , and $\mathbb{A}_<$ the indices of all vertices that have an inferior height. The FDAG presented in Figure 5.7 – reproduced here – will serve as a guideline example all along this subsection. Here, we have $\mathbb{A}_= = \{4, 5\}$ and $\mathbb{A}_< = \{0, 1, 2, 3\}$.

The three expansion rules are now introduced. Let $D = (v_0, \dots, v_n)$. Each of these rules is associated with an explicit symbol, which may be used, when necessary, to designate the rule afterward. It is worth noting that all of these rules will operate according to the vertex of highest index, i.e. v_n .

Branching rule (\curvearrowright) This rule adds an arc between v_n and a vertex below. The end vertex of the new arc is chosen such that $C_\psi(v_n)$ remains a decreasing word. In Figure 6.1, (\curvearrowright) is applied on our guideline example.

Definition 6.2 (\curvearrowright) Let $C_\psi(v_n) = a_0 \cdots a_m$. Choose $a_{m+1} \in \mathbb{A}_<$ such that $a_m \geq_{\text{lex}} a_{m+1}$ and add an arc between $\psi^{-1}(a_{m+1})$ and v_n .

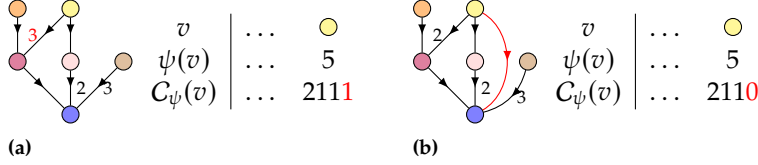


Figure 6.1: Branching rule applied to the FDAG of Figure 5.7. As $C_\psi(v_5) = 211$, the only letters a we can pick from $\mathbb{A}_< = \{0, 1, 2, 3\}$, satisfying $a \leq_{\text{lex}} 1$, are 0 and 1. The only two possible outcomes of (\curvearrowright) are the words (a) 2111 and (b) 2110.

Elongation rule (\uparrow) This rule adds a new vertex v_{n+1} so that $\mathcal{H}(v_{n+1}) = \mathcal{H}(v_n) + 1$. Consequently, the alphabets change and become $\mathbb{A}_= = \{n+1\}$ and $\mathbb{A}_< = \{0, \dots, n\}$. Note that after using this rule, it is not possible to ever add a new vertex at height $\mathcal{H}(v_n)$. See Figure 6.2 for an illustration of this rule on the guideline example.

Definition 6.3 (\uparrow) Add a new vertex v_{n+1} such that $C_\psi(v_{n+1}) = a_0 \in \mathbb{A}_=$.

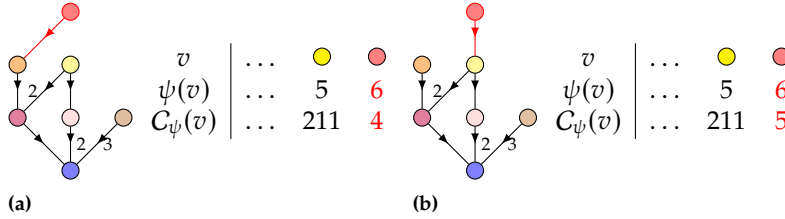


Figure 6.2: Elongation rule applied to the FDAG of Figure 5.7. As $\mathbb{A}_= = \{4, 5\}$, there are only two choices leading to (a) $C_\psi(v_6) = 4$ and (b) $C_\psi(v_6) = 5$. The alphabets become $\mathbb{A}_< = \{0, \dots, 5\}$ and $\mathbb{A}_= = \{6\}$.

Widening rule (\curvearrowleft) This rule adds a new vertex v_{n+1} at height $\mathcal{H}(v_n)$. The vertex is added with children that respects the canonicity of the DAG, that is, such that $C_\psi(v_{n+1}) >_{\text{lex}} C_\psi(v_n)$ – as in Equation 5.6. In other terms, denoting $\Lambda_<$ the language of decreasing words on alphabet $\mathbb{A}_<$, and with $\bar{w} = C_\psi(v_n)$, $C_\psi(v_{n+1})$ must be chosen in $\Lambda_<^{\bar{w}}$ – see Definition 5.4. However, this set is infinite, so we restrict $C_\psi(v_{n+1})$ to be chosen among the minimal words of $\Lambda_<^{\bar{w}}$. It follows from the definition of suffix-cut operator $SC(\cdot)$ that, by inverting the said operator, the other words in $\Lambda_<^{\bar{w}}$ can be obtained by performing repeated (\curvearrowright) operations. Finally, this new vertex is added to $\mathbb{A}_=$.

Definition 6.4 (\curvearrowleft) Add a new vertex v_{n+1} such that

$$C_\psi(v_{n+1}) \in \left\{ w \in \Lambda_<^{\bar{w}} : w \text{ is a minimal word of } \Lambda_<^{\bar{w}} \right\}$$

with $\bar{w} = C_\psi(v_n)$.

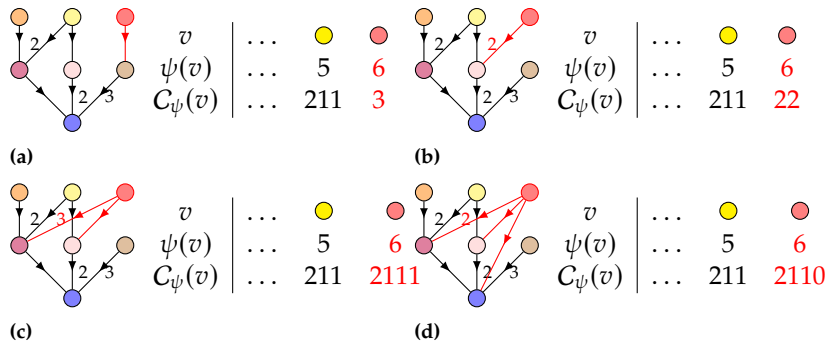
From Proposition 5.3 we know that such minimal words exist. We prove in the upcoming lemma that, as claimed, $\mathcal{H}(v_{n+1}) = \mathcal{H}(v_n)$.

Lemma 6.1 Any element of $\Lambda_{<}^{\bar{w}}$ defines a new vertex v_{n+1} so that $\mathcal{H}(v_{n+1}) = \mathcal{H}(v_n)$.

Proof. From the definition of $\mathcal{H}(\cdot)$, it suffices to prove that v_{n+1} admits at least one child at height $h = \mathcal{H}(v_n) - 1$. Let us denote b_0 and a_0 the first letter of, respectively, $C_\psi(v_{n+1})$ and $C_\psi(v_n)$. Denoting $v = \psi^{-1}(b_0)$ and $u = \psi^{-1}(a_0)$, we already know that $\mathcal{H}(u) = h$ – as ψ respects Equation 5.6 and $C_\psi(v_n)$ is a decreasing word. Therefore, as by construction $C_\psi(v_{n+1}) >_{\text{lex}} C_\psi(v_n)$, either (i) $b_0 = a_0$ and therefore $v = u$, or (ii) $b_0 >_{\text{lex}} a_0$. In the latter, as ψ respects both Equation 5.5 and Equation 5.6, $\mathcal{H}(v) \geq \mathcal{H}(u) = h$. But, as $b_0 \in \mathbb{A}_{<}$, $\mathcal{H}(v) < \mathcal{H}(v_n) = h + 1$. In both cases, $\mathcal{H}(v) = h$. ✍

Figure 6.3 illustrates the use of the widening rule on the followed example. It should be noted that the possible outcomes of (\mathcal{V}) are obtained by using Algorithm 13, applied to $w = C_\psi(v_n)$ and p – with $\mathbb{A}_{<} = \{0, \dots, p\}$.

Figure 6.3: We apply (\mathcal{V}) to the FDAG of Figure 5.7. Here, $\mathbb{A}_{<} = \{0, 1, 2, 3\}$ and $\bar{w} = 211$. As seen in A detour through formal languages (p. 66), the minimal words of $\Lambda_{<}^{\bar{w}}$ are 3, 22, 2111 and 2110. Therefore, there are 4 ways to add a new vertex v_6 via the widening rule, that are such that (a) $C_\psi(v_6) = 3$, (b) $C_\psi(v_6) = 22$, (c) $C_\psi(v_6) = 2111$ or (d) $C_\psi(v_6) = 2110$. Finally, we update $\mathbb{A}_=$ to be equal to $\{4, 5, 6\}$.



Analysis of the rules

Since our goal is to enumerate canonical FDAGs, it is required that the expansion rules indeed construct canonical FDAGs. This is achieved by virtue of the following proposition.

Proposition 6.2 The expansion rules preserve the canonicity property.

Proof. Let $D = (v_0, \dots, v_n)$ be a canonical FDAG. The proposition follows naturally from the definitions:

- (\mathcal{A}) Let a be the letter added to $w = C_\psi(v_n)$. As $wa >_{\text{lex}} w >_{\text{lex}} C_\psi(v_{n-1})$, the ordering is unchanged.
- (\mathcal{U}) The new vertex v_{n+1} is such that $\mathcal{H}(v_{n+1}) > \mathcal{H}(v_n)$, so Equation 5.5 is still met.
- (\mathcal{V}) The new vertex v_{n+1} is chosen so that $\mathcal{H}(v_{n+1}) = \mathcal{H}(v_n)$ and $C_\psi(v_{n+1}) >_{\text{lex}} C_\psi(v_n)$, so Equation 5.6 is also still met.

Therefore, any FDAG obtained from D is still a canonical FDAG. ✍

From now on, since it is assured that the canonicity is preserved, we shall use the term FDAG for “canonical FDAG” for the sake of simplicity.

Furthermore, since our goal is to provide the FDAGs space with an enumeration tree, which will be explored via the expansion rules, it is important that these expansion rules are “bijective” in the following sense: for any FDAG D , there exists a unique FDAG D' such that D is obtained from D' via one of the three rules (\curvearrowright) , (\uparrow) or (\Downarrow) .

Such D' can be constructed via [Algorithm 14](#) as shown in upcoming [Proposition 6.3](#). Conditional expressions applied to D are used to determine which modification should be applied to construct D' . The gray symbol (in the algorithm) next to these modifications indicates which expansion rule allows to retrieve D from D' .

Algorithm 14: ANTECEDENT

Input: $D = (v_0, \dots, v_n)$; $w = C_\psi(v_n)$; $w' = C_\psi(v_{n-1})$

```

1 if  $v_n$  is the only vertex of height  $\mathcal{H}(v_n)$  then
2   if  $\#w = 1$  then
3     |  $(\uparrow)$  Delete vertex  $v_n$ 
4   else
5     |  $(\curvearrowright)$   $w \leftarrow SC(w)$ 
6 else
7   if  $w$  is a minimal word of  $\Lambda_{<}^{w'}$  then
8     |  $(\Downarrow)$  Delete vertex  $v_n$ 
9   else
10    |  $(\curvearrowright)$   $w \leftarrow SC(w)$ 

```


/ SC(·) is the suffix-cut operator defined in [Equation 5.4](#). */*

Proposition 6.3 *Algorithm 14 applied to any FDAG constructs the unique antecedent of this FDAG.*

Proof. Let $D = (v_0, \dots, v_n)$ be a FDAG. Let $w = C_\psi(v_n)$ and $w' = C_\psi(v_{n-1})$. Two cases can occur: either (i) v_n is the only vertex at height $\mathcal{H}(v_n)$, or (ii) it is not.

- (i) It is clear in this case that D can not be obtained from any FDAG via the rule (\Downarrow) – otherwise v_n would not be alone at its height. Concerning (\curvearrowright) and (\uparrow) , let us look at the number of children of v_n .
- ▶ If v_n admits only one child, it must come from an (\uparrow) step, since (\curvearrowright) would imply that $\#w \geq 2$. Therefore, in this case, D can be retrieved among the outcomes of rule (\uparrow) applied to $D' = (v_0, \dots, v_{n-1})$.
 - ▶ Otherwise, when $\#w > 1$, D can not come from an (\uparrow) step, and must therefore come from (\curvearrowright) . Denoting v'_n the vertex with list of children $SC(w)$ – see [Equation 5.4](#), D is one of the outcomes of $D' = (v_0, \dots, v_{n-1}, v'_n)$ via (\curvearrowright) .
- (ii) following the same logic as (i), D can not be obtained via (\uparrow) . We discriminate between rules (\Downarrow) and (\curvearrowright) when comparing w and w' . If w is a minimal word of $\Lambda_{<}^{w'}$, then D can not be obtained from (\curvearrowright) – this would break the canonical order. Therefore, in this

case, D is an outcome of rule (\uparrow) applied to $D' = (v_0, \dots, v_{n-1})$. Otherwise, if w is not a minimal word, then it can not be obtained from (\uparrow) , and must come from a (\curvearrowright) step, applied to $D' = (v_0, \dots, v_{n-1}, v'_n)$ where $C_\psi(v'_n) = SC(w)$.


Whatever the case among those evoked, they correspond exactly to the conditional expressions of the [Algorithm 14](#), which therefore constructs the correct antecedent of D , which is unique by virtue of the previous discussion. 

Enumeration tree

In this subsection, we construct the enumeration tree of FDAGs derived from the expansion rules. As aimed, their inverse is indeed a reduction rule.

Theorem 6.4 *Algorithm 14 is a reduction rule, as in Definition 5.1.*

Proof. Let us denote $f(D)$ the output of [Algorithm 14](#) applied to a FDAG D . We need to prove that: (i) $f(D)$ is a “maximal” subgraph of D and (ii) for any $D \neq D_0$, there exists an integer k such that $f^k(D) = D_0$, where D_0 is the FDAG with one vertex and no arcs.

- ▶ Since [Algorithm 14](#) deletes either one vertex and its leaving arcs, or just one arc, $f(D)$ is indeed a subgraph of D . Moreover, by the construction of the expansion rules (and thus the reduction rule), there is no FDAG D' (different from $f(D)$ and D) such that $f(D)$ is a subgraph of D' and D' a subgraph of D .
- ▶ The sequence of general terms $f^k(D)$ is made of discrete objects whose size is strictly decreasing, therefore the sequence is finite and reaches D_0 . 

The associated expansion rule is exactly, in light of [Proposition 6.3](#), the union of the three expansion rules (\uparrow) , (\curvearrowright) and (\uparrow) . Since D_0 , the DAG with one vertex and no arcs, is a FDAG, by virtue of what precedes and with [Algorithm 12](#) – here with $g(\cdot) = \top$, we just defined an enumeration tree covering the whole set of FDAGs, whose root is D_0 . A fraction of this enumeration tree is shown in [Figure 6.4](#), illustrating the path from the root D_0 to the FDAG of [Figure 5.7](#). Unexplored branches are ignored, but are still indicated by their respective root.

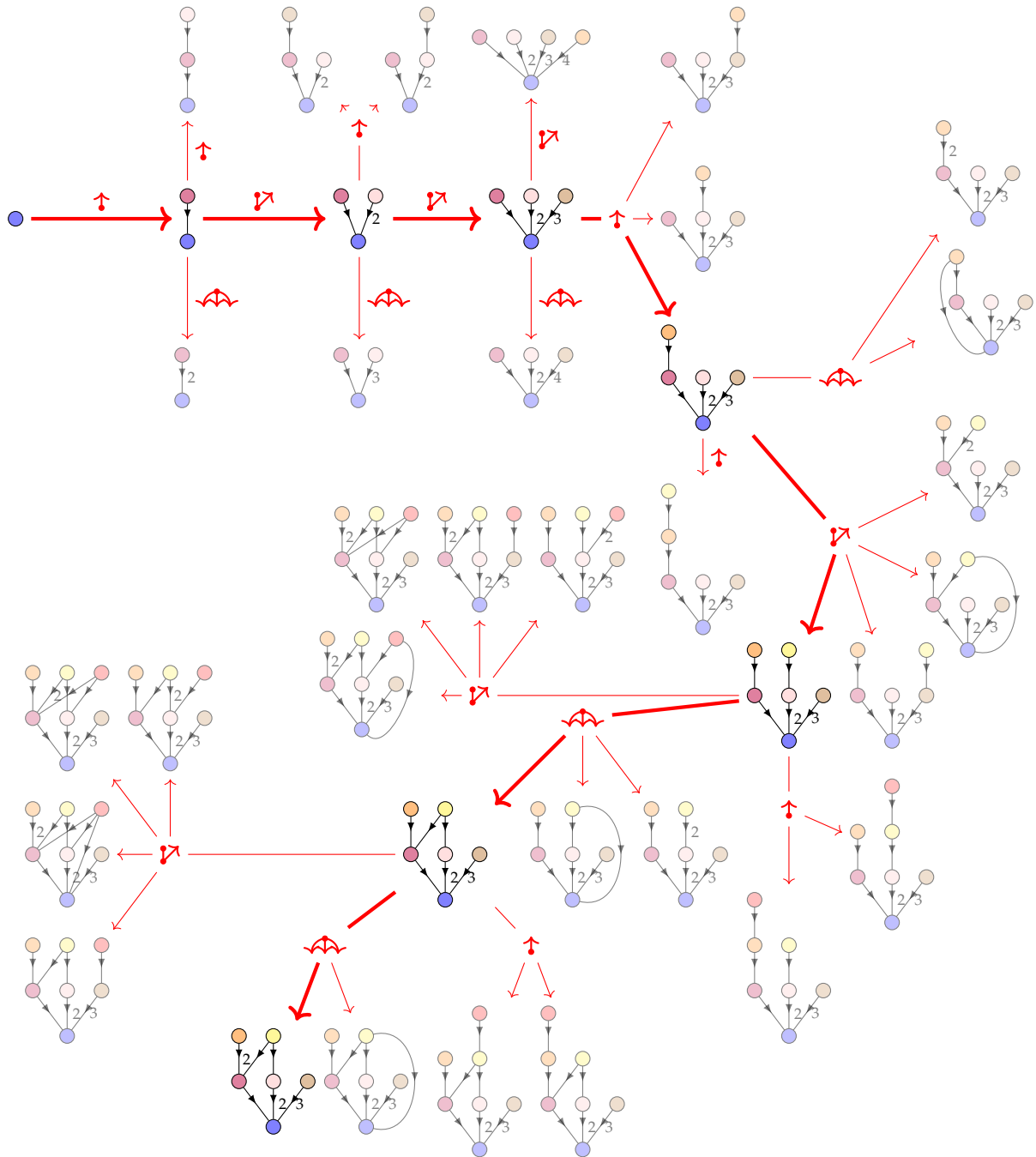


Figure 6.4: The path (in bold) in the FDAGs enumeration tree leading to the FDAG of [Figure 5.7](#). The unexplored branches are only displayed by their root, which are shown partially transparent. The order of insertion of the vertices of each FDAG is always the same, and follows the color code (in the order of insertion): blue, pink, grey, orange, yellow and red. With respect to the canonical ordering, they are numbered 0 to 6 in the same order.

6.2 Growth of the tree

In this section, we analyse the enumeration tree defined in [Section 6.1](#). We exhibit a bijection – [Theorem 6.5](#) – between FDAGs and a class of combinatorial objects from the literature, allowing us to obtain an asymptotic expansion of the growth of the tree. Moreover, we show that any FDAG has a linear number of children in that tree in [Theorem 6.7](#), and that the complexity to construct those children is quadratic – see

[80]: Johnson et al. (1988), ‘On generating all maximal independent sets’

k	$\#E_k$
0	1
1	1
2	3
3	12
4	61
5	380
6	2,815
7	24,213
8	237,348

Table 6.1: Number of FDAGs accessible from D_0 in k steps in the enumeration tree. These numbers were obtained numerically via our implementation with Python library `treex`.

29: OEIS Foundation Inc. (2022), The On-Line Encyclopedia of Integer Sequences, <http://oeis.org/A158691>.

[88]: Hwang et al. (2019), ‘Asymptotics and statistics on Fishburn matrices and their generalizations’

[89]: Jelínek (2012), ‘Counting general and self-dual interval orders’

[90]: Bringmann et al. (2014), ‘Asymptotics for the number of row-Fishburn matrices’

30: “successor” in the sense of “children in the enumeration tree”. We make the distinction to avoid confusion with the children denoted by $C(\cdot)$.

Proposition 6.8. Finally, **Theorem 6.9** states that our algorithm runs with polynomial delay [80].

Asymptotic growth


In this subsection, we show that FDAGs are in bijection with a set of particular matrices, whose combinatorial properties are known and give us access to an asymptotic expansion of the enumeration tree growth.

Let us denote E_k the set of all FDAGs that are accessible from D_0 in exactly k steps in the enumeration tree – with $E_0 = \{D_0\}$; then Table 6.1 depicts the values of $\#E_k$ for the first nine values of k .

Actually, the terms of Table 6.1 coincide with the first terms of OEIS sequence A158691²⁹, which counts the number of *row-Fishburn matrices* that are upper-triangular matrices with at least one nonzero entry in each row. The *size* of such a matrix is equal to the sum of its entries.

Theorem 6.5 *There exists a bijection Φ between the set of FDAGs and the set of row-Fishburn matrices, such that if D is a FDAG and $M = \Phi(D)$, then*

$$D \in E_k \iff \text{size}(M) = k.$$

Proof. The proof lies in **Appendix B**. 

This connection is to our advantage since Fishburn matrices (in general) are combinatorial objects widely explored in the literature as they are in bijection with many others – see [88] for a review. Notably, the asymptotic expansion of the number of row-Fishburn matrices has been conjectured first by Jelínek [89] and then proved by Bringmann et al. [90].

Proposition 6.6 (Jelínek, Bringmann et al.) *As $k \rightarrow \infty$,*

$$\#E_k = k! \left(\frac{12}{\pi^2} \right)^k \left(\beta + O\left(\frac{1}{k}\right) \right)$$

$$\text{with } \beta = \frac{6\sqrt{2}}{\pi^2} e^{\pi^2/24} = 1.29706861206 \dots$$

Branching factor

Given the overall structure of FDAGs, it is no surprise that the enumeration tree grows extremely fast. However, despite this combinatorial explosion, we show in this subsection that the branching factor, i.e., the outdegree of the nodes in the enumeration tree, is controlled. Actually, we prove that any FDAG has a linear number of successors³⁰ in the enumeration tree.

Theorem 6.7 *Any FDAG D has $\Theta(\#D)$ successors in the FDAG enumeration tree.*

Proof. Let $D = (v_0, \dots, v_n)$ be a FDAG. We denote $C_\psi(v_n) = a_0 \cdots a_m$. Depending on the rule chosen:

- (\curvearrowright) a_{m+1} belongs to $\mathbb{A}_< = \{0, \dots, p\}$, so the maximum number of successors is at most $p + 1$, and at least 1, depending on the condition $a_m \geq_{\text{lex}} a_{m+1}$.
- (\uparrow) The child of the new vertex is taken from $\mathbb{A}_= = \{p + 1, \dots, n\}$ so the number of successors is exactly $n - p$.
- (\curvearrowleft) Following [Proposition 5.3](#), the number of successors is exactly $\#\mathbb{A}_< = p + 1$.

Combining everything, the number of successors is at least $n + 2$ and at most $n + p + 2 \leq 2n + 1$ (as $p \leq n - 1$, with equality for FDAGs obtained just after using (\uparrow) rule). \spadesuit

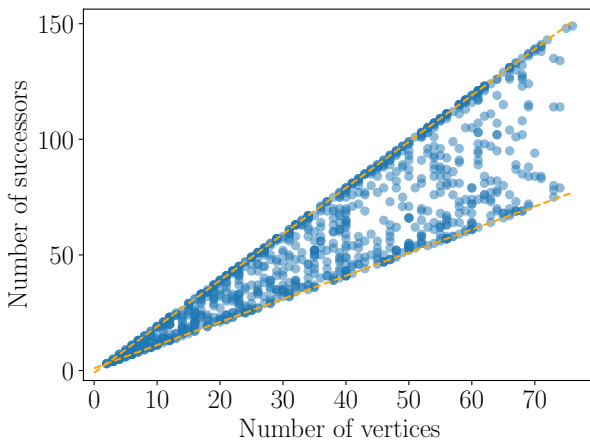


Figure 6.5: Numbers of successors of 1,000 random FDAGs in the enumeration tree, according to their number of vertices. Orange lines have equations $y = n + 1$ and $y = 2n - 1$.

In the previous proof, we have shown that the number of successors of a FDAG with n vertices is between $n + 1$ and $2n - 1$. [Figure 6.5](#) illustrates that these boundaries are tight, on 1 000 randomly generated FDAGs. A random FDAG is constructed as follows.

Definition 6.5 (Random FDAG) *Let $k \geq 0$. Starting from D_0 – the root, construct iteratively D_i as a successor of D_{i-1} in the enumeration tree, picked uniformly at random. We stop after k steps, and keep D_k .*

In [Figure 6.5](#), we generated 10 random FDAGs for each $k \in \{1, \dots, 100\}$.

It is indeed a suitable property that any FDAG admits a linear number of successors; but it would be of little use if the time required to compute those successors is too important. We demonstrate in the following proposition that the complexity is manageable. There are two possible strategies: (i) one can keep the enumeration tree in memory, and store on each node only the increment allowing to construct a FDAG from its predecessor; or (ii) one can explicitly build the successors by copying the starting FDAG, so that the tree can be forgotten. Depending on whether one wants to build the tree itself or only the FDAGs that compose it, one will choose either strategy.

Proposition 6.8 *Computing the successors of any FDAG D has complexity:*

- (i) $O(\#D \deg(D))$ if the construction is incremental from D ;

(ii) $O((\#D \deg(D))^2)$ if the construction involves copying D .

Proof. Let $D = (v_0, \dots, v_n)$ be a FDAG with $n + 1$ vertices, with $C_\psi(v_n) = a_0 \cdots a_m$, $\mathbb{A}_< = \{0, \dots, p\}$ and $\mathbb{A}_= = \{p + 1, \dots, n\}$. Although the alphabets $\mathbb{A}_<$ and $\mathbb{A}_=$ can be retrieved in linear time, it is more efficient to maintain the pair (n, p) during enumeration; how to update these indices has already been presented in [Expansion rules](#) (p. 72), when introducing each rule.

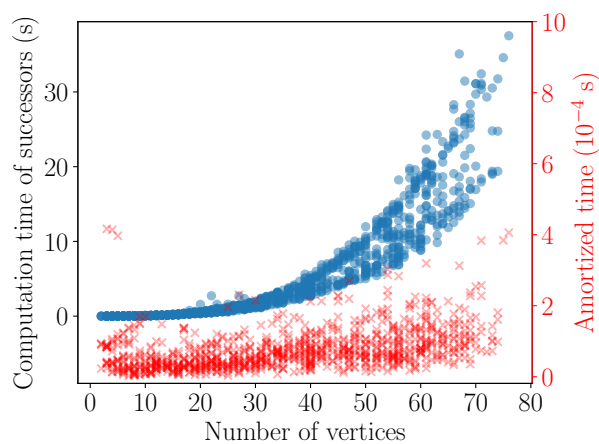
The explicit construction of the successors in case (ii) requires to copy the vertices of D and their children, leading to a complexity in the order of $\sum_{i=0}^n (1 + \deg(v_i))$, which can be roughly bounded by $(n + 1)(\deg(D) + 1)$.

Depending on the expansion rule, the complexity for computing the new vertex or new arc varies:

- (\curvearrowright) The last letter of $C_\psi(v_n)$ determines the number of successors – but it is no more than $p + 1$. In case (i), although we could just store the information of the new letter, it is better to copy $C_\psi(v_n)$ and add the new letter and store the result. Indeed, this allows to always have the knowledge of $C_\psi(v_n)$ in the enumeration tree. The complexity for case (i) is therefore bounded by $\deg(v_n)(p + 1)$; whereas it is $p + 1$ in case (ii) since $C_\psi(v_n)$ is already copied.
- (\uparrow) Each successor is obtained by picking one element of $\mathbb{A}_= = \{p + 1, \dots, n\}$. The complexity is exactly (up to a constant) $n - p$ in both cases.
- (\curvearrowright) The successors are obtained by [Algorithm 13](#), involving copying subwords of $C_\psi(v_n)$ – the overall complexity is bounded by $(p + 1) \deg(v_n)$.

The overall complexity is therefore of the order of $2(p + 1) \deg(v_n) + n - p$ in case (i) and of $(n + 1)(\deg(D) + 1)[(p + 1)(\deg(v_n) + 1) + n - p]$ in case (ii). Using rough bounds, with $\deg(v_n) \leq \deg(D)$ and $p \leq n$, we end up with the stated complexity. 🍃

Figure 6.6: Total computation time (in blue) and amortized time (in red) for the explicit construction of the successors of the 1,000 random FDAGs of [Figure 6.5](#), according to their number of vertices. The computations have been made on a MacBook Pro (2014) with an Intel Core i7 2.8 GHz processor and 16 GB of RAM.



Whereas [Figure 6.5](#) shows the number of successors of 1,000 random FDAGs, we measured the time needed to compute *explicitly* – i.e., implying copy, which is case (ii) in the previous [Proposition 6.8](#) – these successors. The results are depicted in [Figure 6.6](#), where we plotted (in blue) the total time t_D for computing all successors of a given FDAG D , and (in red) what we call *amortized time*, i.e. $t_D / (\#D \deg(D))^2$. As expected from

Proposition 6.8, one can observe an asymptotic quadratic behaviour for the total time (in blue); concerning amortized time (in red), despite some variability, the upper bound seems to be constant.


Polynomial delay

Let $E_{\leq K+1} = \bigcup_{k \leq K+1} E_k$ be the set of all FDAGs reachable in at most $K + 1$ steps from the root of the FDAG enumeration tree. In this subsection, we show that the complexity for enumerating $E_{\leq K+1}$ can be expressed as a function of the cardinality of $E_{\leq K}$ and has polynomial delay.

Theorem 6.9 *Enumerating $E_{\leq K+1}$ has complexity $O(K^2 \#E_{\leq K})$.*

Proof. We adopt the configuration where we keep the enumeration tree in memory and where each node contains the incremental information to construct a FDAG from its predecessor.

We first observe the following: as D_0 , the root, has one vertex and no arcs, and since the rules of expansion can only add one vertex and/or increase the degree of the last vertex by one, for any $D \in E_k$, it follows naturally that $\#D \leq k + 1$ and $\deg(D) \leq k$. It implies that the complexity for generating the successors of a FDAG in E_k is $O(k^2)$, according to case (i) of **Proposition 6.8**.

Thus, enumerating all the elements of E_{k+1} requires a complexity of $O(k^2 \#E_k)$. It follows that we have a complexity of $O(\sum_{k \leq K} k^2 \#E_k)$ for enumerating $E_{\leq K+1}$. Since $k \leq K$ and $\sum_{k \leq K} \#E_k = \#E_{\leq K}$, we end up with the announced complexity. 

As such, our algorithm has a polynomial delay, which is desirable for this kind of enumeration [80].

[80]: Johnson et al. (1988), ‘On generating all maximal independent sets’

6.3 Variations on the enumeration tree

In this section, three variants of the enumeration tree presented in **Section 6.1** are introduced. First, we propose a way to enumerate forests in their classical sense, i.e., where redundancies within the forest are accepted, by adding an extra step following the previous enumeration. Then, a very simple adaptation is proposed to restrict the enumeration to a very specific class of trees, the self-nested trees. Finally, options to constrain the enumeration tree – on maximum number of vertices, height or outdegree – and making it finite are proposed.

Extension to forests with repetitions

The enumeration tree constructed in **Section 6.1** only allows to enumerate, in their compressed form, irredundant forests, where no tree can be a subtree of (or equal to) another. In this subsection, we propose a method to enumerate forests in the usual sense, without this non-redundancy restriction.

Let F be a forest in the classical sense, i.e., where some trees may be identical to or subtrees of other trees. If we compute $D = \mathfrak{R}(F)$ – without resorting to an artificial root, i.e. following the method described in [Irredundant forests](#) (p. 65), all these redundancies will be lost: if a tree is a subtree of another one, then it will be compressed with this subtree as a same vertex in the DAG. This is why DAG compression of forests – without artificial root – is lossless if and only if the forest is irredundant.

We can preserve the information lost by the compression if we keep, in addition, a *presence vector*³¹. Let us rewrite $D = (v_0, \dots, v_n)$ according to the canonical order. Each tree $T \in F$ is associated with an index $i \in \{0, \dots, n\}$ such that $T = \mathfrak{R}^{-1}(D[v_i])$. The presence vector $\pi_F : \{0, \dots, n\} \rightarrow \mathbb{N}$ is constructed such that $\pi_F(i)$ counts how many times the tree $\mathfrak{R}^{-1}(D[v_i])$ appears in F . Thus, the couple (D, π_F) completely characterizes the forest F . To enumerate all (redundant) forests, it is therefore sufficient to enumerate both all FDAGs (corresponding to irredundant forests) and the presence vectors that may be associated with them.

Let D be an FDAG constructed in the FDAG enumeration tree. We define π_D as the presence vector associated to the (irredundant) forest $\mathfrak{R}^{-1}(D)$. This vector can be computed in a linear traversal of D , where the sources of D are assigned a value of 1 and the other vertices are assigned a value of 0. Adding redundancies in a forest means incrementing the presence vector, each +1 resulting in a new tree, whether it is equal to an existing tree or a subtree of it.

Our strategy is to enumerate, from π_D , all presence vectors corresponding to forests whose DAG reduction would be exactly D . To do so, we use a reverse search structure, with the following expansion rule (E). Let j be the index of the last increment, initialized to $j = 0$, and let π be the current presence vector.

Definition 6.6 (E) Choose any index $j' \geq j$. Increase $\pi(j')$ by one and set $j \leftarrow j'$.

This rule allows to get any presence vector from π_D in a unique way, i.e., each index must be increased to its desired final value before moving to the next index. This defines an enumeration tree of presence vectors. If we implement this tree in such a way that each node contains only the new index j' , we obtain an algorithm that enumerates each presence vector from its parent in constant time and space. The growth of this (infinite) new enumeration tree is given by the following proposition.

Proposition 6.10 The number of redundant forests that can be constructed in at most $k \geq 1$ steps from $\mathfrak{R}^{-1}(D)$ – following expansion rule (E) – is given by $\binom{n+1+k}{k} - 1$.

Proof. We first notice that the expected number is exactly the same as the number of presence vectors constructible in at most k steps from π_D . We then notice that if the current node (in the presence vector enumeration tree) has index j , then it has $n + 1 - j$ successors by the expansion rule (E) of [Definition 6.6](#). For instance, since the starting index is 0, for $k = 1$, we obtain the indices $0, \dots, n$ in one copy each. We denote by $n_p(j)$ the

31: Note that this is different from the presence defined in [Connection between a forest and its compressed form](#) (p. 19), where the presence counts how many nodes in a tree or forest have a given equivalence class.

number of times the index j appears in the nodes obtained in exactly p steps from the origin. Thus, $n_1(j) = 1$ by the above. Each index $j' \leq j$ existing at step $p - 1$ will induce a successor with index j at step p , so that $n_p(j) = \sum_{j'=0}^j n_{p-1}(j')$.

We establish by induction on p that $n_p(j) = \binom{k-1+j}{j}$, using the so called hockey-stick identity $\sum_{r=0}^m \binom{n+r}{r} = \binom{n+1+m}{m}$ [91]. Since the number of presence vectors that can be constructed in at most $k \geq 1$ steps from π_D is given by $\sum_{p=1}^k \sum_{j=0}^n n_p(j)$, we obtain the expected result after applying twice the hockey-stick identity. 🍃

We can merge the enumeration tree of repetitions with the enumeration tree of FDAGs, to form a single enumeration tree, which enumerates forests in the classical sense (and in compressed form), as follows: the nodes of the enumeration tree carry a couple (FDAG, presence vector), and the available expansion rules are (\curvearrowright) , (\uparrow) , (\curvearrowleft) and (E) . However, successors created with the last rule produce branches where it becomes the only rule available. In other words, once one chooses repetition, one can not modify any longer the topology of the FDAG – this is to ensure that each forest can only be enumerated in a unique way.

Enumeration of self-nested trees

Self-nested trees are a class of trees introduced in [92] that achieve maximum redundancy in their subtrees. More precisely,

Definition 6.7 A tree T is called self-nested if for any $u, v \in T$,

$$\mathcal{H}(u) = \mathcal{H}(v) \implies T[u] \simeq T[v].$$

Self-nested trees are closely related with linear DAGs (i.e. DAGs containing at least one path going through all vertices), as per the following proposition [61]. See also Figure 6.7.

Proposition 6.11 (Godin, Ferraro) A tree T is self-nested if and only if $\mathfrak{R}(T)$ is linear.

These trees have many algorithmic qualities, both from the point of view of compression rates and the evaluation of recursive functions, or even the calculation of editing distances, as shown in [93]. In particular, it is shown that self-nested trees are particularly rare within the space of unordered trees. It follows that the exhaustive exploration of the space of self-nested trees is exponentially easier than for general trees – although [93] does not provide an enumeration algorithm for it.

Thanks to the expansion rules introduced earlier, it becomes straightforward to propose an enumeration tree for self-nested trees: it is enough to forbid the (\curvearrowleft) rule, and keep only (\curvearrowright) and (\uparrow) . Indeed, there can be only one vertex at each height (obtained by (\uparrow)), and thus the enumerated FDAGs will be linear. Also note that since each enumerated FDAG has only one root, it does compress a single tree and not a forest.

[91]: Jones (1994), ‘Generalized Hockey Stick Identities and N-dimensional Blockwalking’

[92]: Greenlaw (1996), ‘Subtree isomorphism is in DLOG for nested trees’

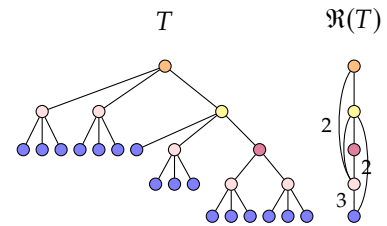


Figure 6.7: A self-nested tree T (left) and its DAG reduction $\mathfrak{R}(T)$ (right). Nodes are colored according to their class of equivalence.

[61]: Godin et al. (2009), ‘Quantifying the degree of self-nestedness of trees: application to the structural analysis of plants’

[93]: Azaïs et al. (2019), ‘Approximation of trees by self-nested trees’

Constraining the enumeration

[30]: Nakano et al. (2003), ‘Efficient generation of rooted trees’

[93]: Azaïs et al. (2019), ‘Approximation of trees by self-nested trees’

In [30], the authors propose an algorithm to enumerate all trees with at most n vertices. They simply check whether the current tree has n vertices or not, and as their expansion rule adds one vertex at a time, they decide to cut a branch in the enumeration tree once they have reached n vertices. Similarly, adding a vertex to a tree can only increase its height or outdegree, so we can proceed in the same way to enumerate all trees with maximal height H and maximal outdegree d . Indeed, the number of trees satisfying those constraints is finite [93].

This property also holds with the approach presented in Section 6.1: following one of the three expansion rules, we can only increase the height, outdegree or number of vertices of the FDAG. So, it makes sense to define similar constraints on the enumeration. However, for this constrained enumeration to generate a finite number of FDAGs, constraints must be chosen wisely, as shown in the following proposition.

Proposition 6.12 *The enumeration tree of FDAGs is finite if at least one of those set of constraints is chosen:*

- (i) maximum number of vertices n and maximum outdegree d ,
- (ii) maximum height H and maximum outdegree d .

Proof. As (\curvearrowright) allows to add arcs indefinitely without changing the numbers of vertices, constraining on the maximum outdegree is mandatory in both cases. As the two others rules add vertices, constraining by the number of vertices leads to a finite enumeration tree – (i) is proved. To conclude, we only need to prove that (\curvearrowleft) can not be repeated an infinite number of times, i.e. there is only a finite number of new vertices that can be added at a given height, up to the maximum outdegree. This is achieved by virtue of the upcoming lemma.


Let $H > 2$ and $d \geq 1$. Let D be the FDAG constructed so that for each $0 \leq h \leq H$, D has the maximum possible number n_h of vertices of height h and with maximum outdegree d . Initial values are $n_0 = 1$ and $n_1 = d$.

Lemma 6.13 $\forall 2 \leq h \leq H$,

$$n_h = \sum_{k=1}^d \binom{k + n_{h-1} - 1}{k} \binom{d - k + n_0 + \dots + n_{h-2}}{d - k}.$$

Let $h \geq 2$ be fixed. To lighten the notation, let $n = n_{h-1}$ and $m = n_0 + \dots + n_{h-2}$. Let v be a vertex to be added at height h . For any vertex v_i at height $h - 1$, let x_i be the multiplicity of v_i in $C(v) - 0$ if $v_i \notin C(v)$. Similarly, for any vertex v_j with $\mathcal{H}(v_j) \leq h - 2$, y_j is the multiplicity of v_j in $C(v) -$ possibly 0. By definition of $\mathcal{H}(\cdot)$ – see Equation 2.1, at least one x_i is non-zero. Therefore, there exist $k \in \llbracket 1, d \rrbracket$ such that:

$$\begin{aligned} x_1 + \dots + x_n &= k \\ y_1 + \dots + y_m &\leq d - k \end{aligned}$$

By virtue of the stars and bars theorem [94], for a fixed k , there are $\binom{k+n-1}{k}$ choices for variables x_i , and $\binom{d-k+m}{d-k}$ for variables y_j . Summing upon all values for k proves the claim. 

[94]: Feller (2008), *An introduction to probability theory and its applications, vol 2*

Remark 6.1 In the constrained enumeration proposed in [30], all the trees with n vertices are the leaves of the enumeration tree. To get all trees with $n + 1$ vertices, it suffices to add to the enumeration all children of these leaves, i.e. trees obtained by adding a single vertex to them. This property – moving from one parameter value to the next by enumerating just one step further – does not hold anymore as soon as our set of constraints involve the maximum outdegree d , both for trees and FDAGs. For instance, from a FDAG of height H , one can obtain FDAG of height $H + 1$ by using (\uparrow) once and repeating (\curvearrowright) up to $d - 1$ times.

[30]: Nakano et al. (2003), ‘Efficient generation of rooted trees’

6.4 Enumeration of forests of subtrees

Once the reverse search scheme has been set up to enumerate a certain type of structure, it is natural to move to a finer scale by using the same scheme to enumerate substructures. However, the notion of “substructure” is not obvious to derive from the main structure, as several choices are possible – e.g. for trees one can think of subtrees [37], subset trees [35], etc. From a practical point of view, the enumeration of substructures permits to solve the frequent pattern mining problem.

[37]: Vishwanathan et al. (2004), ‘Fast kernels for string and tree matching’

[35]: Collins et al. (2001), ‘Convolution kernels for natural language’

Forests of subtrees

In this section we define forests of subtrees, which will be our substructures. Compressed as FDAGs, these objects will be called subFDAGs. We then address the problem of enumerating all subFDAGs appearing in an FDAG D – similar as the one of enumerating all subtrees of a tree.

Definition Similarly to a forest being a tuple of trees, *forests of subtrees* are tuple of subtrees, satisfying Equation 5.3. Formally:

Definition 6.8 Let F and f be two irredundant forests. f is a forest of subtrees of F if and only if

$$\forall t \in f, \exists T \in F, t \in \mathcal{S}(T).$$

Forests of subtrees can be directly constructed from FDAGs, as shown by the upcoming proposition. Let D be a FDAG, and V be a subset of vertices of D .

Proposition 6.14 If $\forall v \in V, C(v) \subseteq V$, then V defines a FDAG Δ , such that $\mathcal{R}^{-1}(\Delta)$ is a forest of subtrees of $\mathcal{R}^{-1}(D)$.

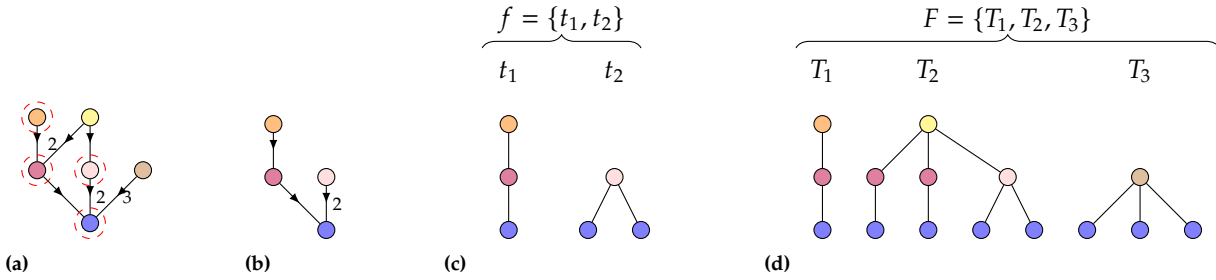


Figure 6.8: Construction of a forest of subtrees from a FDAG. (a) A FDAG D . The set V is circled in red. (b) The FDAG Δ . (c) The forest f compressed by Δ . (d) The forest F compressed by D . One can spot that $t_1 \in \mathcal{S}(T_1)$, $t_2 \in \mathcal{S}(T_2)$ so f is a forest of subtrees of F , and Δ a subFDAG of D .

Proof. We recall that the notation $D[v]$ stands for the subDAG of D rooted in v composed of v and all its descendants $\mathcal{D}(v)$. The notation $\mathfrak{R}^{-1}(D[v])$ stands for the tree compressed by $D[v]$. The proof is in two steps. (i) Remove from D the vertices that do not belong to V ; as there are no arcs that leave V by hypothesis, end up with a FDAG. Let us call Δ this FDAG. (ii) Let ρ be a root of Δ . By construction, ρ is also a vertex in D . Among all roots of D , there exists a root r such that $\rho \in \mathcal{D}(r)$. Therefore, $D[\rho]$ is a subDAG of $D[r]$, and then $t = \mathfrak{R}^{-1}(D[\rho])$ is a subtree of $T = \mathfrak{R}^{-1}(D[r])$ – with $T \in F = \mathfrak{R}^{-1}(D)$. As $\Delta[\rho]$ and $D[\rho]$ are isomorphic, $t \in f = \mathfrak{R}^{-1}(\Delta)$. Therefore we have proved that $\forall t \in f, \exists T \in F, t \in \mathcal{S}(T)$. ✔

32: Not to be confused with *subDAG*, introduced in Section 2.3. A subDAG admits a single root and therefore compresses a single tree, whereas a subFDAG admits several roots and compresses a forest.

We say that the FDAG Δ is a *subFDAG*³² of D . Figure 6.8 provides an example of such a construction.

Enumeration of subFDAGs We now solve the following enumeration problem: given a forest F , find all forests of subtrees of F . Equally, given a FDAG D , find all subFDAGs of D . To address this, we make extensive use of the reverse search technique, adapting the one presented in Section 6.1.

Since a subFDAG is also a FDAG, it admits successors in the enumeration tree defined in Section 6.1. We are interested in those of these successors that are also subFDAGs (if any). In fact, since a subFDAG can be defined from a set of vertices, all one has to do is determine which new vertex can be chosen to expand an existing subFDAG – corresponding to a (\uparrow) or (\uparrow') step. The covering of all added new arcs is implicit in this construction and corresponds to some steps of (\curvearrowright) .

Let Δ be a subFDAG of D and v its last inserted vertex – it is also the vertex with the largest ordering number in Δ . We denote by $S(\Delta)$ the set of all vertices $v' \in D$ that can be added to Δ to expand it to a new subFDAG. Let us call $S(\Delta)$ the *set of candidate vertices* of Δ . More precisely:

Lemma 6.15 $S(\Delta)$ is the set of vertices $v' \in D$ that satisfies both:

- (i) $C(v') \subseteq \Delta$,
- (ii) $\psi(v') > \psi(v)$,

where $\psi(\cdot)$ is the canonical ordering of D .

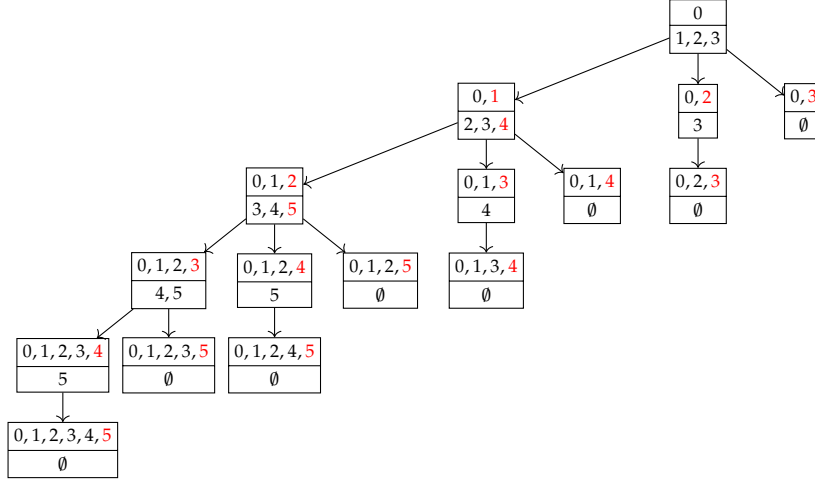


Figure 6.9: Enumeration tree of the subFDAGs of the FDAG of Figure 5.7, using both Algorithm 12 and Algorithm 15. The indices of the vertices correspond to the canonical ordering defined in Figure 5.7. In each vertex, the upper part corresponds to the current subFDAG Δ whereas the lower part stands for the set $S(\Delta)$. Numbers in red indicate what changes for an heir compared to its parent.

Proof. (i) This condition is necessary so that $\Delta' = \Delta \cup \{v'\}$ fulfill the requirements for Proposition 6.14. (ii) This condition is necessary so that Δ' remains a FDAG. As $\psi(v') > \psi(v)$, either $\mathcal{H}(v') = \mathcal{H}(v) + 1$ – then it is a (\uparrow) step – or $\mathcal{H}(v') = \mathcal{H}(v)$ and $C_\psi(v') >_{\text{lex}} C_\psi(v)$ – for a (\uparrow') step. \spadesuit

Algorithm 15: HEIRS

Input: $D, [\Delta, S(\Delta)]$

- 1 Set L to the empty list
 - 2 **for** $s \in S(\Delta)$ **do**
 - 3 Let S' be a copy of $S(\Delta)$
 - 4 $S' \leftarrow S' \setminus \{v' \in S' : C_\psi(v') \leq_{\text{lex}} C_\psi(s)\}$
 - 5 $S' \leftarrow S' \cup \{v' \in D : s \in C(v') \subseteq \Delta \cup \{s\}\}$
 - 6 Add $[\Delta \cup \{s\}, S']$ to L
 - 7 **return** L
-

When $S(\Delta)$ is not empty, picking $s \in S(\Delta)$ ensure that $\Delta' = \Delta \cup \{s\}$ is a subFDAG of D . With respect to the enumeration tree of Section 6.1, Δ is an ancestor of Δ' – but not necessarily its parent, since the steps of (\uparrow) are implicit. Δ' is called an *heir* of Δ . We can in turn calculate $S(\Delta')$, by updating $S(\Delta)$: (i) remove from $S(\Delta)$ all vertices v' such that $\psi(s) > \psi(v')$; (ii) in D , look only after the vertices v' such that $s \in C(v') \subseteq \Delta \cup \{s\}$ and add them to $S(\Delta')$.

If Δ' is an heir of Δ , then by removing the last inserted vertex of Δ' , one can retrieve Δ . This defines a reduction rule f , and therefore an enumeration tree. Algorithm 15 is meant to construct the set $f^{-1}(\Delta)$. Applying Algorithm 12 together with it, and starting from $\Delta = \mathcal{L}(D)$ ³³ – in this case, $S(\Delta)$ is the set of parents of $\mathcal{L}(D)$ of height 1 – permits to enumerate all subFDAGs of D . Figure 6.9 provides an example by enumerating all subFDAGs of the FDAG of Figure 5.7.

³³: where $\mathcal{L}(D)$ designates the leaf of D , i.e. the only vertex without children.

Frequent subFDAG mining problem

We focus here on a particular instance of the frequent mining pattern problem, as defined in Section 5.1. The problem we consider is the following: given a dataset of trees $\mathcal{X} = \{T_1, \dots, T_n\}$, account for forests of

subtrees that appear simultaneously in different T_i 's. In other words, if we denote \mathcal{F}_i the set of all forests of subtrees appearing in the forest formed by $\{T_i\}$, we are interested in the study of $\bigcap_{i \in I_\sigma} \mathcal{F}_i$ where $I_\sigma \subseteq \llbracket 1, n \rrbracket$, such that $\#I_\sigma \geq \sigma \cdot n$.

A first, naive strategy would be to first build the \mathcal{F}_i 's, e.g. by using [Algorithm 15](#) on $\mathfrak{R}(T_i)$, and then construct $\bigcap_{i \in I_\sigma} \mathcal{F}_i$ for all possible choices of I_σ . Obviously, this approach has its weaknesses: (i) many subFDAGs will be enumerated for nothing or in several copies, and (ii) it does not take into account that \mathcal{X} is itself a forest. Our aim is to propose a variant of [Algorithm 15](#) that, applied to $\mathfrak{R}(\mathcal{X})$, would enumerate only subFDAGs appearing in the $\mathfrak{R}(T_i)$'s with a large enough frequency.

Given a forest $F = \{T_1, \dots, T_n\}$ and its DAG compression $D = \mathfrak{R}(F)$, we have to retrieve, for each vertex in D , its origin in the dataset, that is, which tree they come from. This issue has already been addressed in [Connection between a forest and its compressed form](#) (p. 19), under the name of *origin*. We recall that, the origin of a vertex $v \in D$, denoted by $\circ(v)$, represents the set of trees in F for which $\mathfrak{R}^{-1}(D[v])$ is a subtree.

Let Δ be a subFDAG of D . For Δ to compress a forest of subtrees of a tree T_i , it is necessary that $i \in \circ(v)$ for all $v \in \Delta$. Therefore, the set of trees for which Δ compress a forest of subtrees – the *origin* of Δ , denoted by $\circ(\Delta)$ – is equal to

$$\circ(\Delta) = \bigcap_{v \in \Delta} \circ(v).$$

If $\Delta' = \Delta \cup \{s\}$ is an heir of Δ – as defined earlier, then $\circ(\Delta') = \circ(\Delta) \cap \circ(s)$. [Algorithm 15](#) can therefore be refined so that Δ' should be ignored if $\circ(\Delta') = \emptyset$ – as Δ' does not anymore compresses any forest of subtrees actually present in the trees of F .

So far we neglected the threshold σ . We only want to keep subFDAGs that appear in at least $\sigma\%$ of the data. If $\#\circ(\Delta)/\#F < \sigma$, then the successors of Δ are not investigated. Indeed, as $\circ(\cdot)$ is a decreasing function, successors of Δ can not exceed the threshold again.

We can finally introduce [Algorithm 16](#) that solves the frequent subFDAG mining problem for trees. With the notations of [Section 5.1](#), this algorithm builds the set $\{\Delta' \in f^{-1}(\Delta) : \text{freq}(\Delta', F) \geq \sigma\}$, with $\text{freq}(\Delta', F) = \#\circ(\Delta')/\#F$. The set is also built directly, without any posterior filtering, which is suitable as discussed at the beginning of the present subsection.

Algorithm 16: FREQUENTHEIRS

Input: $D = \mathfrak{R}(F)$, $[\Delta, S(\Delta), \circ(\Delta)]$, σ

```

1 Set  $L$  to the empty list
2 for  $s \in S(\Delta)$  do
3   if  $\circ(\Delta) \cap \circ(s) \neq \emptyset$  and  $\#\circ(\Delta) \cap \circ(s) \geq \sigma \cdot \#F$  then
4     Let  $S'$  be a copy of  $S(\Delta)$ 
5      $S' \leftarrow S' \setminus \{v' \in S' : C_\psi(v') \leq_{\text{lex}} C_\psi(s)\}$ 
6      $S' \leftarrow S' \cup \{v' \in D : s \in C(v') \subseteq D_0 \cup \{s\}\}$ 
7     Add  $[\Delta \cup \{s\}, S', \circ(\Delta) \cap \circ(s)]$  to  $L$ 
8 return  $L$ 

```

We stated earlier that we wanted to avoid generating unnecessary or multiple copies of subFDAGs, which is achieved with [Algorithm 16](#). We now empirically study what we have gained from this, by comparing the use of [Algorithm 16](#) on $D = \mathfrak{R}(F)$, with the use of [Algorithm 15](#) on each $\mathfrak{R}(T_i)$. As in [Branching factor \(p. 78\)](#), we generated 1 000 random FDAGs D_k , 10 repetitions for each $k \in \{1, \dots, 100\}$, creating D_k as in [Definition 6.5](#). We assume $D_k = \mathfrak{R}(f)$ where $f = \{\mathfrak{R}^{-1}(D_k[r]) : r \in \mathcal{R}(D_k)\}$. For each D_k , we have computed the quotient

$$Q(D_k) = \frac{\#\{\text{subFDAGs of } D_k \text{ enumerated via Algorithm 16}\}}{\sum_{r \in \mathcal{R}(D_k)} \#\{\text{subFDAGs of } D_k[r] \text{ enumerated via Algorithm 15}\}}$$

with parameter $\sigma = 0$ when using [Algorithm 16](#). The results are provided in [Figure 6.10](#). Despite a rather marked variability, there is a general trend of decreasing as the number of vertices increases. We obtain fairly low quotients, around 20%, quite quickly. Given the combinatorial explosion of the objects to be enumerated, such an advantage is of the greatest interest.

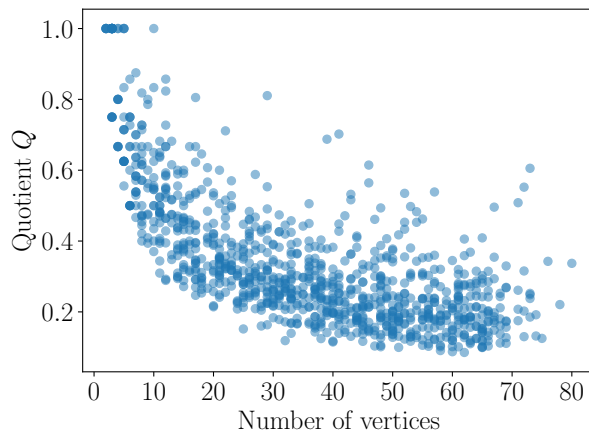


Figure 6.10: Quotient $Q(D)$ according to the number of vertices of D . Here 1 000 random FDAGs are displayed.

THE SUBTREE KERNEL REVISITED

When you're outnumbered by trees
your perspective shifts.

—————
Jessica Marie Baumgartner

In this chapter and the next one, we focus on the issue of statistical analysis of tree data, which is difficult given their non-Euclidean nature.

In [Section 7.1](#), we introduce kernel methods, a family of techniques particularly useful for dealing with tree data. In particular, we introduce the subtree kernel, on which our analysis focuses.

[Section 7.2](#) is dedicated to the introduction of a stochastic tree model that allows us to perform a performance analysis of said kernel, and in particular to infer a desirable property on one of the kernel parameters – named the weight.

Parts of this chapter are reproduced from [39], in particular [Section 7.2](#).

- 7.1 Kernel methods 93
 - Kernel trick 94
 - Tree kernels 95
- 7.2 Theoretical study 96
 - Two trees as different as possible 96
 - A stochastic model of 2-classes tree data 97
 - Theoretical guarantees on the subtree kernel 97
 - Weight of leaves 99

7.1 Kernel methods

Suppose we have data (of any nature) and we want to find relations and structures in this data – e.g. clusters, correlations, etc. There is no guarantee that these relationships, if they exist, are immediately accessible. Sometimes it is better not to use the raw data, but rather to transform them, via an user-specified *feature map*. They are sent to a (inner product) *feature space*, in which it is expected that relations between the data will appear more naturally. This approach is particularly relevant for non-Euclidean data (such as trees), since by choosing an appropriate feature space, one can use standard statistical methods to process them. A very simple example is presented in [Figure 7.1](#).

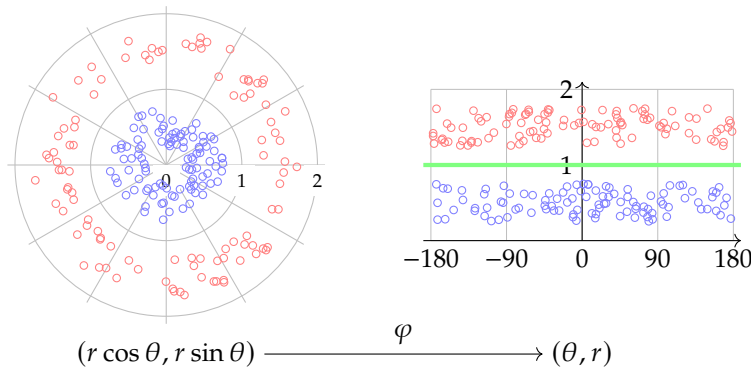


Figure 7.1: Suppose we want to build a classifier to separate the red and blue data (on the left). However, imagine the algorithm we have only allows to build linear classifiers: there is no hope to succeed. On the other hand, via a well-chosen feature map φ , one can send the data into a space in which they are linearly separable – see the green line (on the right).

On the other hand, choosing the proper feature space and finding out the mapping might be very difficult. Furthermore, the curse of

[39]: Azaïs et al. (2020), ‘The weight function in the subtree kernel is decisive’

dimensionality takes place and the feature space may be extremely big, therefore impossible to use. Fortunately, a wide range of prediction algorithms do not need to access that feature space, but only the inner product between elements of the feature space. Building a function, called a kernel, that simulates an inner product in an implicit feature space, frees us from constructing a mapping.

Kernel trick

Let us first define kernels. Let \mathcal{X} be the space of considered data.

Definition 7.1 A symmetric function $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ is said to be a (positive semi-definite) kernel on \mathcal{X} if, given $n \in \mathbb{N}$ and $c_1, \dots, c_n \in \mathbb{R}$,


$$\sum_{i=1}^n \sum_{j=1}^n c_i c_j K(x_i, x_j) \geq 0 \quad (7.1)$$

for any $x_1, \dots, x_n \in \mathcal{X}$.

Equivalently, K is a kernel if, for any $x_1, \dots, x_n \in \mathcal{X}$, the Gram matrix [95] expressed as $[K(x_i, x_j)]_{1 \leq i, j \leq n}$ is positive semi-definite.

The main interest of kernels comes from Mercer's theorem [96].

Theorem 7.1 (Mercer) If K is a (positive semi-definite) kernel, then there exists a (inner product) feature space \mathcal{Y} and a mapping $\varphi : \mathcal{X} \rightarrow \mathcal{Y}$ such that, for any $x, y \in \mathcal{X}$, $K(x, y) = \langle \varphi(x), \varphi(y) \rangle_{\mathcal{Y}}$.

Proof. The proof can be found in [33]. 

This technique is known as the kernel trick. It is sufficient to construct a function K verifying Equation 7.1 on the data space (even non-Euclidean), to be certain of the existence of a feature space, on which K plays the role of an inner product. From then on, all the statistical methods which only use the inner product of the data become accessible, without even needing to explicitly exhibit the feature space nor the mapping allowing to access it.

Algorithms that can use kernels include Support Vector Machines (SVM), Principal Components Analysis (PCA) and many others. We refer the reader to the books [33, 97–99] and the references therein for more detailed explanations of theory and applications of kernels.

The construction of kernels is facilitated by virtue of the following proposition, whose proof can also be found in [33].

Proposition 7.2 Let K_1, K_2 be two kernels on \mathcal{X} and $a \in \mathbb{R}_+$. Let also $(K_n)_{n \in \mathbb{N}}$ be a sequence of kernels on \mathcal{X} . Then, are also kernels on \mathcal{X} :

- ▶ $K_1 + K_2$, aK_1 and $K_1 K_2$,
- ▶ $K = \lim_{n \rightarrow \infty} K_n$.

[95]: Horn et al. (2012), *Matrix analysis*

[96]: Mercer (1909), 'XVI. Functions of positive and negative type, and their connection the theory of integral equations'

[33]: Cristianini et al. (2000), *An introduction to support vector machines and other kernel-based learning methods*

[97]: Schölkopf et al. (2001), *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*

[98]: Shawe-Taylor et al. (2004), *Kernel methods for pattern analysis*

[99]: Hastie et al. (2009), *The elements of statistical learning: data mining, inference, and prediction*

Let $x, z \in \mathcal{X}$, $f : \mathcal{X} \rightarrow \mathcal{R}$, $\varphi : \mathcal{X} \rightarrow \mathcal{Y}$, K_3 a kernel on \mathcal{Y} and B a positive semi-definite symmetric matrix. Those are also kernels on \mathcal{X} :

- ▶ $K(x, z) = f(x)f(z)$,
- ▶ $K(x, z) = K_3(\varphi(x), \varphi(z))$,
- ▶ $K(x, z) = x^\top Bz$ – with $\mathcal{X} = \mathbb{R}^d$.

Tree kernels

To use kernel-based algorithms with tree data, one needs to design kernel functions adapted to trees. Convolution kernels, introduced by Haussler [34], measure the similarity between two complex combinatorial objects based on the similarity of their substructures. Based on this strategy, many authors have developed convolution kernels for trees, among them the subset tree kernel [35], the subtree kernel [37] and the subpath kernel [100].

A recent state-of-the-art on kernels for trees can be found in the thesis of Da San Martino [36], as well as original contributions on related topics. Let us also mention [101], which establishes a unifying formalism to express most convolution kernels on trees.

In this chapter and the next, we focus on the subtree kernel as defined by [37]. The subtree kernel is a convolution kernel on trees for which the similarity between two trees is measured through the similarity of their subtrees. A subtree kernel K on trees (ordered or not, unlabeled³⁴) is defined as,

$$\forall T_1, T_2 \in \mathcal{T}, K(T_1, T_2) = \sum_{t \in \mathcal{T}} w_t \kappa(N_t(T_1), N_t(T_2)), \quad (7.2)$$

where w_t is a weight associated to the tree t , $N_t(T)$ ³⁵ counts the number of subtrees of T that are isomorphic to t and κ is a kernel function on \mathbb{N} , \mathbb{Z} or \mathbb{R} . Assuming $\kappa(0, n) = \kappa(n, 0) = 0$, Equation 7.2 becomes

$$K(T_1, T_2) = \sum_{t \in \mathcal{S}(T_1) \cap \mathcal{S}(T_2)} w_t \kappa(N_t(T_1), N_t(T_2)), \quad (7.3)$$

making the sum finite. Indeed, all the subtrees $t \in \mathcal{T} \setminus (\mathcal{S}(T_1) \cap \mathcal{S}(T_2))$ do not count in the sum Equation 7.2. Here, as for [37], we assume that $\kappa(n, m) = nm$, then we get the subtree kernel as introduced in [37].

Definition 7.2 The subtree kernel between two trees T_1 and T_2 is defined as

$$K(T_1, T_2) = \sum_{t \in \mathcal{S}(T_1) \cap \mathcal{S}(T_2)} w_t N_t(T_1) N_t(T_2). \quad (7.4)$$

An example of kernel computation is provided in Figure 7.2.

The weight function $t \mapsto w_t$ is the only remaining parameter to be tuned. In the literature, the weight is usually assumed to be a function of a quantity measuring the “size” of t , in particular its height $\mathcal{H}(t)$. Then w_t is taken as an exponential decay of this quantity, $w_t = \lambda^{\mathcal{H}(t)}$ for some $\lambda \in [0, 1]$ – as in [35–37, 100] and [102]. This choice can be justified in the following manner. If a subtree t is counted in the kernel, then all its

[34]: Haussler (1999), *Convolution kernels on discrete structures*

[35]: Collins et al. (2001), ‘Convolution kernels for natural language’

[37]: Vishwanathan et al. (2004), ‘Fast kernels for string and tree matching’

[100]: Kimura et al. (2011), ‘A subpath kernel for rooted unordered trees’

[36]: Da San Martino (2009), ‘Kernel methods for tree structured data’

[101]: Shin et al. (2010), ‘A Generalization of Haussler’s Convolution Kernel—Mapping Kernel and Its Application to Tree Kernels’

34: The definition naturally extends to labeled trees if we impose equality of labels in the isomorphism.

35: $N_t(T)$ is exactly equal, with the notations of *Connection between a forest and its compressed form* (p. 19), to the presence $\pi([t])$ of vertex $[t]$ in the DAG $\mathfrak{R}(T)$. This will be exploited in upcoming Chapter 8.

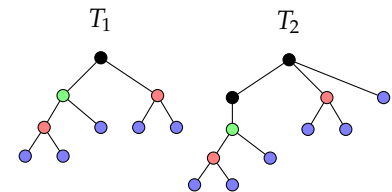


Figure 7.2: Computation of the subtree kernel on two trees T_1 (left) and T_2 (right). Only subtrees present in both T_1 and T_2 have been colored according to their equivalence class. With Equation 7.3, the value of the kernel is $K(T_1, T_2) = w_{\bullet} \kappa(5, 6) + w_{\bullet} \kappa(2, 2) + w_{\bullet} \kappa(1, 1)$. Adopting the definition of Equation 7.4, the kernel is then expressed as $K(T_1, T_2) = 30w_{\bullet} + 4w_{\bullet} + w_{\bullet}$. Finally, with $w_t = \lambda^{\mathcal{H}(t)}$, the kernel becomes $K(T_1, T_2) = 30 + 4\lambda + \lambda^2$.

[102]: Aiolli et al. (2006), ‘Fast on-line kernel learning for trees’

subtrees are also counted. Then an exponential decay counterbalances the growth of subtrees. On the other hand, as can be seen from the example of [Figure 7.2](#), the contribution of leaves is generally preponderant over the contribution of other subtrees since $w_{\bullet} = \lambda^0 = 1$. One of the goals of [Section 7.2](#) is to show that it is better to impose $w_{\bullet} = 0$.

7.2 Theoretical study

In this section, we define a stochastic model of 2-classes tree data. From this ideal data set, we prove the efficiency of the subtree kernel and derive the sufficient size of the training data set to get a classifier with a given prediction error. We also state on this simple model that the weight of leaves should always be 0. We emphasize that this study is valid for both ordered and unordered trees.

Two trees as different as possible

Our goal is to build a 2-classes data set of random trees. To this end, we first define two typical trees T_0 and T_1 that are as different as possible in terms of subtree kernel.

Let T_0 and T_1 be two trees that fulfill the following conditions:

- (i) $\forall i \in \{0, 1\}, \forall u, v \in T_i \setminus \mathcal{L}(T_i)$, if $u \neq v$ then $T_i[u] \neq T_i[v]$, i.e., two subtrees of T_i are not isomorphic (except leaves).
- (ii) $\forall u \in T_0 \setminus \mathcal{L}(T_0), \forall v \in T_1 \setminus \mathcal{L}(T_1)$, $T_0[u] \neq T_1[v]$, i.e., any subtree of T_0 is not isomorphic to a subtree of T_1 (except leaves).

These two assumptions ensure that the trees T_0 and T_1 are as different as possible. Indeed, it is easy to see that

$$K(T_0, T_1) = w_{\bullet} \# \mathcal{L}(T_0) \# \mathcal{L}(T_1),$$

which is the minimal value of the kernel and where w_{\bullet} is the weight of leaves. We refer to [Figure 7.3](#) for an example of trees that satisfy these conditions.

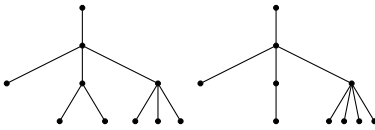


Figure 7.3: Two trees T_0 and T_1 that fulfill conditions (i) and (ii).

Trees of class i will be obtained as random editions of T_i . In the sequel, $T_i(v \mapsto \tau)$ denotes the tree T_i in which the subtree rooted at v has been replaced by τ . These random edits will tend to make trees of class 0 closer to trees of class 1. To this end, we introduce the following additional assumption. Let (τ_h) a sequence of trees such that $\mathcal{H}(\tau_h) = h$.

- (iii) Let $u \in T_0$ and $v \in T_1$. We consider the edited trees $T'_0 = T_0(u \mapsto \tau_{\mathcal{H}(u)})$ and $T'_1 = T_1(v \mapsto \tau_{\mathcal{H}(v)})$. Then, $\forall u' \in T'_0 \setminus (\tau_{\mathcal{H}(u)} \cup \mathcal{L}(T'_0))$, $\forall v' \in T'_1 \setminus (\tau_{\mathcal{H}(v)} \cup \mathcal{L}(T'_1))$, $T'_0[u'] \neq T'_1[v']$.

In other words, if one replaces subtrees of T_0 and T_1 by subtrees of the same height, then any subtree of T_0 is not isomorphic to a subtree of T_1 (except the new subtrees and leaves). This means that the similarity between random edits of T_0 and T_1 will come only from the new subtrees and not from collateral modifications. We refer to [Figure 7.4](#) for an example of trees that satisfy these conditions.

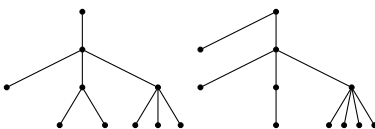


Figure 7.4: Two trees T_0 and T_1 that fulfill conditions (i), (ii) and (iii).

A stochastic model of 2-classes tree data

From now on, we assume that, for any $h > 0$, τ_h is not a subtree of T_0 or T_1 . For the sake of simplicity, T_0 and T_1 have the same height H . In addition, if $u \in T_i$ then T_i^u denotes $T_i(u \mapsto \tau_{\mathcal{H}(u)})$.

The stochastic model of 2-classes tree data that we consider is defined from the binomial distribution $P_\rho = \mathcal{B}(H, \rho/H)$ on support $\{0, \dots, H\}$ with mean $\bar{P}_\rho = \rho$. The parameter $\rho \in [0, H]$ is fixed. In the data set, class i is composed of random trees T_i^u , where the vertex u has been picked uniformly at random among vertices of height h in T_i , where h follows P_ρ . Furthermore, the considered training data set is well-balanced in the sense that it contains the same number of data of each class.

Intuitively, when ρ increases, the trees are more degraded and thus two trees of different class are closer. ρ somehow measures the similarity between the two classes. In other words, the larger ρ , the more difficult is the supervised classification problem.

Remark 7.1 The structure of a markup document such as an HTML page can be described by a tree (see [Preliminaries \(p. 106\)](#) and especially [Figure 8.2](#) for more details). In this context, the tree $T_i, i \in \{0, 1\}$, can be seen as a model of the structure of a webpage template. By assumption, the two templates of interest are as different as possible. However, they are completed in a similar manner, for example to present the same content in two different layouts. Edition of the templates is modeled by random edit operations. They tend to bring trees from different templates closer.

Theoretical guarantees on the subtree kernel

[103] have introduced a theory that describes the effectiveness of a given kernel in terms of similarity-based properties. A similarity function over \mathcal{X} is a pairwise function $K : \mathcal{X}^2 \rightarrow [-1, 1]$ [103, Definition 1]. It is said (ϵ, γ) -strongly good [103, Definition 4] if, with probability at most $1 - \epsilon$,

$$\mathbb{E}_{x', y} [K(x, x') - K(x, y)] \geq \gamma,$$

where $\text{label}(x) = \text{label}(x') \neq \text{label}(y)$. From this definition, the authors derive the following simple classifier: the class of a new data x is predicted by 1 if x is more similar on average to points in class 1 than to points in class 0, and 0 otherwise – see [Algorithm 17](#). In addition, they prove [103, Theorem 1] that a well-balanced training data set of size $32/\gamma^2 \log(2/\delta)$ is sufficient so that, with probability at least $1 - \delta$, the above algorithm applied to an (ϵ, γ) -strongly good similarity function produces a classifier with error at most $\epsilon + \delta$.

We aim to prove comparable results for the subtree kernel – which is not a similarity function. To this end, we focus for $i \in \{0, 1\}$ on

$$\Delta_x^i = \mathbb{E}_{u, v} [K(T_i^x, T_i^u) - K(T_i^x, T_{1-i}^v)]. \quad (7.5)$$

[103]: Balcan et al. (2008), ‘A theory of learning with similarity functions’

Algorithm 17: CLASSIFIER

Input: $x \in \mathcal{X}$

1 Let S_i be the set of known data of class i .

2 **if**

$$\frac{1}{\#S_1} \sum_{y \in S_1} K(x, y) \geq \frac{1}{\#S_0} \sum_{y \in S_0} K(x, y)$$

then

3 | **return** 1

4 **else**

5 | **return** 0


We emphasize that the two following results ([Proposition 7.3](#) and [Corollary 7.4](#)) assume that the weight of leaves ω_\bullet is 0. For the sake of readability, we introduce the following notations, for any $0 \leq h \leq H$ and $i \in \{0, 1\}$,

$$\begin{aligned} K_{i,h} &= \max_{\{u \in T_i : \mathcal{H}(u)=h\}} K(T_i[u], T_i[u]), \\ C_{i,h} &= \frac{K(T_i, T_i) - K_{i,h}}{\#\mathcal{L}(T_i)}, \\ G_\rho(h) &= 1 - \sum_{k=h+1}^H P_\rho(k). \end{aligned}$$

The following results are expressed in terms of a parameter $0 \leq h < H$. The statement is then true with probability $G_\rho(h)$. This is equivalent to state a result that is true with probability $1 - \epsilon$, for any $\epsilon > 0$ – except that here, ϵ is not arbitrary but taken from a discrete range of values.

Proposition 7.3 *If $w_{T_i} > 0$ then $\Delta_x^i = 0$ if and only if $x = \mathcal{R}(T_i)$. In addition, if $\rho > H/2$, for any $0 \leq h < H$, with probability $G_\rho(h)$, one has*

$$\Delta_x^i \geq P_\rho(0)C_{i,h}. \quad (7.6)$$

Proof. The proof lies in [Appendix A.5](#). 

This result shows that the two classes can be well-separated by the subtree kernel. The only data that can not be separated are the trees completely edited. In addition, the lower-bound in [Equation 7.6](#) is of order $H \exp(-\rho)$ (up to a multiplicative constant).


Corollary 7.4 *For any $0 \leq h \leq H$, a well-balanced training data set of size*

$$\frac{2 \max_i K(T_i, T_i)^2 \exp(2\rho)}{\min_i C_{i,h}^2} \frac{\exp(2\rho)}{H^2} \log\left(\frac{2}{\delta}\right)$$

is sufficient so that, with probability at least $1 - \delta$, the aforementioned classification [Algorithm 17](#) produces a classifier with error at most $1 - G_\rho(h) + \delta$.

Proof. The proof is based on the demonstration of [[103](#), Theorem 1]. However, in our setting, the kernel K is bounded by $\max_i K(T_i, T_i)$ and not by 1. Consequently, by Hoeffding bounds, the sufficient size of the training data set is of order

$$2 \log\left(\frac{2}{\delta}\right) \frac{\max_i K(T_i, T_i)^2}{\gamma^2}, \quad (7.7)$$

where γ can be read in [Proposition 7.3](#), $\gamma = P_\rho(0)C_{i,h} \geq P_\rho(0) \min_i C_{i,h}$. The coefficient 2 lies because we consider here the total size of the data set and not only the number of examples of each class. Together with $P_\rho(0) \sim H \exp(-\rho)$, we obtain the expected result. 

Weight of leaves

Here K^+ is the subtree kernel obtained from the weights used in the computation of K together with a positive weight on leaves, $w_\bullet > 0$. We aim to show that K^+ separates the two classes less than K . $\Delta_x^{+,i}$ denotes the conditional expectation Equation 7.5 computed from K^+ .

Proposition 7.5 For any $x \in T_i$,

$$\Delta_x^{+,i} = \Delta_x^i + w_\bullet \# \mathcal{L}(T_i[x]) D_{i,1-i},$$


where $D_{i,1-i} = \mathbb{E}_{u,v} [\# \mathcal{L}(T_i^u) - \# \mathcal{L}(T_{1-i}^v)]$.

Proof. We have the following decomposition, for any trees T_1 and T_2 ,

$$K^+(T_1, T_2) = K(T_1, T_2) + w_\bullet \# \mathcal{L}(T_1) \# \mathcal{L}(T_2),$$

in light of Equation 7.4. Thus, with Equation 7.5,

$$\begin{aligned} \Delta_x^{+,i} &= \mathbb{E}_{u,v} [K(T_i^x, T_i^u) + w_\bullet \# \mathcal{L}(T_i^x) \# \mathcal{L}(T_i^u) - K(T_i^x, T_{1-i}^v) - w_\bullet \# \mathcal{L}(T_i^x) \# \mathcal{L}(T_{1-i}^v)] \\ &= \Delta_x^i + \mathbb{E}_{u,v} [w_\bullet \# \mathcal{L}(T_i^x) (\# \mathcal{L}(T_i^u) - \# \mathcal{L}(T_{1-i}^v))], \end{aligned}$$

which ends the proof. 

The sufficient number of data provided in Corollary 7.4 is obtained in Equation 7.7 through the square ratio of $\max_i K(T_i, T_i)$ over $\min_i \Delta_x^i$. First, it should be noticed that $K^+(T_i, T_i) > K(T_i, T_i)$. In addition, by virtue of Proposition 7.5, either $\Delta_x^{+,0} \leq \Delta_x^0$ or $\Delta_x^{+,1} \leq \Delta_x^1$ (and the inequality is strict if trees of classes 0 and 1 have not the same number of leaves on average). Consequently,

$$\min_i \Delta_x^{+,i} \leq \min_i \Delta_x^i,$$

and thus the sufficient number of data mentioned above is minimum for $w_\bullet = 0$.

Remark 7.2 The results stated in this section establish that the subtree kernel is more efficient when the weight of leaves is 0. It should be placed in perspective with the exponential weighting scheme of the literature [35–37, 100, 102] for which the weight of leaves is maximal. We conjecture that the accuracy of the subtree kernel should be in general improved by imposing a null weight to any subtree present in two different classes. This can not be established from the model for which the only such subtrees are the leaves. Relying on this, one of the objectives of the next chapter is to develop a learning method for the weight function that improves in practice the classification results.

[102]: Aiolli et al. (2006), ‘Fast on-line kernel learning for trees’

[35]: Collins et al. (2001), ‘Convolution kernels for natural language’

[36]: Da San Martino (2009), ‘Kernel methods for tree structured data’

[100]: Kimura et al. (2011), ‘A subpath kernel for rooted unordered trees’

[37]: Vishwanathan et al. (2004), ‘Fast kernels for string and tree matching’

A new framework for computing the subtree kernel

8

All forests have their own personality.

Charles de Lint

In the previous chapter, we introduced the subtree kernel and showed, in the framework of a stochastic model, that it was preferable to set the weight of the leaves to zero.

In this chapter, we briefly present the existing methods to compute the subtree kernel before introducing our framework, in Section 8.1, which computes the kernel from the DAG compression of the trees, and that allows us to choose an arbitrary weight function. In particular, we propose the discriminance weight function, which sets the leaf weights to zero and learns from the data what weight to assign to each subtree, in a supervised classification context.

We then apply our method to 8 different datasets in Section 8.2, for which we show that the discriminance weight function can improve the performance of the kernel. Finally, the chapter ends with a discussion of the interests of our approach in Section 8.3.

The majority of the chapter is reproduced from [39].

8.1 Framework	101
State of the art	101
Context	102
Kernel computation on DAGs	103
Discriminance weight function	104
8.2 Real data analysis	105
Preliminaries	106
Prediction of the language of a Wikipedia article from its topology	108
Markup documents data sets	111
Biological data sets	113
LOGML	114
8.3 Interest of the DAG approach	115
Learning the weight function	115
Computation time	117

[39]: Azais et al. (2020), ‘The weight function in the subtree kernel is decisive’

8.1 Framework

State of the art

We present here an overview of the existing methods to compute the subtree kernel, whose general formula is recalled below.

$$K(T_1, T_2) = \sum_{t \in \mathcal{S}(T_1) \cap \mathcal{S}(T_2)} w_t N_t(T_1) N_t(T_2). \quad (\text{Equation 7.4})$$

Substrings The approach of [37] is based on string representations of (labeled or not) *ordered* trees, as seen in [Encoding of trees](#) (p. 11). Recall that this allows a tree to be encoded as a sequence of brackets. In this context, a substring with balanced brackets corresponds exactly to a subtree. Since the authors of [37] introduce a string kernel operating on substrings, they notice that by giving a zero weight to substrings with unbalanced brackets, they retrieve exactly the subtree kernel.

[37]: Vishwanathan et al. (2004), ‘Fast kernels for string and tree matching’

Given two strings x and y , y is converted into a suffix tree (which enumerates all the suffixes of a string). This tree allows to check which substrings of x also exist in y , which then enables the calculation of the kernel. We refer the reader to [37] for complete details. The weights

allowed by the algorithm are typically exponential, constant, or binary (0 or 1).

Recursive computation Given a tree T , the subtree rooted in u is defined as u and all its descendants. We construct a *subset tree*, rooted in u , in the following way: either we take all the children of u , or none of them – and then recursively on the nodes kept. If we choose consistently to take all the children of all the nodes encountered, we get the subtree rooted in u .

[35]: Collins et al. (2001), ‘Convolution kernels for natural language’

The subset tree kernel, introduced in [35], is defined exactly as the subtree kernel, but where the subtrees are replaced by the subset tree we just defined. In this paper, a recursive method for computing the subset tree kernel is proposed. Noting that subtrees are a special case of subset trees, it turns out that the proposed recursive formula can be modified to compute, instead, the subtree kernel – see [36]. This recursive formula imposes an exponential or constant weight. Here also, trees are ordered.

Computing $K(T_1, T_2)$ has complexity $O(\#T_1 + \#T_2)$ for the first option, and $O(\#T_1\#T_2)$ for the second one. Also, to the best of our knowledge, the case of unordered trees has only been considered through the arbitrary choice of a sibling order. Finally, labeled trees are treated by imposing label equality.

Our proposition Our goal in this chapter is to propose a new way of computing the kernel, based on the explicit enumeration of $\mathcal{S}(T_1) \cap \mathcal{S}(T_2)$. In fact, we already know a tool that allows to enumerate the subtrees, i.e. DAG compression; so we propose to compute the kernel directly from the DAG compression of trees – the forest when treating a dataset.

[102]: Aiolli et al. (2006), ‘Fast on-line kernel learning for trees’

[36]: Da San Martino (2009), ‘Kernel methods for tree structured data’

It is not the first time that DAG compression is used in a kernel computation context since the authors of [36, 102] extensively use DAG reduction – with a focus on ordered trees. However, the method developed by [36, 102] is only adapted to exponential weights (see equations (3.12) and (6.2) from [36]).

Our ambition is to propose a general framework capable of handling ordered or unordered, labeled³⁶ or unlabeled trees, and where the weight function can be chosen arbitrarily. Finally, in [102, Section 4], the time-complexities are studied only from a numerical point of view, while we state theoretical results.

36: Where we impose equality of labels in the isomorphisms.

Context

We place ourselves in a context of supervised classification. We consider a data set composed of two parts: the training data set $\mathcal{X}_{\text{train}} = (T_1, \dots, T_n)$ where the class of each tree is assumed to be known, and the data set $\mathcal{X}_{\text{pred}} = (T_{n+1}, \dots, T_N)$ whose classes we want to predict.

Our aim is to compute two Gram matrices $G = [K(T_i, T_j)]_{i,j}$, where:

- ▶ $(i, j) \in \mathcal{X}_{\text{train}} \times \mathcal{X}_{\text{train}}$ for the training matrix G_{train} ;
- ▶ $(i, j) \in \mathcal{X}_{\text{pred}} \times \mathcal{X}_{\text{train}}$ for the prediction matrix G_{pred} .

SVM algorithms will use G_{train} to learn their classifying rule, and G_{pred} to make predictions [33]. Other algorithms, such as kernel PCA, would also require to compute a Gram matrix before processing [97].

[33]: Cristianini et al. (2000), *An introduction to support vector machines and other kernel-based learning methods*

Kernel computation on DAGs

Our goal here is to compute the subtree kernel $K(T_i, T_j)$ between two trees T_i and T_j from the DAG compression of T_i and T_j – as defined in Section 2.3. In particular, we compress all the trees into a same DAG, using the method developed in Section 2.4, by introducing an artificial root. We denote by $\Delta = \mathfrak{R}(\mathcal{X}_{\text{train}} \cup \mathcal{X}_{\text{pred}})$ the DAG reduction of the data set and, for any $1 \leq i \leq N$, $D_i = \mathfrak{R}(T_i)$. Since the root $\mathcal{R}(\Delta)$ is artificial, and to simplify the notations, in the following we will note $v \in \Delta$ as an abuse of language for $v \in \Delta \setminus \mathcal{R}(\Delta)$.

[97]: Schölkopf et al. (2001), *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*

DAG computation of the subtree kernel requires to annotate the DAG with different pieces of information. We use extensively the notions of origin and presence, defined in [Connection between a forest and its compressed form](#) (p. 19), whose definition we briefly recall here:

- ▶ The *origin* $o(v)$ of a vertex $v \in \Delta$ is defined at the subset of $\llbracket 1, N \rrbracket$ so that $i \in o(v) \iff v \in D_i$ – in other words, $\mathfrak{R}^{-1}(\Delta[v]) \in \mathcal{S}(\mathfrak{R}^{-1}(D_i))$.
- ▶ The *presence vector* $\pi(v)$ of a vertex $v \in \Delta$ is defined, for each $i \in \llbracket 1, N \rrbracket$, as $\pi_i(v) = \#\{u \in T_i : [u] = v\}$. Equivalently, denoting $t = \mathfrak{R}^{-1}(\Delta[v])$, $\pi_i(v) = N_i(T_i)$ – with the notation of Equation 7.2.

Both origin and presence can be computed in linear time in one exploration of Δ – as stated in [Connection between a forest and its compressed form](#) (p. 19).

DAG weighting The last thing that we lack to compute the kernel is the weight function. Remember that it is defined for trees as a function $w : \mathcal{T} \rightarrow \mathbb{R}^+$. As we only need to know the weights of the subtrees associated with vertices of Δ , we define the weight function for DAG as, for any $v \in \Delta$, $\omega_v = w_t$, where $t = \mathfrak{R}^{-1}(\Delta[v])$.

Note that ω_v can be computed during the same exploration of Δ for comparing $o(\cdot)$ and $\pi(\cdot)$, as soon as ω_v depends only on a topological property of v that can be computed recursively, such as $\mathcal{H}(\cdot)$ – see Equation 2.1.

DAG computation of the subtree kernel We introduce the matching subtrees function \mathcal{M} as

$$\begin{aligned} \mathcal{M}: \{1, \dots, N\}^2 &\rightarrow 2^\Delta \\ (i, j) &\mapsto \{v \in \Delta : \{i, j\} \subseteq o(v)\} \end{aligned}$$

where 2^Δ is the powerset of the vertices of Δ . Note that \mathcal{M} is symmetric. This leads us to the following proposition.

Proposition 8.1 For any $T_i, T_j \in \mathcal{X}_{\text{train}} \cup \mathcal{X}_{\text{pred}}$, we have

$$K(T_i, T_j) = \sum_{v \in \mathcal{M}(i, j)} \omega_v \pi_i(v) \pi_j(v).$$

Proof. By construction, it suffices to show that $\mathfrak{R}(\mathcal{S}(T_i) \cap \mathcal{S}(T_j)) = \mathcal{M}(i, j)$. Let $t \in \mathcal{S}(T_i) \cap \mathcal{S}(T_j)$. Then $\mathfrak{R}(t) \in \mathfrak{R}(T_i)$ and $\mathfrak{R}(t) \in \mathfrak{R}(T_j)$. Necessarily, $\mathfrak{R}(t) \in \Delta$ and $\{i, j\} \subseteq o(\mathfrak{R}(t))$. So $\mathfrak{R}(t) \in \mathcal{M}(i, j)$. Reciprocally, let $v \in \mathcal{M}(i, j)$. We denote $t = \mathfrak{R}^{-1}(v)$. As $\{i, j\} \subseteq o(v)$, then $t \in \mathcal{S}(T_i) \cap \mathcal{S}(T_j)$. ✍

Remark 8.1 \mathcal{M} can be created in $\mathcal{O}(N^2\#\Delta)$ within one exploration of Δ and allows afterward computations of the subtree kernel $K(T_i, T_j)$ in $\mathcal{O}(\#\mathcal{M}(i, j)) = \mathcal{O}(\min(\#D_i, \#D_j))$, which is more efficient than the $\mathcal{O}(\#T_i + \#T_j)$ algorithm proposed by [37] (the complexity is announced by [100]). However, since the whole process through [Algorithm 3](#) or [Algorithm 4](#) is costly, the global method that we propose here is not faster than existing algorithms. Nonetheless, our algorithm is particularly adapted to repeated computations from the same data, e.g., for tuning parameters. Indeed, once \mathcal{M} and Δ have been created, they can be stored and are ready to use. An illustration of this property is provided from experimental data in [Figure 8.15](#).

[37]: Vishwanathan et al. (2004), ‘Fast kernels for string and tree matching’

[100]: Kimura et al. (2011), ‘A subpath kernel for rooted unordered trees’

Discriminance weight function

For a given probability level and a given classification error, and under the stochastic model of [Section 7.2](#), we state in [Weight of leaves \(p. 99\)](#) that the sufficient size of the training data set is minimum when the weight of leaves is 0. In other words, counting the leaves, which are the only subtrees that appear in both classes, does not provide any relevant information to the classification problem associated with this model. As mentioned in [Remark 7.2](#), we conjecture that, in a more general model, this result would be true for any subtree present in both classes. In this section, we propose to rely on this idea by defining a new weight function, learned from the data and called discriminance weight that assigns a large weight to subtrees, that help to discriminate the classes, i.e., that are present or absent in exactly one class, and a low weight otherwise.

The training data set is divided into two parts: $\mathcal{X}_{\text{weight}} = (T_1, \dots, T_m)$ to learn the weight function, and $\mathcal{X}_{\text{class}} = (T_{m+1}, \dots, T_n)$ to estimate the Gram matrix. For the sake of readability, Δ denotes the DAG reduction of the whole data set, including $\mathcal{X}_{\text{weight}}$, $\mathcal{X}_{\text{class}}$ and $\mathcal{X}_{\text{pred}}$. In addition, we assume that the data are divided into K classes numbered from 1 to K .

For any vertex $v \in \Delta$, we define the vector ρ_v of length K as,

$$\forall 1 \leq k \leq K, \rho_v(k) = \frac{1}{\#\mathcal{C}_k} \sum_{T_i \in \mathcal{C}_k} \mathbb{1}_{\{i \in o(v)\}},$$

where $(\mathcal{C}_k)_{1 \leq k \leq K}$ forms a partition of $\mathcal{X}_{\text{weight}}$ such that $T_i \in \mathcal{C}_k$ if and only if T_i is in class k . In other words, $\rho_v(k)$ is the proportion of data

in class k that contain the subtree $\mathfrak{R}^{-1}(\Delta[v])$. Therefore, ρ_v belongs to the K -dimensional hypercube. It should be noticed that ρ_v is a vector of zeros as soon as $\mathfrak{R}^{-1}(\Delta[v])$ is not a subtree of a tree of $\mathcal{X}_{\text{weight}}$.

For any $1 \leq k \leq K$, let $\mathbb{0}_k$ ($\mathbb{1}_k$, respectively) be the vector of zeros with a unique 1 in position k (vector of ones with a unique 0 in position k , respectively). If $\rho_v = \mathbb{0}_k$, the vertex v corresponds to the subtree $\mathfrak{R}^{-1}(\Delta[v])$, which only appears in class k : v is thus a good discriminator of this class. Otherwise, if $\rho_v = \mathbb{1}_k$, the vertex v appears in all the classes except class k and is still a good discriminator of the class. For any vertex v , δ_v measures the distance between ρ_v and its nearest point of interest $\mathbb{0}_k$ or $\mathbb{1}_k$,

$$\delta_v = \min_{k=1}^K (|\rho_v - \mathbb{0}_k|, |\rho_v - \mathbb{1}_k|).$$

It should be noted that the maximum value of δ_v depends on the number of classes and can be larger than 1. If δ_v is small, then ρ_v is close to a point of interest. Consequently, since v tends to discriminate a class, its weight should be large. In light of this remark, the discriminance weight of a vertex v is defined as $\omega_v = f(1 - \delta_v)$, where $f : (-\infty, 1] \rightarrow [0, 1]$ is increasing with $f(x) = 0$ for $x \leq 0$ and $f(1) = 1$. Figure 8.1 illustrates some usual choices for f . In the sequel, we chose $\omega_v = f^*(1 - \delta_v)$ with the smoothstep function $f^* : x \mapsto 3x^2 - 2x^3$. We borrowed the smoothstep function from computer graphics [104], where it is mostly used to have a smooth transition in a threshold function.

Since leaves appear in all the trees of the training data set, ρ_\bullet is a vector of ones and thus $\delta_\bullet = 1$, which implies $\omega_\bullet = 0$. This is consistent with the result developed in Section 7.2 on the stochastic model. As aforementioned, the discriminance weight is inspired from the theoretical results established in Weight of leaves (p. 99) and the conjecture presented in Remark 7.2. The relevance in practice of this weight function will be investigated in the sequel of this chapter.

Remark 8.2 The discriminance weight is defined from the proportion of data in each class that contain a given subtree, for all the subtrees appearing in the data set. It is thus required to enumerate all these subtrees. This is done, without redundancy, via the DAG reduction Δ of the data set defined and investigated in Section 2.3. As the m trees of the training data set dedicated to learning the discriminance weight are partitioned into K classes, computing one ρ_v vector is of complexity $\mathcal{O}(m)$. Therefore, computing all of them is in $\mathcal{O}(\#\Delta m)$. In addition, computing all values of δ_v is in $\mathcal{O}(\#\Delta K^2)$, as there are $2K$ Euclidean distances to be computed for each vector of length K . All gathered, computing the discriminance weight function has an overall complexity of $\mathcal{O}(\#\Delta(N + K^2))$.

8.2 Real data analysis

This section is dedicated to the application of the methodology developed before to eight real data sets with various characteristics in order to show its strengths and weaknesses.

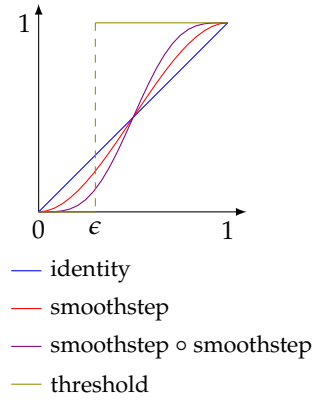


Figure 8.1: The discriminance weight is defined by $\omega_t = f(1 - \delta_t)$ where $f : (-\infty, 1] \rightarrow [0, 1]$ is increasing with $f(0) = 0$ and $f(1) = 1$. This figure presents some usual choices for f .

[104]: Ebert et al. (2003), *Texturing & modeling: a procedural approach*

As mentioned in [Kernel computation on DAGs \(p. 103\)](#), our approach consists in computing the Gram matrices of the subtree kernel via DAG reduction and with a new weight function called the discriminance (see [Section 36](#)). In particular, we aim to compare the usual exponential weight of the literature and the latter in terms of prediction capability. In all the sequel, the Gram matrices are used as inputs to SVM algorithms in order to tackle these classification problems. We emphasize that this approach is not restricted to SVM but can be applied with other prediction algorithms.

Preliminaries

In this subsection, we introduce (i) the protocol that we have followed to investigate several data sets, together with a description of (ii) the classification metrics that we use to assess the quality of our results, (iii) an extension of DAG reduction to take into account discrete labels on vertices of trees, and (iv) the standard method to convert a markup document into a tree. It should be already noted that all the data sets presented in the sequel are composed of trees (that can be ordered or unordered, labeled or not) together with their class.

Protocol For each data set, we have followed the same presentation and procedure. First, a description of the data is made notably via histograms describing the size, outdegree, height and class distribution of trees. Given the dispersion of some of these quantities, we have binned together the values that do not fit inside the interval $[Q_1 - 1.5 \cdot IQR; Q_3 + 1.5 \cdot IQR]$ where $IQR = Q_3 - Q_1$ is the interquartile range. Therefore, the flattened-large bins that appears in some histograms represents those outliers bins. The objective of this part is to show the wide range of data sets considered in this chapter.

Second, we evaluated the performance of the subtree kernel on a classification task via two methods: (i) for exponential weights $t \mapsto \lambda^{\mathcal{H}(t)}$ we randomly split the data in thirds, two for training a SVM, and one for prediction; (ii) for discriminance weight, we also randomly split the data in thirds, one for training the discriminance weight, one for training a SVM, and the last one for prediction. We repeated 50 times this random split for discriminance, and for different values of λ . The classification results are assessed by some metrics defined in the upcoming paragraph, and gathered in boxplots. The first application example, presented in [Prediction of the language of a Wikipedia article from its topology \(p. 108\)](#), is slightly different since (i) we have worked with 50 distinct databases, and (ii) the results have been completed with a deeper analysis of the discriminance weights, in relation to the usual weighting scheme of the literature.

Classification metrics To quantify the quality of a prediction, we use four standard metrics: accuracy, precision, recall and F-score. For a class k , one can have true positives TP_k , false positives FP_k , true negatives TN_k and false negatives FN_k . In a binary classification problem, those

metrics are defined as,

$$\begin{aligned} \text{Accuracy}(k) &= \frac{TP_k + TN_k}{TP_k + FP_k + FN_k + TN_k}, \\ \text{Precision}(k) &= \frac{TP_k}{TP_k + FP_k}, \\ \text{Recall}(k) &= \frac{TP_k}{TP_k + FN_k}, \\ \text{F-score}(k) &= \frac{2 \text{Precision}(k) \times \text{Recall}(k)}{\text{Precision}(k) + \text{Recall}(k)}. \end{aligned}$$

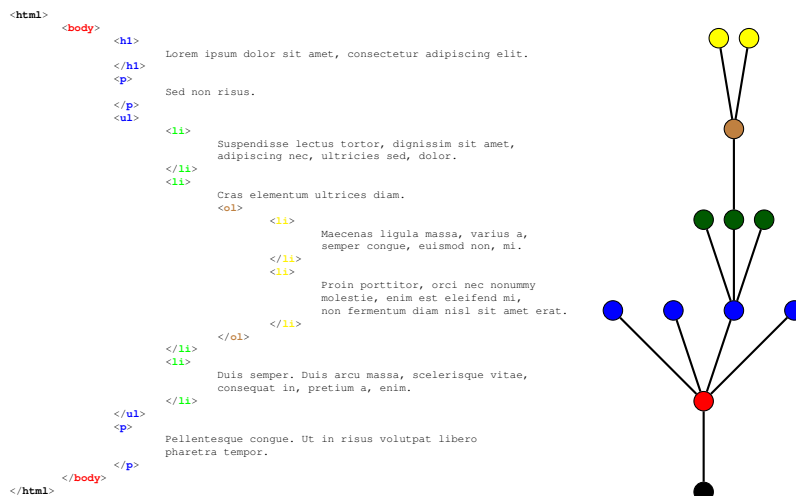
For a problem with $K > 2$ classes, we adopt the macro-average approach, that is,

$$\text{Metric} = \frac{1}{K} \sum_{k=1}^K \text{Metric}(k).$$

We used the implementation available in the `scikit-learn` Python library, namely via the functions `accuracy_score` and `precision_recall_fscore_support`.

DAG reduction with labels In the sequel, some of the presented data sets are composed of labeled trees. The isomorphism between them is assumed to impose the equality of labels, as was done in [36, 102], but for ordered trees only.

From a markup document to a tree Some of the data sets come from markup documents (XML or HTML files). From such a document, one can extract a tree structure, identifying each couple of opening and closing tags as a vertex, whose children are the inner tags. It should be noticed that, during this transcription, semantic data is forgotten: the tree only describes the topology of the document. Figure 8.2 illustrates the conversion from HTML to tree on a small example. Such a tree is ordered but can be considered as unordered. Finally, a tag can also be chosen as a label for the corresponding vertex in the tree.



[102]: Aiolli et al. (2006), 'Fast on-line kernel learning for trees'

[36]: Da San Martino (2009), 'Kernel methods for tree structured data'

Figure 8.2: Underlying ordered tree structure (right) present in a HTML document (left). Each level in the tree is colored in the same way as the corresponding tags in the document. Natural order from top to bottom in the HTML document corresponds to left-to-right order in the tree.

Prediction of the language of a Wikipedia article from its topology

Classification problem and results Wikipedia pages are encoded in HTML and, as aforementioned, can therefore be converted into trees. In this context, we are interested in the following question: does the (ordered or unordered) topology of a Wikipedia article (as an HTML page) contain the information of the language in which it has been written? This can be formulated as a supervised classification problem: given a training data set composed of the tree structures of Wikipedia articles labeled with their language, is a prediction algorithm able to predict the language of new data only from its topology? The interest of this question is discussed in [Remark 8.3](#).

In order to tackle this problem, we have built 50 databases of 480 trees each, converted from Wikipedia articles as follows. Each of the databases is composed of 4 data sets:

- ▶ a data set to predict $\mathcal{X}_{\text{pred}}$ made of 120 trees;
- ▶ a small train data set $\mathcal{X}_{\text{train}}^{\text{small}}$ made of 40 trees;
- ▶ a medium train data set $\mathcal{X}_{\text{train}}^{\text{medium}}$ made of 120 trees;
- ▶ and a large train data set $\mathcal{X}_{\text{train}}^{\text{large}}$ made of 200 trees.

For each data set, and each language, we picked Wikipedia articles at random using the Wikipedia API³⁷, and converted them into unlabeled trees. It should be noted that the probability to have the same article in at least two different languages is extremely low. For each database, we aim at predicting the language of the trees in $\mathcal{X}_{\text{pred}}$ using a SVM algorithm based on the subtree kernel for ordered and unordered trees, and trained with $\mathcal{X}_{\text{train}}^{\text{size}}$ where $\text{size} \in \{\text{small}, \text{medium}, \text{large}\}$. [Figure 8.3](#) provides the description of one typical database. All trees seem to share common characteristics, regardless of their class.

37: <https://www.mediawiki.org/wiki/API:Random>

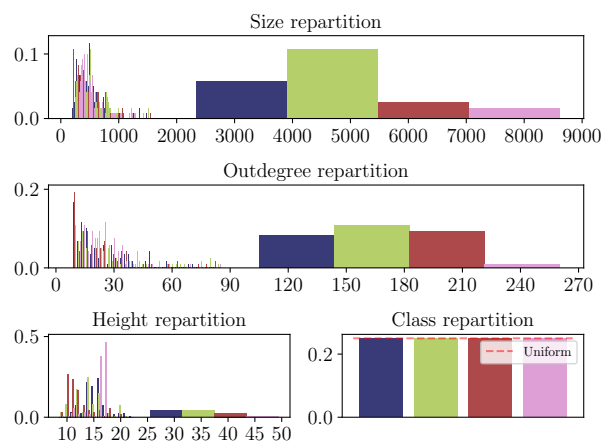


Figure 8.3: Description of a Wikipedia data set (480 trees).

Classification results over the 50 databases are displayed in [Figure 8.4](#). Discriminance weighting achieves highly better results than exponential weighting, with all metrics greater than 90% on average from only 200 training data. This points out that the language information exists in the structure of Wikipedia pages, whether they are considered as ordered or unordered trees, unlike what intuition as well as subtree kernel with exponential weighting suggest. It should be added that the variance of

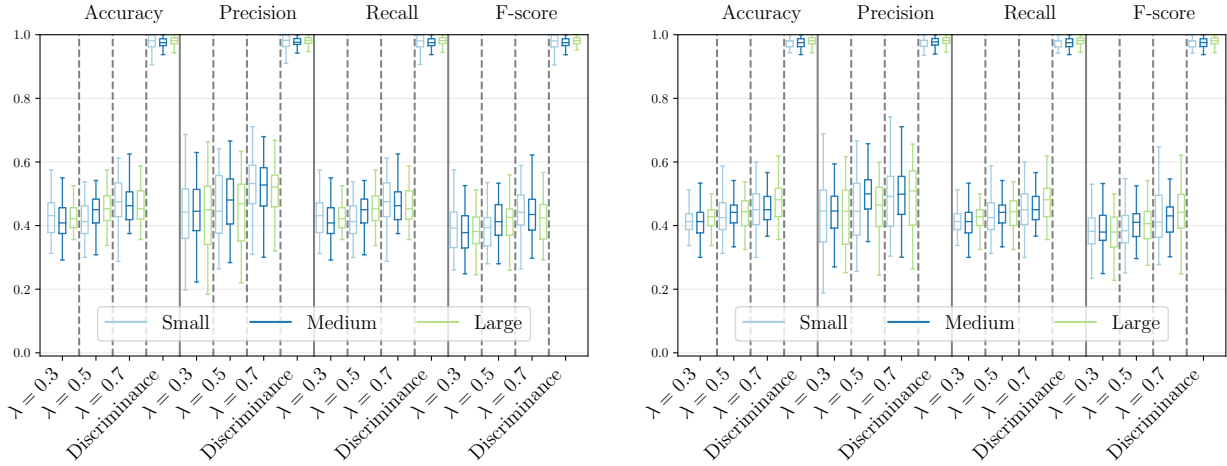


Figure 8.4: Classification results for the 50 Wikipedia databases as ordered (left) and unordered (right) trees. λ values stands for exponential decay weight of the form $t \mapsto \lambda^{\mathcal{H}(t)}$. The colors of the boxplot indicate, for each size $\in \{\text{small, medium, large}\}$, the results obtained for the classification of $\mathcal{X}_{\text{pred}}^{\text{size}}$ from $\mathcal{X}_{\text{train}}^{\text{size}}$.

all metrics seem to decrease with the size of the training data set when using discriminance.

These numerical results show the great interest of the discriminance weight, in particular with respect to an exponential weight decay. Nevertheless, it should be compelling in this context to understand the classification rule learned by the algorithm. Indeed, this could lead to explain how the information of the language is present in the topology of the article.

Comprehensive learning and data visualization When a learning algorithm is efficient for a given prediction problem, it is interesting to understand which features are significant. In the subtree kernel, the features are the subtrees appearing in all the trees of all the classes. Looking at [Equation 7.4](#), the contribution of any subtree t to the subtree kernel with discriminance weighting is the product of two terms: the discriminance weight w_t quantifies the ability of t to discriminate a class, while $\kappa(N_t(T_1), N_t(T_2))$ evaluates the similarity between T_1 and T_2 with respect to t through the kernel κ . As explained in [Section 36](#), if w_t is close to 1, t is an important feature in the prediction problem.

As shown in [Section 2.3](#), DAG reduction provides a tool to compress a data set without loss. We recall that each vertex of the DAG represents a subtree appearing in the data. Consequently, we propose to visualize the important features on the DAG of the data set where the vertices are drawn with a radius that is an increasing function of the discriminance weight (i.e. the bigger the weight, the bigger the node). In addition, each vertex of the DAG can be colored by the class that it helps to discriminate, either positively (the vertex of the DAG corresponds to a subtree that is present almost only in the trees of this class), or negatively. This provides a visualization at a glance of the whole data set that highlights the significant features for the underlying classification problem. We refer the reader to [Figure 8.5](#) for an application to one of our data sets. Thanks to this tool, we have remarked that the subtree corresponding to the

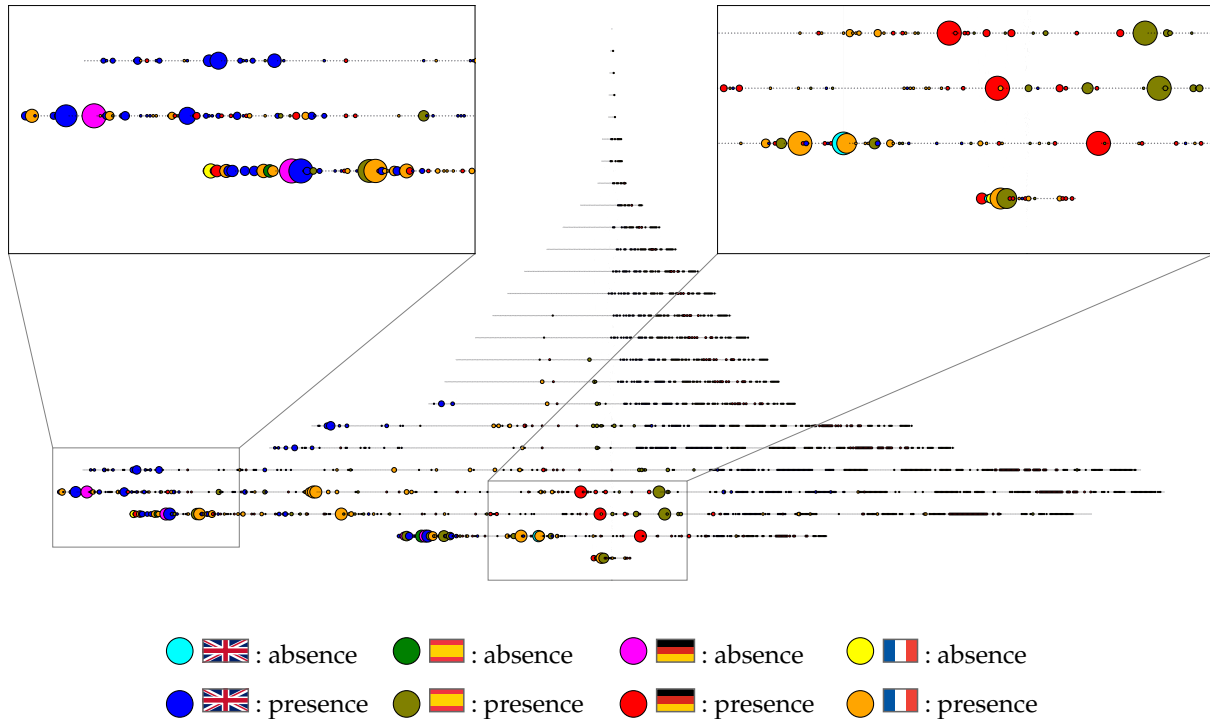


Figure 8.5: Visualisation of one data set $\mathcal{X} = \mathcal{X}_{\text{train}}^{\text{medium}} \cup \mathcal{X}_{\text{pred}}$ of unordered trees among the 30 Wikipedia databases. Each vertex $v \in \mathfrak{R}(\mathcal{X})$ is scaled according to $f^*(1 - \delta_v)$ so that the largest vertices are those that best discriminate the different classes. For each v , we find the class k such that ρ_v has minimal distance to either $\mathbb{0}_k$ or $\mathbb{1}_k$. If it is $\mathbb{0}_k$, we say that v discriminates by its presence, and if it is $\mathbb{1}_k$, v discriminates by its absence. We color v following this distinction according to the legend, where the flags indicate the language.

License at the bottom of any article highly depends on the language, and thus helps to predict the class.

Distribution of discriminance weights To provide a better understanding of our results, we have analyzed in Figure 8.6 the distribution of discriminance weights of one of our large training data sets. It shows that the discriminance weight behaves on average as a shifted exponential. Considering the great performance achieved by the discriminance weight, this illustrates that exponential weighting presented in the literature is indeed a good idea, when setting $w_{\bullet} = 0$ as shown in Weight of leaves (p. 99) or suggested in [37]. However, a closer look at the distribution in Figure 8.6 (left) reveals that important features in the kernel are actually outliers: relevant information is both far from the average behavior and scarce. To a certain extent and regarding these results, discriminance weight is the second order of the exponential weight.

[37]: Vishwanathan et al. (2004), ‘Fast kernels for string and tree matching’

Remark 8.3 The classification problem considered in this subsection may seem unrealistic as ignoring the text information is obviously counterproductive in the prediction of the language of an article. Nevertheless, this application example is of interest for two main reasons. First, this prediction problem is difficult as shown by the bad results obtained from the subtree kernel with exponential weights (see Figure 8.4). As highlighted in Figure 8.5 and Figure 8.6 (left), the subtrees that can discriminate the classes are very infrequent and diverse (in terms of size and structure), so difficult to be identified.

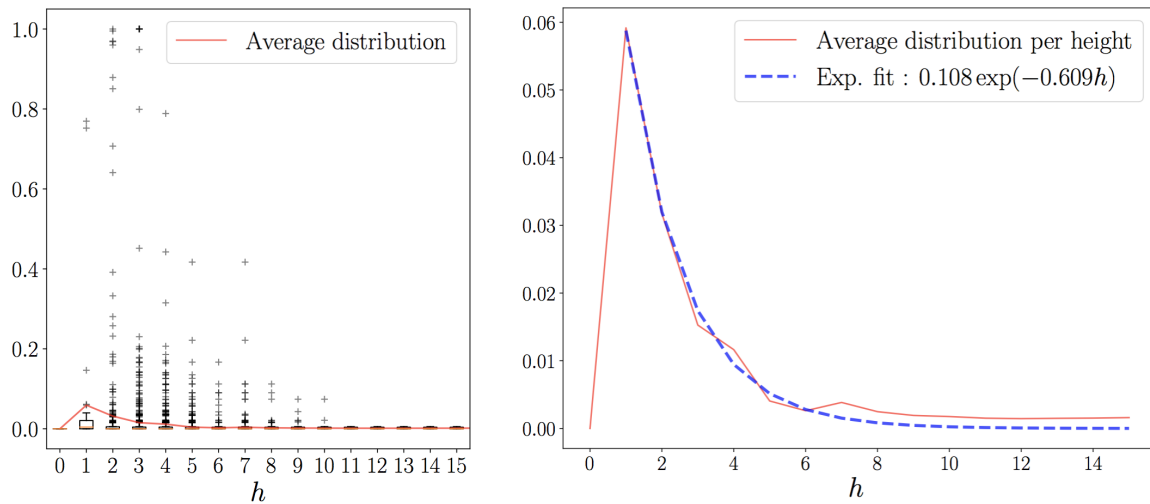


Figure 8.6: Estimation of the distribution of the discriminance weight function $h \mapsto \{w_v : \mathcal{H}(v) = h, v \in \mathfrak{R}(X)\}$ from one large training Wikipedia data set of unordered trees (left) and fit of its average behavior (in red) to an exponential function (in blue). All ordered and unordered data sets show a similar behavior.

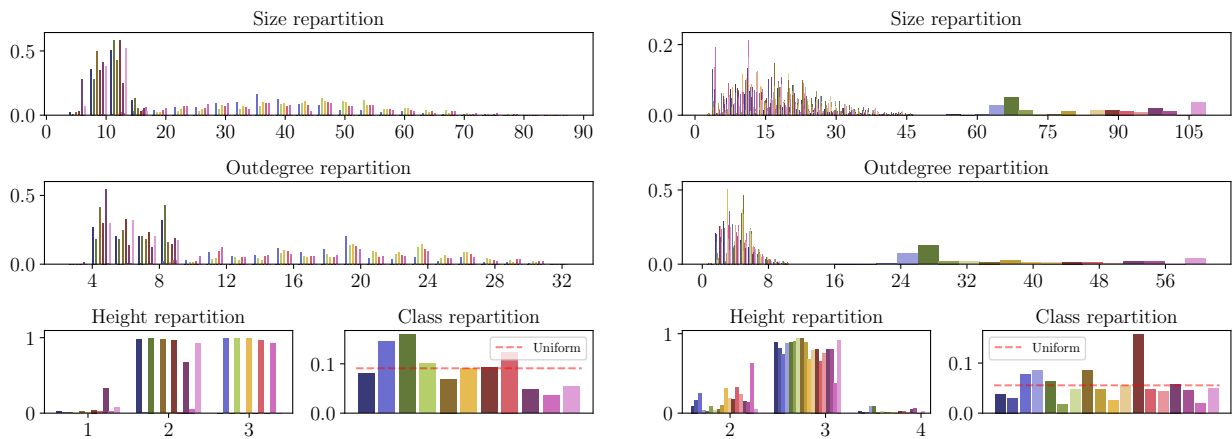


Figure 8.7: Description of INEX 2005 (9,630 trees, left) and INEX 2006 (12,107 trees, right) data sets.

On a different level, as Wikipedia has a very large corpus of pages, it provides a practical tool to test our algorithms and investigate the properties of our approach. Indeed, we can virtually create as many different data sets as we want by randomly picking articles, ensuring that we avoid overfitting.

Markup documents data sets

We present and analyse in this subsection three data sets obtained from markup documents.

INEX 2005 and 2006 These data sets originate from the INEX competition [105]. They are XML documents, that we have been considering as ordered and unordered in our experiments. INEX 2005 is made of 9,630 documents arranged in 11 classes, whereas INEX 2006 has 18 classes for 12,107 documents. For INEX 2005, classes can be split into two groups of trees with similar characteristics, as shown in Figure 8.7 (left). However,

[105]: Denoyer et al. (2007), 'Report on the XML mining track at INEX 2005 and INEX 2006: categorization and clustering of XML documents'

inside each group, all trees are alike. In the case of INEX 2006, no special group seems to emerge from topological characteristics of the data, as pointed out in Figure 8.7 (right).

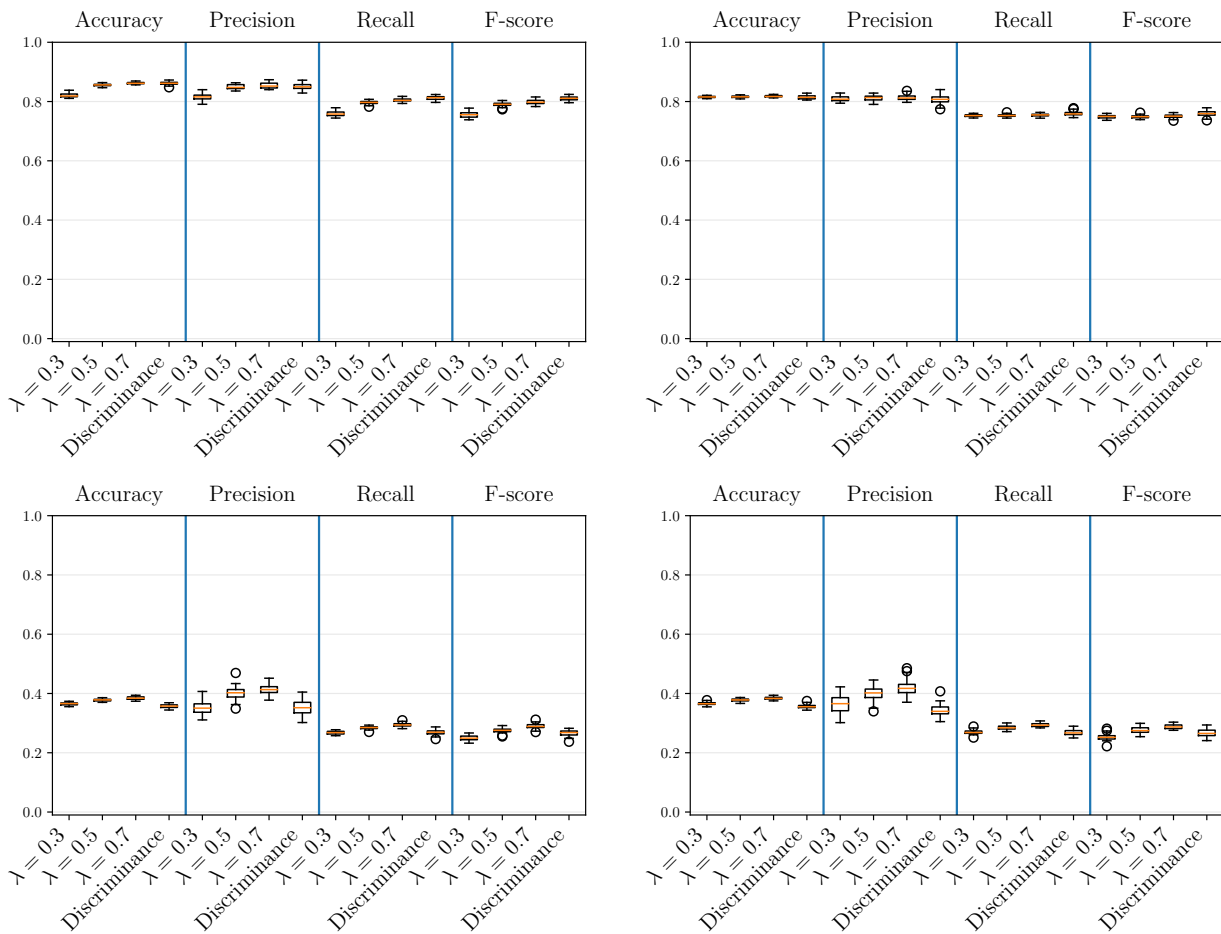


Figure 8.8: Classification results for INEX 2005 (top) and INEX 2006 (bottom) as ordered (left) and unordered (right) trees.

The classification results are depicted in Figure 8.8, for both data sets, and with trees considered successively as ordered and unordered. For INEX 2005, both exponential decay and discriminance achieve similar good performance. However, for INEX 2006, neither of them are able to achieve significant results. Actually, discriminance performs slightly worse than exponential decay. From these results we deduce that subtrees do not seem to form the appropriate substructure to capture the information needed to properly classify the data.

Videogame sellers We manually collected, for two major websites selling videogames³⁸, the URLs of the top 100 best-selling games, and converted them into ordered labeled trees. Although webpages might seem similar to some extent, the trees are actually very different, as highlighted in Figure 8.9. We found that the subtree kernel retrieves this information as, for both exponential decay and discriminance weights, we achieved 100% of correct classifications in all our tests.

38: <https://store.steampowered.com> and <https://www.gog.com>

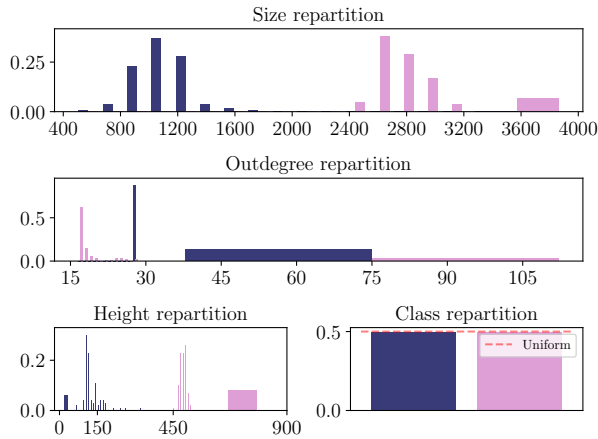


Figure 8.9: Description of the videogame sellers data set (200 trees).

Biological data sets

In this subsection, three data sets from the literature are analyzed, all related to biological topics.

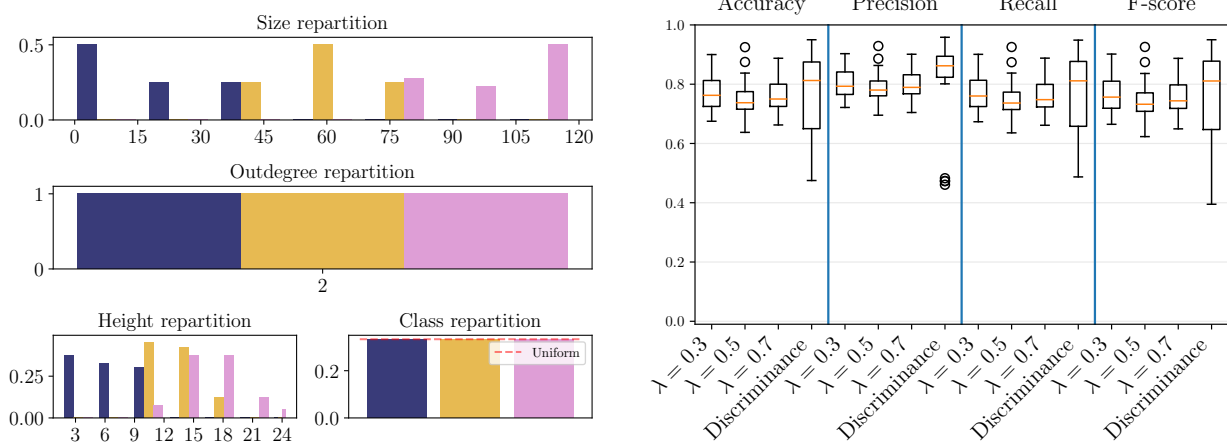


Figure 8.10: Description of the VascuSynth data set (120 trees, left) and classification results (right).

VascuSynth The VascuSynth data set from [106, 107] is composed of 120 unordered trees that represent blood vasculatures with different bifurcations numbers. In a tree, each vertex has a continuous label describing the radius r of the corresponding vessel. We have discretized these continuous labels in three categories: small if $r < 0.02$, medium if $0.02 \leq r < 0.04$ and large if $r \geq 0.04$ (all values are in arbitrary unit). We split up the trees into three classes, based on their bifurcation number. Based on Figure 8.10 (left), we can distinguish between the three classes by looking only at the size of trees. Contrary to the videogame sellers data set that had the same property, the classification does not achieve 100% of good classification, as depicted in Figure 8.10 (right). On average, discriminance performs better than the other weights, despite having a larger variance. This is probably due to the small size of the data set, as the discriminance is learned only with around 13 trees per class.

[106]: Hamarneh et al. (2010), 'VascuSynth: Simulating Vascular Trees for Generating Volumetric Image data with Ground Truth Segmentation and Tree Analysis'
 [107]: Jassi et al. (2011), 'VascuSynth: Vascular Tree Synthesis Software'

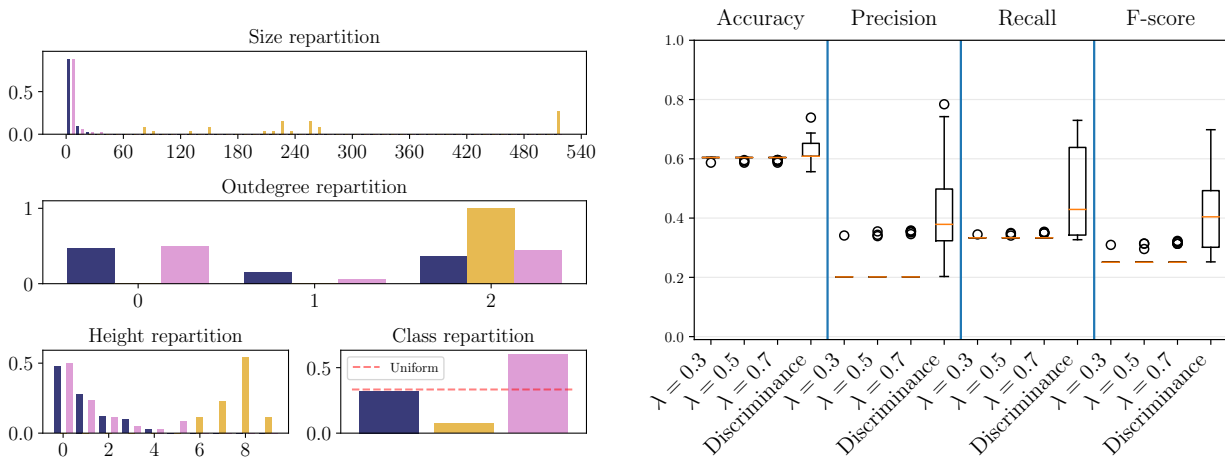


Figure 8.11: Description of the Hicks et al. data set (345 trees, left) and classification results (right).

[108]: Hicks et al. (2019), 'Maps of variability in cell lineage trees'

39: <https://doi.org/10.1101/267450>

[109]: Faure et al. (2015), 'An algorithmic workflow for the automated processing of 3D+ time microscopy images of developing organisms and the reconstruction of their cell lineage'

40: <https://bioemergences.eu/bioemergences/openworkflow-datasets.php>

41: <https://science.rpi.edu/computer-science>

Hicks et al. cell lineage trees Across cellular division, tracking the lineage of a single cell naturally defines a tree. In a recent article, [108] have been investigating the variability inside cell lineage trees of three different species. From the encoding of the data that they have provided as supplementary material³⁹, we have extracted ordered unlabeled trees that are presented in Figure 8.11 (left). The data set contains, for two classes, trees of outdegree 0 (i.e., isolated leaves) that can be considered as noise. With respect to the exponential weight, the value of the kernel between such trees will be identical, whether they belong to the same class or to two different classes. They therefore contribute to reducing the kernel's ability to effectively discriminate between these two classes. On the other hand, the discriminance weight will assign them a zero value, "de-noising", in a way, the data. This observation may explain why discriminance weight achieves better results than exponential weight.

Faure et al. cell lineage trees [109] have developed a method to construct cell lineage trees from microscopy and provided their data online⁴⁰. We extracted 300 unordered and unlabeled trees, divided into three classes. It seems from Figure 8.12 (left) that one class among the three can be distinguished from the two others. Classification results can be found in Figure 8.12 (right): the discriminance weight performs better than the exponential weight, whatever the value of the parameter.

LOGML

The LOGML data set is made of user sessions on an academic website, namely the Rensselaer Polytechnic Institute Computer Science Department website⁴¹, that registered the navigation of users across the website. 23,111 unordered labeled trees are present, divided into two classes. The trees are very alike, as shown in Figure 8.13 (left), and the classification results of Figure 8.13 (right) are very similar to INEX 2005, where all weight functions behave similarly, without any advantage for the discriminance weight in terms of prediction.

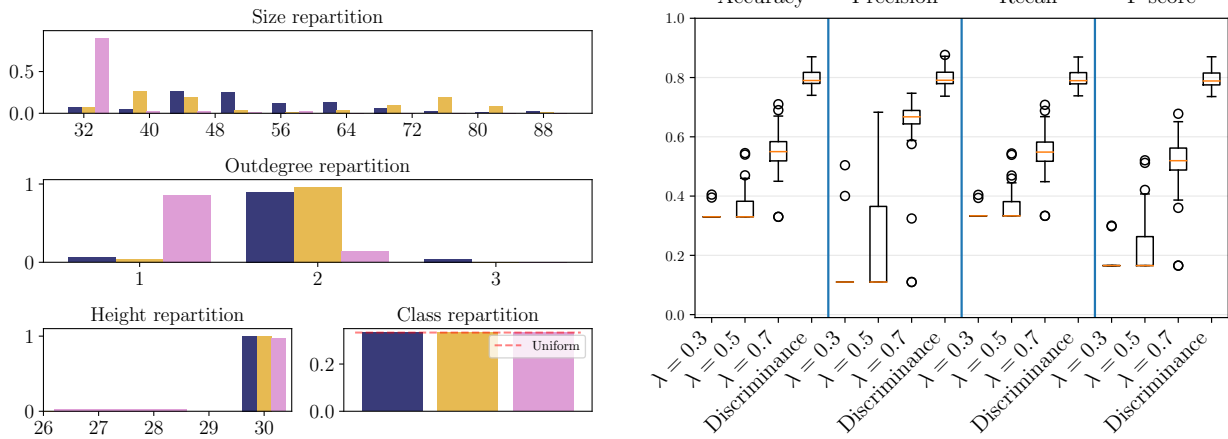


Figure 8.12: Description of the Faure et al. data set (300 trees, left) and classification results (right).

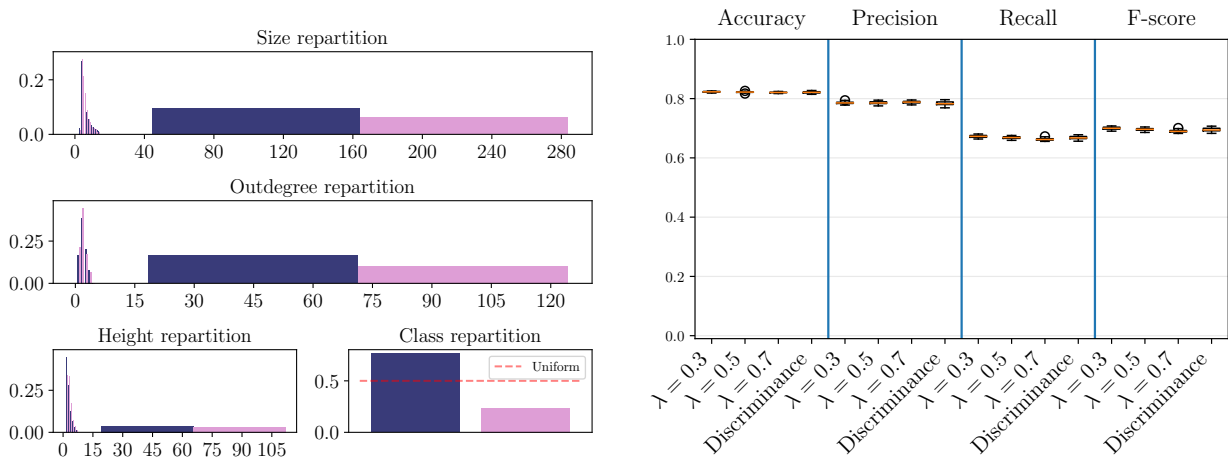


Figure 8.13: Description of the LOGML data set (23,111 trees, left) and classification results (right).

8.3 Interest of the DAG approach

Learning the weight function

In Section 7.2, we have shown on a 2-classes stochastic model that the efficiency of the subtree kernel is improved by imposing that the weight of leaves is null. As explained in Remark 7.2, we conjecture that the weight of any subtree present in two different classes should be 0. The main interest of the DAG approach developed in Kernel computation on DAGs (p. 103) is that it allows to learn the weight function from the data, as developed in Section 36 with the discriminance weight function. Our method has been implemented and tested on eight real data sets with very different characteristics that are summed up in Table 8.1.

As a conclusion of our experiments, we have analyzed the relative improvement in prediction obtained with the discriminance weight against the best exponential weight in order to show both the importance of the weight function and the relevance of the method we developed in this paper. More precisely, for each data set and each classification

Table 8.1: Summary of the 8 data sets.

data set	Wikipedia	Videogames	INEX 2005	INEX 2006	VascuSynth	Hicks et al.	Faure et al.	LOGML
ord. / unord.	both	ord.	both	both	unord.	ord.	unord.	unord.
labeled	✗	✓	✓	✓	✓	✗	✗	✓
# of trees	160 – 320	200	9,630	12,107	120	345	300	23,111
# of classes	4	2	11	18	3	3	3	2

metric, we have calculated

$$RI = \frac{\text{Metric}_{\text{discr}} - \max(\text{Metric}_\lambda)}{\max(\text{Metric}_\lambda)},$$

from the average values of the different metrics. The results are presented in Figure 8.14. We have found that, except in one case, discriminance behaves as good as exponential weight decay and even performs better in most of the data sets. Furthermore, one can observe a kind of trend, where the relative improvement decreases when the number of trees in the training data set is increasing, which proves the great interest of the discriminance to handle small data sets, provided that (i) the problem is difficult enough that the exponential weights are not already high performing, as it is the case in the Videogames sellers data set, and (ii) the data set is not too small, as for VascuSynth. Indeed, as the discriminance is learned independently from the SVM, one must have enough training data to divide them efficiently. Nevertheless, it should be noted that, in the framework of the DAG approach, results from the discriminance weight can be obtained much faster due to the fact that the Gram matrices are estimated from one half of the training data set, while learning the discriminance is very fast as it can be done in one traversal of the DAG (see the complexity presented in Remark 8.2). Finally, we have investigated on a single example some properties of the discriminance, discovering that it can be interpreted as a second-order exponential weight, as well as a method for visualizing the important features in the data.

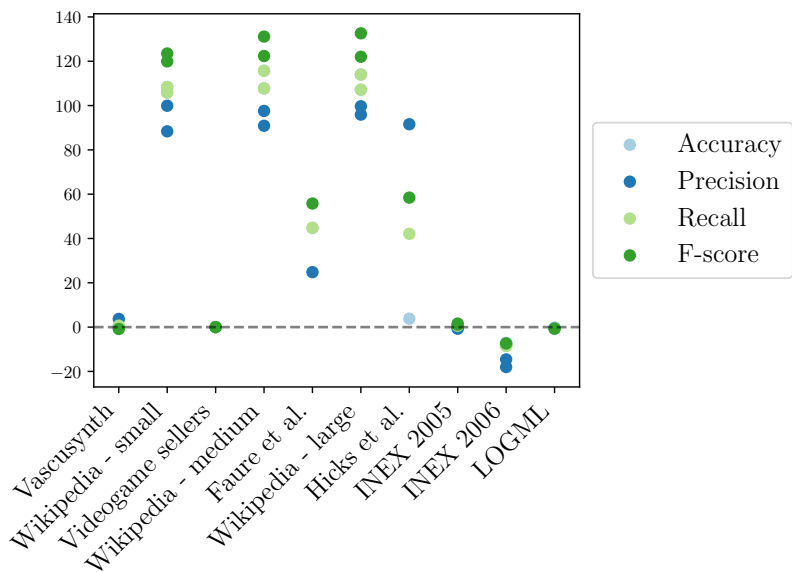
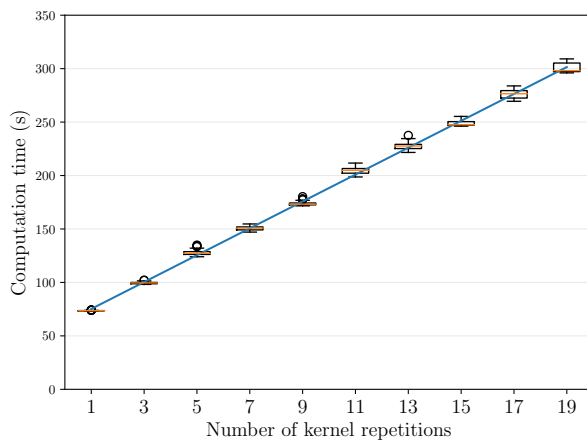


Figure 8.14: Relative improvement RI (in percentage) of the discriminance against the best value of λ for all data sets (sorted by increasing number of trees in the training data set) and all metrics.

Computation time

As shown in Figure 8.12 (right), the exponential decay classification results for the Faure et al. data set are very dependent on the value chosen for the parameter λ . In this case, it can be interesting to tune this parameter and estimate its best value with respect to a prediction score. This requires to compute the Gram matrices from different weight functions. We present in Figure 8.15 the computation time required to compute the Gram matrices from a given number of values of the parameter. As expected from the theoretical results, we observe a linear dependency: the intercept corresponds to the computation time required to compute and annotate the DAG reduction, while the slope is associated with the time required to compute the Gram matrices, which is proportional to the average of $\mathcal{O}(\min(\#T_i, \#T_j))$ (see Remark 8.1). This can be compared to the complexity of the algorithm developed in [37], which is the average of $\mathcal{O}(\#T_i + \#T_j)$. Consequently, the corresponding computation times should be proportional to at least twice the slope that we observe with the DAG approach. This shows another interest of our method that is not related to the discriminance weight function. It should be faster to compute several repetitions of the subtree kernel from the DAG approach than from the previous algorithm [37] provided that the number of repetitions is large enough.



[37]: Vishwanathan et al. (2004), 'Fast kernels for string and tree matching'

Figure 8.15: Computation time required to compute several repetitions of the kernel on the Faure et al. data set. All those calculations have been repeated 50 times for each number of repetitions. The intercept corresponds to the DAG compression of the data set, which is independent on the number of repetitions. The blue curve is a linear fitting of all the measurement points.

I took a walk in the woods and came out taller than the trees.

Henry David Thoreau

In this chapter, we review what has been presented in this thesis, and provide a number of perspectives for further research.

In [Section 9.1](#), we consider isomorphisms of labeled trees, as treated in [The Tree Ciphering Isomorphism Problem](#). In particular, (i) we propose ways to further refine the algorithm proposed in [Chapter 4](#); (ii) we consider whether the algorithm is suitable to address the graph isomorphism problem; and finally (iii) we introduce an optimization problem where the goal is to find a tree ciphering that optimizes a given criterion, rather than simply the first one found.

In the context of the search for frequent patterns, [Section 9.2](#) establishes two perspectives based on the enumeration of DAGs compressing irredundant forests as developed in [Enumeration Trees: from Trees to Forests](#): the extension to labeled DAGs, and the enumeration of (general) DAGs.

In [The Subtree Kernel Revisited](#), we focused on convolution kernels and in particular on the subtree kernel. In [Section 9.3](#), we introduce a new kernel based on the enumeration of forests of subtrees presented in [Chapter 6](#); followed by a discussion of the extent to which the lessons learned from the subtree kernel would be adaptable to other kernels.

9.1 Tree isomorphisms

In [Chapter 3](#) and [Chapter 4](#), we focused on isomorphisms of labeled trees – where two trees are isomorphic *up to a cipher* if and only if they are identical up to label rewriting.

In the same way that DAG compression is derived – in [Section 2.3](#) – from the usual tree isomorphism, we were able to define, from this new definition, a DAG compression of labeled trees. This compression is lossless, and we have provided [Algorithm 3](#) which allows to build that DAG from a tree.

The algorithm to build the DAG compression of an unlabeled tree is based on AHU algorithm [\[22\]](#), that determines whether two trees are isomorphic – which is linear. Our new labeled compression algorithm is also based on an algorithm that determines whether two labeled trees are isomorphic up to a cipher. However, the latter is not linear. Indeed, determining whether two trees are isomorphic up to a cipher is as hard as determining whether two graphs are isomorphic, a problem whose complexity is assumed to be intermediate between P and NP [\[67\]](#).

- 9.1 Tree isomorphisms 119
 - Refining the algorithm . . . 120
 - Application to graph isomorphism 121
 - Optimization on the cipher space 122
- 9.2 Search for frequent patterns 124
 - Extension to labeled FDAGs 124
 - Toward the enumeration of DAGs 126
- 9.3 Classification of trees 127
 - The forest kernel 127
 - Explicit enumeration with other kernels 129

[\[22\]](#): Aho et al. (1974), ‘The design and analysis of computer algorithms’

[\[67\]](#): Schönig (1987), ‘Graph isomorphism is in the low hierarchy’

We have devoted [Chapter 4](#) to the treatment of this problem, and proposed an algorithm that constructs an isomorphism – when it exists – between two labeled trees. Our algorithm operates in two phases: (i) the first drastically breaks the cardinality of the search space, by determining a number of mandatory mappings of the isomorphism; and (ii) the second is a backtracking phase that completes the construction.

Refining the algorithm

We saw in [Section 4.4](#) that the first phase, preprocessing, drastically reduces the search space, which allows the subsequent backtracking to be efficient. On the other hand, in some configurations, the latter can be considerably slow to converge, so there is still room for improvement. Two possibilities are presented here.

Bags on labels If two trees do not have the same number of nodes, nor the same height or degree, it is immediate to determine that they are not isomorphic. This criterion offers a simple way to decide without having to deploy the full artillery of the AHU algorithm.

In the same way, one can detect that two trees are not isomorphic up to a cipher simply if they do not have the same number of labels. To be more precise, and inspired by coloring refinement algorithms such as Weisfeiler-Leman [[68](#), [110](#)], we can compare the histograms of the number of occurrences of the labels (sorted from the most frequent to the least frequent) and if they differ, decide on non-isomorphism.

If the histograms are identical, not only is there a chance that the trees are isomorphic, but labels with identical numbers of occurrences are candidates to be mapped together. The notion of bags, used for the nodes in our algorithm, naturally arises to treat the labels as well. A deduction rule similar to [Deduction Rule 4.1](#) can be used, which maps labels together if they are alone in their bag.

This refinement is likely to improve the performance of the algorithm when the number of labels is smaller than the number of nodes in the trees. Unfortunately, the most problematic cases at the moment are not in this category. To apply our algorithm to real data, nevertheless, it will be desirable to implement this improvement.

Backtracking strategy So far, the strategy used in backtracking, in light of [Theorem 4.3](#), is to prioritize bags and collections according to their number of elements, but without consideration for the elements themselves.

Consider what could be gained by taking into account the elements of the bags rather than just their cardinality.

A reward could be associated with each possible mapping, for example in terms of the reduction of the search space that it produces. This reward is difficult to estimate without actually doing the mapping and observing all the deductions involved. On the other hand, we can set a “horizon”, a limit level of recursion in the mappings and deductions which allows us to estimate this reward.

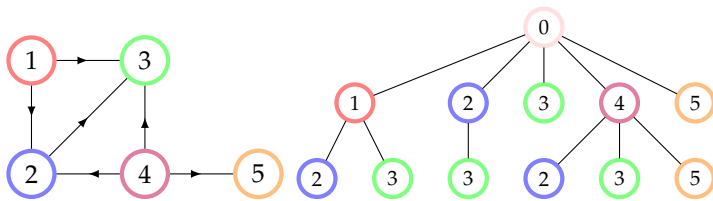
[68]: Weisfeiler et al. (1968), ‘The reduction of a graph to canonical form and the algebra which appears therein’
 [110]: Grohe et al. (2017), ‘Color refinement and its applications’

For example, a horizon of 1 would look at whether the mapping of two nodes also causes the mapping of their parents; and how the bags and collections containing their children are split (without going into the recursion of `SPLITCHILDREN`). Since the depth of nodes has an influence on the amount of maximal successive parent mappings possible, and on the maximal level of recursion of the `SPLITCHILDREN` procedure, this could be an indicator of great interest.

The backtracking tree exploration problem might also be approached with stochastic methods like Monte Carlo Tree Search – a method widely used to explore decision trees (e.g. in the context of game theory) whose large number of states makes exhaustive exploration impossible. The tree is explored partially, via random sampling, in order to explore only the most promising options. On that topic, see [111].

Application to graph isomorphism

The problem of tree ciphering isomorphism is graph isomorphism complete, by virtue of [Theorem 4.1](#). [Figure 4.1](#), reproduced below, illustrates how to transform a graph into a labeled tree – note that this reduction is linear in the size of the graph. Since we have constructed an algorithm to determine tree isomorphism up to a cipher, one may wonder how well it would solve the graph isomorphism problem. Recall that most algorithms addressing the problem [25] do not directly construct isomorphisms between graphs, but rather check if they belong to the same equivalence class (by constructing a representative of each class).



[111]: Browne et al. (2012), ‘A survey of Monte Carlo tree search methods’

[25]: McKay et al. (2014), ‘Practical graph isomorphism, II’

Figure 4.1: A graph (left) and its reduction as a labeled tree (right). Colors have been added for better readability.

Given a graph $G = (V, E)$, we can notice that the tree T_G constructed by the reduction has $\#V + \#E + 1$ nodes, is of height 2, and of degree $\#V$. All leaves have the same equivalence class (for \simeq), and the equivalence class of intermediate nodes is directly related to their degree: for any nodes u, v with $\mathcal{H}(u) = \mathcal{H}(v) = 1$, we have $[u] = [v] \iff \#C(u) = \#C(v)$. Therefore, we can only expect the preprocessing to

- ▶ map the vertices of the graph that are the only ones to have a given degree;
- ▶ partition the other vertices of the graph according to their degree.

Actually, this classification of the graph vertices by degree corresponds exactly to the state of the graph after the first color refinement pass of the Weisfeiler-Leman algorithm [68, 110].

Since we then proceed with a backtracking step, there is little hope that our algorithm is competitive against the state of the art for graph isomorphism. Recall, however, that competing algorithms do not explicitly construct isomorphism, and so our algorithm paves the way in this direction. Moreover, given the very particular shape of the trees obtained by the reduction, one can imagine rethinking the algorithm while taking into

[68]: Weisfeiler et al. (1968), ‘The reduction of a graph to canonical form and the algebra which appears therein’

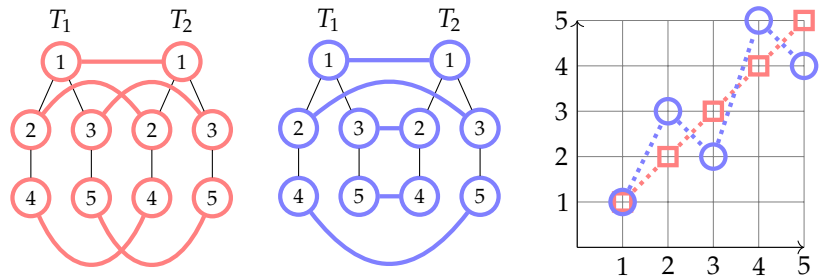
[110]: Grohe et al. (2017), ‘Color refinement and its applications’

account these specificities (the algorithm being currently much more generic), or even following the same kind of strategy but operating directly on the graph.

Optimization on the cipher space

Consider two isomorphic labeled trees T_1 and T_2 , and assume that $\#\mathcal{A}(T_i) = \#T_i$ – in other words, all labels are different. In this case, any tree isomorphism between T_1 and T_2 is also a tree ciphering, i.e., $\text{Isom}(T_1, T_2) = \text{Cipher}(T_1, T_2)$. So far, we have been interested in tree isomorphisms that induce a bijection on the label space. One may want to impose additional conditions on the ciphers; especially in our example where all isomorphisms are valid. For instance, one might want to favor the red isomorphism in the example of Figure 9.1 for its underlying linearity, as opposed to the “sawtooth” behaviour of the blue isomorphism.

Figure 9.1: Two topologically isomorphic unordered trees T_1 and T_2 and the only two possible tree isomorphisms between them (left and center); as well as the two induced bijections on labels (right), where the dotted lines interpolates the points.



We developed in Chapter 4 an algorithm to build *one* tree ciphering; we are interested here in finding the *best* one (even in the case where $\#\mathcal{A}(T_i) \neq \#T_i$). There are many ways to make sense of “best” so, in order to fix things, we consider here just one particular problem among others.

Let T_1 and T_2 be two labeled trees. Let us denote by \mathbb{G} the space of functions $\mathcal{A}(T_1) \rightarrow \mathcal{A}(T_2)$. Suppose there exist $\phi_0 \in \text{Cipher}(T_1, T_2)$ and $f_0 \in \mathbb{G}$ so that $\forall u \in T_1, \overline{\phi_0(u)} = f_0(\overline{u})$. Moreover, let \mathbb{F} be a subspace of \mathbb{G} ; suppose that $f_0 \in \mathbb{F}$ – in the example of Figure 9.1, where $\mathcal{A}(T_i) \subseteq \mathbb{R}$, \mathbb{G} is the space of real functions. We could take $\mathbb{F} = \{x \mapsto ax + b : a, b \in \mathbb{R}\}$ – i.e. linear functions. We have $f_0 = \text{id} \in \mathbb{F}$.

If we know ϕ_0 , then finding the function f_0 is akin to a standard regression problem. f_{ϕ_0} provides a list of couples $(\overline{u}, \overline{\phi_0(u)})$ on which the regression can be performed; and since $f_0 \in \mathbb{F}$, we are sure to retrieve f_0 .

Now suppose that we ignore ϕ_0 . For each $\phi \in \text{Cipher}(T_1, T_2)$, we have an associated candidate f_ϕ . Suppose that \mathbb{G} is an inner product space, and denote by $\|\cdot\|$ the (discrete) norm induced by the inner product. Let us denote by $\text{Proj}_{\mathbb{F}}(f_\phi)$ the projection of f_ϕ on \mathbb{F} ⁴². Then, to f_ϕ , we can associate $\|f_\phi - \text{Proj}_{\mathbb{F}}(f_\phi)\|$. The goal is then to find ϕ that minimizes this quantity – that is, find f_ϕ that is the most regular with respect to the functions of \mathbb{F} .

For the sake of example, suppose $\mathbb{F} = \{x \mapsto ax + b : a, b \in \mathbb{R}\}$. Let $\hat{f} = \text{Proj}_{\mathbb{F}}(f_\phi)$. To calculate \hat{f} , we use the pairs $(\overline{u}, \overline{\phi(u)})$ provided by f_ϕ . By

42: Note that determining $\text{Proj}_{\mathbb{F}}(f_\phi)$ is in itself an optimization problem. See below.

least squares, finding \hat{f} is equivalent to minimizing $\sum_{u \in T_1} (a\bar{u} + b - \overline{\phi(u)})^2$ with respect to $a, b \in \mathbb{R}^{43}$. In the end, we obtain the following optimization problem:

$$\min_{\phi \in \text{Cipher}(T_1, T_2)} \min_{a, b \in \mathbb{R}} \sum_{u \in T_1} (a\bar{u} + b - \overline{\phi(u)})^2.$$

More generally, if we suppose that $\mathcal{A}(T_2)$ is embedded in a metric space whose metric is denoted by d , the problem is equivalently expressed as

$$\min_{\phi \in \text{Cipher}(T_1, T_2)} \min_{f \in \mathbb{F}} \sum_{u \in T_1} d(f(\bar{u}), \overline{\phi(u)}). \tag{9.1}$$

Note that since they are of the same nature, the min operators can be freely exchanged. Given a fixed ϕ , finding an optimal $f \in \mathbb{F}$ is as easy as solving a regression problem on \mathbb{F} . On the other hand, with fixed f , finding ϕ that minimizes the objective function is as difficult as exploring the space $\text{Cipher}(T_1, T_2)$. This prevents us from solving the problem by component-wise optimization (i.e., fix ϕ , minimize according to f , fix f , minimize according to ϕ , etc), since we would need to investigate this challenging space numerous times.

On the other hand, one can imagine a greedy strategy allowing to build a pair (ϕ, f) whose least squares error will hopefully be not too far from the actual minimum. Let us only perform preprocessing between T_1 and T_2 to get a partial mapping of nodes and a preliminary list of label pairs. From this partial list, we can estimate a first candidate $f \in \mathbb{F}$. The collections resulting from the preprocessing are merged to create bags, and then we can look, among the possible pairings of nodes in all bags, at the pair closest to f . We map these two nodes, and make deductions as for the normal algorithm. We then re-estimate f on the basis of the mapping obtained, and we repeat until we run out of nodes.

Assuming that this greedy algorithm allows to build a fairly good solution – which would remain to be investigated, we can go further and imagine an approximate DAG compression. Let’s go back to the example in Figure 3.6 (only the tree is reproduced here). If the coordinates of the points are noisy, it will no longer be possible to find the symmetries of the figure using an exact cipher – any association of the labels becoming acceptable.

On the other hand, considering the above optimization problem and choosing \mathbb{F} the space of rigid transformations of the plane, one will notice that the left red subtree is *approximately* transformed into the right red subtree, that the top left blue subtree is *approximately* transformed into the bottom left blue subtree, and so on.

We can then recursively construct a DAG representing the symmetries of the tree, at the cost of an approximation error. It is notable that the DAG is constructed in the same way as in Section 3.3, by placing the transformations on the arcs, each vertex having a label representing its class (no longer of equivalence, but of approximation).

43: Summing over $u \in T_1$, instead of $\alpha \in \mathcal{A}(T_1)$, amounts to weighting each element $\mathcal{A}(T_1)$ by its number of occurrences in T_1 . If $\#\mathcal{A}(T_1) = \#T_1$, the two sums are equivalent.

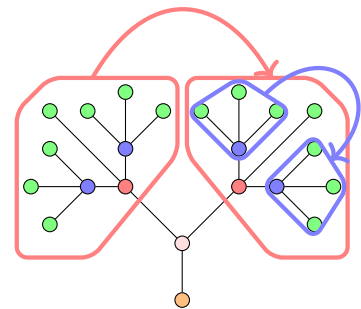


Figure 3.6: A labeled tree, where the labels are the (x, y) coordinates of each node. Some symmetries of the tree are highlighted.

We can then reconstruct a tree from this DAG using [Algorithm 5](#). Because of the successive and recursive approximations, one can expect the labels of this tree to be further from the original ones the deeper the nodes are in the tree; but if the greedy algorithm is effective, one can hope that the global error will not be too high either.

9.2 Search for frequent patterns

In [Chapter 5](#) and [Chapter 6](#), we developed a reverse search scheme allowing, first of all, to enumerate irredundant forests in their compressed DAG form. We have also shown that this enumeration can be adapted to address several related problems: (i) the enumeration of forests in their classical sense, (ii) the enumeration of “subforests” of a forest, and (iii) the frequent “subforest” problem. These results generalize previous results from the literature allowing to enumerate trees and to deal with the problem of frequent pattern mining on trees.

Extension to labeled FDAGs

In our enumeration of irredundant forests, we considered only unlabeled trees. One way to add labels could follow the strategy developed for classical forests, in [Extension to forests with repetitions \(p. 81\)](#): we build an unlabeled FDAG, stop the enumeration, and start a second enumeration procedure to add labels.

This new enumeration will obviously depend on the underlying isomorphism (imposing label equality or up to a cipher); and it should also be noted that it will require modifying the FDAG structure. Indeed, two subtrees with the same topological structure have no particular reason to be isomorphic when labels are added, and thus could generate two vertices in a labeled DAG instead of one. For example, with equality of labels, a leaf with label A is not compressed together with a leaf with label B .

Note that even if we modify the structure, we create topologically equivalent vertices and therefore we can not accidentally create an FDAG obtained by another path in the enumeration tree: the uniqueness of the enumeration is preserved.

The number of distinct labels in a forest is limited by the total number of nodes in the forest. Given a FDAG D , and a vertex $v \in D$, we denote by $s(v)$ the number of nodes in $\mathfrak{R}^{-1}(D[v])$; i.e., with $T_v = \mathfrak{R}^{-1}(D[v])$, $s(v) = \#T_v$. We have

$$s(v) = \begin{cases} 1 & \text{if } v \in \mathcal{L}(D), \\ 1 + \sum_{u \in \mathcal{C}(v)} n(v, u) s(u) & \text{otherwise;} \end{cases}$$

where $n(v, u)$ is the multiplicity of arc $v \rightarrow u$ in D . Therefore, the maximal number of distinct labels we can choose is given by $s(D) = \sum_{r \in \mathcal{R}(D)} s(r)$. Let $\mathcal{A} = \{1, \dots, s(D)\}$ be the alphabet from which we will draw labels⁴⁴.

44: One may want to fix the alphabet before observing the FDAG D . It does not affect the argument afterwards, but it will obviously affect the combinatorics and the number of labeled DAGs that can be constructed from D .

If we were to enumerate labeled FDAGs according to the definition of isomorphism imposing equality of labels, we would end up enumerating the same FDAG several times, up to a rewriting of the labels – for example, one leaf with label A and another with label B. From the point of view of parsimonious enumeration, it would therefore be more judicious⁴⁵ to adopt another approach, i.e. isomorphism up to a cipher.

This question is closely related to the number of non-isomorphic ways one can label a tree of a given size. Then, we have to retranscribe these labelings into a DAG, and in particular find the ciphers that go on the arcs. Moreover, the DAG compression with labels is not unique in general and we will have to find a systematic way to choose the compression (a kind of canonical DAG compression). These arguments are developed below.

Let $a(n)$ be the number of non-isomorphic (from the topology point of view) trees of size n , and $b(n)$ the number of non-isomorphic (up to a cipher) labeled trees of size n . Table 9.1 gives the first values of $a(n)$ and $b(n)$. The values for $n = 1, 2, 3$ come from Figure 9.2, $n = 4$ was obtained by manual counting (not reproduced here) and for $n \geq 5$ the work remains to be done – but with little hope of easily obtaining further values manually. The On-Line Encyclopedia of Integer Sequences references 20 sequences starting with 1, 2, 9, 48, none of which seem trivially related to the problem, so this is a genuine open question.

Remark 9.1 Note that the difficulty of the problem is not so much in the total number of possibilities as a function of n , but rather in the way this number grows by going from n to $n + 1$. Indeed, the reverse search method is only effective if the successors of an element in the enumeration tree can be calculated in a reasonable time. More than $b(n)$, it is therefore $b(n + 1) - b(n)$ that is the quantity of interest; and we can see that it seems to grow faster than $a(n + 1) - a(n)$.

Let us now relate non-isomorphic labeled trees to their labeled DAG. We can already identify several difficulties to be taken into account in the enumeration.

First, there is no uniqueness of the labeled DAG compressing a given tree, since there might be several possible choices of representatives for each DAG vertex, as shown in Figure 9.3. One way to deal with this would be to find a “canonical” compression that provides a systematic way to choose among the possible representatives of a vertex.

Second, you need to make sure that the ciphers on the arcs compose correctly with each other. This typically requires choosing the labels and ciphers on lower height vertices before the higher ones. A cipher placed on an arc leading to a subtree with k different colors has a domain of cardinality at most k – which represents, at most, $\sum_{i=1}^k \binom{k}{i}$ different ciphers, which must then be reported on the arcs, with the resulting combinatorics.

Note that the enumeration of labeled trees (as DAGs or not) contains the enumeration of labeled trees of height 2, and thus a fortiori, the entirety of (unlabeled) graphs by the reduction of Theorem 4.1 – whose combinatorics is illustrated in Table 9.2; see also [112]. This inclusion

45: With the notable exception where an alphabet would be imposed for the enumeration, and where the labels would not be interchangeable.

n	1	2	3	4	5	6
$a(n)$	1	1	2	4	9	20
$b(n)$	1	2	9	48	?	?

Table 9.1: First values of the sequences $a(n)$ and $b(n)$ counting the number of non-isomorphic trees of size n , respectively for tree isomorphism and tree isomorphism up to a cipher. Recall that $a(n)$ is sequence A000081 in *The On-Line Encyclopedia of Integer Sequences* (2022), <https://oeis.org/A000081>.

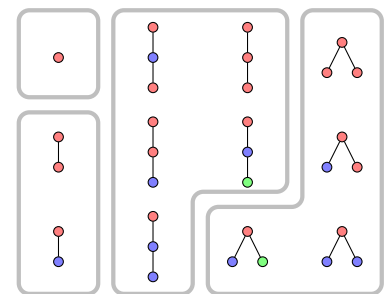


Figure 9.2: All possible non-isomorphic (up to a cipher) labeled trees of size at most 3. The color of nodes is their label.

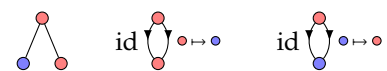


Figure 9.3: Example of non-uniqueness of labeled DAGs compressing a given labeled tree.

[112]: Harary (1955), ‘The number of linear, directed, rooted, and connected graphs’

of unlabeled graphs in the set of labeled trees highlights the gain in difficulty of the problem.

Table 9.2: Comparison between the number $g(n)$ of unlabeled connected graphs with n vertices with the number $a(n)$ of trees with n vertices for some values of n . $g(n)$ is sequence A001349 and $a(n)$ is sequence A000081 in *The On-Line Encyclopedia of Integer Sequences* (2022), <https://oeis.org/A001349> and <https://oeis.org/A000081>.

n	$a(n)$	$g(n)$
1	1	1
5	9	21
10	719	11,716,571
15	87,811	31,397,381,142,761,241,960

As for the enumeration of FDAGs, one can imagine a frequent pattern mining application to the enumeration of labeled FDAGs, but in view of the additional difficulty, a very efficient algorithm will be required to hopefully address the issue on a practical level.

Toward the enumeration of DAGs

Our enumeration of FDAGs opens the way to the enumeration of DAGs in general, which to the best of our knowledge has never been addressed in the literature. In detail, we recall that the key to our method relies on the existence of a canonical ordering on the vertices of a FDAG, allowing then to enumerate only such canonical FDAGs, as developed in *Canonical FDAGs* (p. 67). Given any DAG D , we introduced the following constraints on an ordering ψ :

$$\forall (u, v) \in D^2, \mathcal{H}(u) > \mathcal{H}(v) \implies \psi(u) > \psi(v); \quad (\text{Equation 5.5})$$

$$\forall (u, v) \in D^2, (\mathcal{H}(u) = \mathcal{H}(v)) \wedge (C_\psi(u) >_{\text{lex}} C_\psi(v)) \implies \psi(u) > \psi(v). \quad (\text{Equation 5.6})$$

We proved in *Theorem 5.4* that these constraints are sufficient to ensure the uniqueness of an ordering if and only if D is a FDAG; indeed, it is not possible to find in such a DAG two vertices u and v such that $C_\psi(u) = C_\psi(v)$ and $u \neq v$ – which is the only ambiguous case not covered by the two previous constraints.

If one were to attempt to define a canonical ordering for any generic DAG, based on the constraints introduced here, then one could introduce the following additional constraint.

$$\forall (u, v) \in D^2, (\mathcal{H}(u) = \mathcal{H}(v)) \wedge (C_\psi(u) = C_\psi(v)) \wedge (\mathcal{P}_\psi(u) >_{\text{lex}} \mathcal{P}_\psi(v)) \implies \psi(u) > \psi(v); \quad (9.2)$$

where $\mathcal{P}_\psi(u)$ is the sequence $(\psi(w) : w \in \mathcal{P}(u))$ sorted in decreasing order with respect to the lexicographical order – similarly to C_ψ .

This new constraint would then allow to discriminate more vertices, but would not allow to distinguish between vertices u, v such that $\mathcal{P}_\psi(u) = \mathcal{P}_\psi(v)$. One could also discriminate according to other computable properties on the vertices, but we have exhausted the most obvious ones. In fact, one can hardly expect any better by virtue of the following argument.

Let us assume that we can exhibit a canonical ordering on (general) DAGs, computable in polynomial time. Therefore, given two DAGs D_1 and D_2 , canonically ordered, we can establish, in linear time, whether $D_1 \simeq D_2$ (i.e. D_1 and D_2 are isomorphic graphs) simply by checking that each vertex in order has the same ordered list of children and parents. Yet, DAG isomorphism is graph isomorphism complete [66].

Therefore, unless graph isomorphism is polynomial – for which we have no proof [67], we can not hope to find a canonical ordering on DAGs that is computable in polynomial time, which implies that the discriminating properties must be more sophisticated than those we considered in our approach. We recall that all properties we used to discriminate vertices (such as height) are computable at worst in linear time.

In order to enumerate DAGs by imposing a canonical ordering, one would have to develop a completely different method from ours to find such an ordering. Actually, since finding a canonical ordering is equivalent to defining a representative of the equivalence class of a given DAG (for the graph isomorphism equivalence relation), it would be appropriate to consider methods from the literature that already have this kind of approach [25].

[66]: Zemlyachenko et al. (1985), ‘Graph isomorphism problem’

[67]: Schönig (1987), ‘Graph isomorphism is in the low hierarchy’

[25]: McKay et al. (2014), ‘Practical graph isomorphism, II’

9.3 Classification of trees

In [Chapter 7](#), we focused on the issue of statistical analysis of trees, and in particular on convolution kernels, which are widely used in the literature. We were especially interested in the subtree kernel, and we showed, in a theoretical framework, that it is always preferable that the leaves of the trees have a weight of zero in the computation of the kernel.

We then developed in [Chapter 8](#) a new framework to compute the subtree kernel, based on DAG compression of trees, which allowed us to introduce the discriminance weight, learned from the data, that sets the weight of leaves to zero. Generally speaking, it assigns a high weight to subtrees allowing to better discriminate the classes (in a supervised classification context) and a zero weight to the others. On several datasets, we showed that this new weight significantly improves the performance of the kernel. We emphasize here that the calculation of this discriminance weight was made possible by the exhaustive enumeration of the subtrees.

The forest kernel

In [Section 6.4](#), it was shown how to exploit the enumeration of irredundant forests of [Section 6.1](#) to address the frequent subforest problem. We can reuse this result to define a new kernel on trees: the forest kernel. Recall that \mathcal{F} denotes the set of irredundant forests; similarly to [Equation 7.2](#), we define this new kernel K as

$$\forall T_1, T_2 \in \mathcal{T}, K(T_1, T_2) = \sum_{f \in \mathcal{F}} w_f \kappa(N_f(T_1), N_f(T_2)), \quad (9.3)$$

46: The definition of forest of subtrees is given only for irredundant forests. Here, we use T as an abuse of language for the forest $\{T\}$, which is trivially irredundant.

where $N_f(T)$ counts how many times the forest f is a *forest of subtrees* of T – as in [Definition 6.8](#)⁴⁶. We can relate $N_f(T)$ to $N_t(T)$ (where t is a tree) with $N_f(T) = \min_{t \in f} N_t(T)$. Besides, w_f is the weight associated to f and κ a kernel on \mathbb{N}, \mathbb{Z} or \mathbb{R} . As for the subtree kernel, assuming $\kappa(0, \cdot) = \kappa(\cdot, 0) = 0$ makes the sum finite and then

$$K(T_1, T_2) = \sum_{f \in \mathcal{F}(T_1) \cap \mathcal{F}(T_2)} w_f \kappa(N_f(T_1), N_f(T_2)), \quad (9.4)$$

where $\mathcal{F}(T)$ stands for the set of forests of subtrees of T – which we can enumerate via [Algorithm 15](#), from the DAG compression of T .

Since we introduced, in [Section 6.4](#), the [Algorithm 16](#), which directly enumerates the most frequent forests of subtrees in a tree database, we can compute this new kernel in a very similar way to the one we developed in [Kernel computation on DAGs](#) (p. 103).

Given a database of trees $\{T_1, \dots, T_n\}$, we compress them together into a single DAG, set the frequency threshold σ to 0 (to enumerate all the forests of subtrees) and run [Algorithm 16](#). We can compute $N_t(\cdot)$ using [Equation 2.4](#), and so the value of $N_f(\cdot)$ follows.

Actually, one must imagine that the number of enumerated forests of subtrees may be much too large to handle large databases. Since we have not yet defined the weight, and following a logic close to the discriminance weight introduced in [Section 36](#), we can mitigate the previous problem. Indeed, it is useless to enumerate the forests of subtrees whose weight would be negligible.

In the context of supervised classification, we can in fact replace the frequency function used in the enumeration by one that reflects the idea of discriminance. We only want to enumerate forests of subtrees that occur fairly frequently in certain *classes* (rather than just the trees themselves), so that we can use them to further discriminate these classes.

Note that this will allow forests present in all classes (such as leaves), but we must accept them to preserve the antimonotonicity of the frequency function. It will then be necessary to filter them to have a real discriminating weight.

What can we expect to gain from this new kernel compared to the subtree kernel? First, the computation time is likely to be longer – since all the forests of subtrees have to be enumerated in addition to the existing procedure. Second, since subtrees are forests made of singletons, the forest kernel is expected to retrieve at least the same information as the subtree kernel. But more importantly, it is expected to capture much richer information.

Consider the example in [Figure 9.4](#): the subtree kernel – via the discriminance weight – would be unable to distinguish the classes from each other, since no subtree is sufficient to discriminate (by its presence or absence) the classes. On the other hand, just by considering forests of two trees, we are able to discriminate perfectly each class. The forest kernel does not consider only the appearance (or absence) of one tree in a class, but the simultaneous appearance (or absence) of several of them.

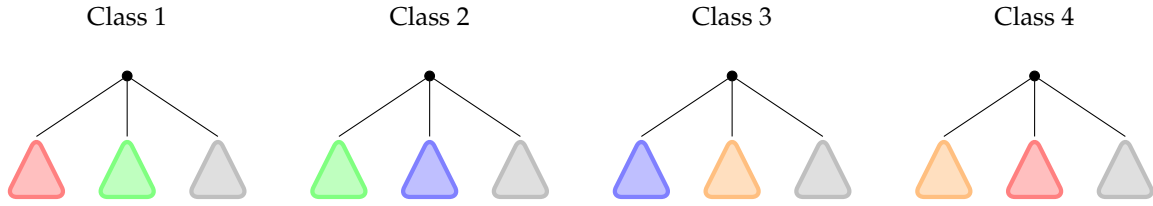


Figure 9.4: Suppose we have four classes, such that typical trees in class 1 all have (●, ●) subtrees; those in class 2 have (●, ●) subtrees; those in class 3 have (●, ●) subtrees and those in class 4 have (●, ●) subtrees. None of these four types of subtrees discriminates either by its presence, or by its absence, any of the 4 classes. On the other hand, each of the pairs of subtrees (i.e. a two element forest) discriminates by its presence the class where it appears.

Explicit enumeration with other kernels

Constructing a convolution kernel on the space of trees requires choosing a family of substructures, and finding a way to enumerate them either explicitly or implicitly – for example via a recursive kernel computation, as in [34, 35]. There is a wide variety of possible choices of substructures, an overview of which can be found in [36].

We have shown in Chapter 8 the interest of having an explicit enumeration, especially in the choice of weights. But as soon as we try to consider substructures finer than subtrees, the number of elements to enumerate increases drastically. It is therefore essential to implement strategies to reduce the number of elements enumerated, such as frequent pattern mining.

Subset trees Along with the subtree kernel, the subset tree kernel [35] uses one of the most straightforward substructures to build. Recall that a subset tree is a subgraph of the tree such that, for each node, either all children are taken or none is taken. Let T be a tree, and we denote by $\lambda(v)$ the number of possible subset trees rooted in v . We have

$$\lambda(v) = \begin{cases} 1 & \text{if } v \in \mathcal{L}(T); \\ 1 + \prod_{u \in \mathcal{C}(v)} \lambda(u) & \text{otherwise.} \end{cases} \quad (9.5)$$

The number of subset trees of T is then given by $\sum_{v \in T} \lambda(v)$ – recall that the number of subtrees of T is $\leq \#T$.

If we assume that T is a full tree of degree d and height H , we obtain the recurrence $\lambda_{h+1} = 1 + \lambda_h^d$ with $\lambda_0 = 1$. It can be shown that the general term of this kind of sequence is expressed as $\lceil c^{d^h} \rceil$, where $\lceil x \rceil$ denotes the integer part of x , and $c > 1$ is a constant [113]. The total number of subset trees of T is then given by $\sum_{i=0}^h d^{h-i} \lceil c^{d^i} \rceil$. The number of nodes of

T is given by $\frac{d^{h+1} - 1}{d - 1}$ and provides an upper bound on the number of subtrees: we see that the increase of the number of substructures is of an exponential order of magnitude.

[34]: Haussler (1999), *Convolution kernels on discrete structures*

[35]: Collins et al. (2001), ‘Convolution kernels for natural language’

[36]: Da San Martino (2009), ‘Kernel methods for tree structured data’

[35]: Collins et al. (2001), ‘Convolution kernels for natural language’

[113]: Aho et al. (1973), ‘Some doubly exponential sequences’

Subgraph At the other end of the spectrum of substructures on trees, the most general are (connected, induced) subgraphs. If we were able to enumerate them efficiently, we could construct the most generic convolution kernel on trees possible. Moreover, by constructing a weight that filters out all subgraphs except those of a particular class, we could recover the kernel associated with that class of substructures.

Naturally, such an universal kernel is very ambitious. To get an idea of the difficulty, let us look at the number of possible subgraphs of a tree.

Let T be a tree, and $v \in T$. The subgraphs of $T[v]$ can be separated into two groups:

- ▶ If the subgraph does not contain v , then it is completely contained in one of the subtrees of v . Let $\alpha(v)$ be the number of these subgraphs.
- ▶ If the subgraph contains v , then the rest of the subgraph is distributed over the subtrees of v and contains their respective roots. Let $\beta(v)$ be the number of these subgraphs.

The total number of subgraphs of $T[v]$ is given by $\alpha(v) + \beta(v)$, and thus those of T by $\alpha(\mathcal{R}(T)) + \beta(\mathcal{R}(T))$. We have

$$\begin{aligned}\alpha(v) &= \sum_{u \in \mathcal{C}(v)} (\alpha(u) + \beta(u)), \\ \beta(v) &= \prod_{u \in \mathcal{C}(v)} (1 + \beta(u)),\end{aligned}\tag{9.6}$$

with $\alpha(\bullet) = 0$ and $\beta(\bullet) = 1$.

If we resume the example of a full tree T of height h and degree d , we obtain

$$\begin{aligned}\alpha_h &= d(\alpha_{h-1} + \beta_{h-1}), \\ \beta_h &= (1 + \beta_{h-1})^d,\end{aligned}$$

with $\alpha_0 = 0$ and $\beta_0 = 1$. By induction on h one can easily show that $\alpha_h = \sum_{i=0}^h d^{i+1} \beta_{h-i}$. With [113] again, $\beta_h = \lceil \gamma^{d^h} \rceil$, where $\gamma > 1$ is a constant.

[113]: Aho et al. (1973), ‘Some doubly exponential sequences’

Remark 9.2 Equation 9.5 and Equation 9.6 can be directly computed on the DAG of T since these numbers depend only on the equivalence class of the node $v \in T$. Doing the calculation on the DAG reduces the number of recursive calls and allows to compute the quantity only once for each equivalence class.

Since $\beta_h - \lambda_h > 0$, $\gamma > c$; and the two sequences have the same exponent d^h . In light of Remark 9.1, it is not the absolute number of objects to be enumerated that is an issue, but rather the growth of this number as the size of the objects increases. Therefore, what is important here is the ratio γ/c , which determines how much faster β_h grows compared to λ_h .

Both constants should be computed to get an idea of how much simpler the enumeration of subset trees is compared to subgraphs. Typically, if γ and c are of the same order of magnitude, it may be worthwhile to attempt the enumeration of subgraphs directly; whereas if $\gamma \gg c$, it may be more reasonable to address the enumeration of subset trees prior to the enumeration of subgraphs.

APPENDIX

Technical proofs

A

Time spent amongst trees is never wasted time.

Katrina Mayer

A.1 Proof of Proposition 2.7

This proof is reproduced from [39].

We denote by D^h the set of vertices at height h in any DAG D , and denote by $\star \in \{\text{ordered}, \text{unordered}\}$ the type of trees considered.

From the forest (D_1, \dots, D_N) , we construct the DAG Δ such that (i) D_i is a subDAG of Δ for all i , (ii) $\mathcal{H}(\Delta) = \max_i \mathcal{H}(D_i)$, (iii) all vertices in Δ have degree $\max_i \deg(D_i)$, and (iv) at each height except 0 and $\mathcal{H}(D)$, $\#\Delta^h = \max_i \#D_i^h$. If Δ is placed N times under an artificial root, and then recompressed by the algorithm, indeed the output contains the recompression of the original forest. Therefore, this case is the worst possible for the algorithm, and we claim that it achieves the proposed complexity.

Let D be now a DAG with the following properties: $\#D = m$, $\mathcal{H}(D) = H$, at each height $h \notin \{0, H\}$, $\#D^h = n$ (so that $n(H-2) + 2 = m$), and all vertices have degree d . Δ is the super-DAG obtained after placing N copies of D under an artificial root. We then have $\#\Delta = 1 + Nm$ so that $O(\#\Delta) = O(Nm) = O(NHn)$ and $\deg(F) = \deg(D) = d$.

At the beginning of the algorithm, constructing the mapping $h \mapsto \Delta^h$ in one exploration of Δ has complexity $O(\#\Delta)$. We will now examine the complexity of the further steps, with respect to n, m, d, H and N . We introduce the following lemma:

Lemma A.1 *Constructing $\sigma(h)$ has complexity:*

- ▶ $O\left(\sum_{v \in \Delta^h} \#C(v) \log \#C(v)\right)$ for unordered trees;
- ▶ $O\left(\sum_{v \in \Delta^h} \#C(v)\right)$ for ordered trees.

Proof. When sorting lists of size L , merge sort is known to have $O(L \log L)$ complexity in the worst case [60]. Accordingly, we introduce


$$g^\star(x) = \begin{cases} x & \text{if } \star = \text{ordered}; \\ x(1 + \log x) & \text{if } \star = \text{unordered}. \end{cases}$$

At height h , we construct $\sigma(h) = \{f^{-1}(S) : S \in \text{Im}(f), \#f^{-1}(S) \geq 2\}$ where $f : v \in \Delta^h \mapsto C(v)$. Finding the preimage of f requires first to construct f , by copying the children of each vertex in Δ^h (in the unordered

A.1 Proof of Proposition 2.7 . . .	133
A.2 Proofs of Section 3.2	134
Preliminary reminders . . .	134
Proof of Theorem 3.1	135
Proof of Proposition 3.2 . . .	135
Proof of Proposition 3.3 . . .	136
A.3 Proof of Theorem 3.6	136
A.4 Proofs of Section 4.4	137
Proof of Theorem 4.3	139
Remaining proofs	141
Proof of Proposition 4.4 . . .	143
A.5 Proof of Proposition 7.3 . . .	144

[39]: Azais et al. (2020), ‘The weight function in the subtree kernel is decisive’

[60]: Skiena (2012), ‘Sorting and searching’

case, we also need to sort them, so that we get rid of the order and can properly compare them). Then we only need to explore once the image and check whether an element has two or more antecedents. The global cost is then $O(\sum_{v \in \Delta^h} g^*(\#C(v)))$. 

We reuse the notation g^* from the proof of [Lemma A.1](#). With respect to Δ , the complexity for constructing $\sigma(\cdot)$ is $O(Nng^*(d))$. Exploring the elements of $\sigma(h)$ for (i) choosing a vertex v_M to remain, and (ii) delete the other elements δ_M has complexity $O(Nn)$. In addition, at height $h' > h$, exploring the children to replace them or not costs $O(\sum_{v \in \Delta^{h'}} \#C(v)) = O(Ndn)$.

The global complexity $C(\Delta)$ of the algorithm is then

$$C(\Delta) = O(\#\Delta) + \sum_{h=0}^{\mathcal{H}(\Delta)} O(Nng^*(d)) + O(Nn) + \sum_{h'>h} O(Ndn).$$

Remark that $\sum_{h=0}^{\mathcal{H}(D)} O(Nn) = O(Nm) = O(\#\Delta)$, this leads to

$$C(\Delta) = O(\#\Delta g^*(\deg(F))) + O\left(Ndn \sum_{h=0}^{\mathcal{H}(D)} \sum_{h'>h} 1\right).$$

The right-hand inner sum is in $O(H^2)$. As

$$O(NdnH^2) = O(\#\Delta Hd) = O(\#\Delta \mathcal{H}(\Delta) \deg(F)),$$

this leads to our statement.

A.2 Proofs of [Section 3.2](#)

[26]: Ingels et al. (2021), 'Isomorphic Unordered Labeled Trees up to Substitution Ciphering'

The preliminary reminders and the first proof are reproduced from [26] while the last two are new material.

Preliminary reminders

Let R be a relation over sets E and F . R is a bijection if and only if

- ▶ $\forall x \in E, \exists! y \in F, x R y$;
- ▶ $\forall y \in F, \exists! x \in E, x R y$.

The converse relation R^{-1} over sets F and E is defined as $y R^{-1} x \iff x R y$. If R is a bijection, then so is R^{-1} .

Let also S be a relation over sets F and G .

The composition of R and S , denoted by $S \circ R$, is a relation over E and G , and defined as $x (S \circ R) z \iff \exists y \in F, (x R y) \wedge (y S z)$. If R and S are bijections, then so is $S \circ R$.

S is said to be finer than R if $x S y \implies x R y$. If R is a bijection, then so is S .

Proof of Theorem 3.1

Let T_1, T_2 and T_3 be trees such that $T_1 \xrightarrow{\phi} T_2$ and $T_2 \xrightarrow{\psi} T_3$. Therefore, $\phi \in \text{Isom}(T_1, T_2)$ and $\psi \in \text{Isom}(T_2, T_3)$; also, R_ϕ and R_ψ are bijections.

It should be clear that trivially, $T_1 \xrightarrow{\text{id}} T_1$. We aim to prove the following:

- ▶ $T_1 \xrightarrow{\psi \circ \phi} T_3$;
- ▶ $T_2 \xrightarrow{\phi^{-1}} T_1$.


First of all, it is trivial that $\psi \circ \phi \in \text{Isom}(T_1, T_3)$. The proof then follows directly from the reminders above and the two following lemmas.

Lemma A.2 $R_{\psi \circ \phi} = R_\psi \circ R_\phi$.

Proof. Let $x \in \mathcal{A}(T_1)$ and $z \in \mathcal{A}(T_3)$. It suffices to show

$$x R_{\psi \circ \phi} z \iff \exists y \in \mathcal{A}(T_2), (x R_\phi y) \wedge (y R_\psi z).$$


(\implies) There exists $u \in T_1$ so that $x = \bar{u}$ and $z = \overline{(\psi \circ \phi)(u)}$. Let $v = \phi(u)$ and $y = \bar{v}$; then $\bar{u} R_\phi \bar{v}$, so $x R_\phi y$; similarly $\bar{v} R_\psi \overline{\psi(v)}$ leads to $y R_\psi z$.

(\impliedby) There exists $u \in T_1$ so that $\bar{u} = x$ and $y = \overline{\phi(u)}$. Let $v = \phi(u)$. As $y R_\psi \overline{\psi(v)}$, then $\overline{\psi(v)} = z$ and it follows that $x R_{\psi \circ \phi} z$. 

Lemma A.3 $R_\phi^{-1} = R_{\phi^{-1}}$.

Proof. Let $x \in \mathcal{A}(T_1)$ and $y \in \mathcal{A}(T_2)$. It suffices to show $x R_\phi y \iff y R_{\phi^{-1}} x$.

(\implies) There exists $u \in T_1$ so that $x = \bar{u}$ and $y = \overline{\phi(u)}$. Let $v = \phi(u)$. Since $u = \phi^{-1}(v)$, $y R_{\phi^{-1}} x$.

(\impliedby) There exists $v \in T_2$ so that $\bar{v} = y$ and $x = \overline{\phi^{-1}(v)}$. Let $u = \phi^{-1}(v)$. Since $v = \phi(u)$, $x R_\phi y$. 

Proof of Proposition 3.2

Let $T_1 \xrightarrow{\phi} T_2$ and $u \in T_1$. Let ψ be the restriction of ϕ to the nodes of $T_1[u]$. We aim to prove that $T_1[u] \xrightarrow{\psi} T_2[\phi(u)]$. From the definition of tree isomorphism, it is clear that $\psi \in \text{Isom}(T_1[u], T_2[\phi(u)])$. To prove that R_ψ is bijective, it suffices to prove that R_ψ is finer than R_ϕ ; this is achieved by virtue of the following lemma.

Lemma A.4 $x R_\psi y \implies x R_\phi y$.

Proof. Let $x R_\psi y$. Then, there exists $w \in T_1[u]$, such that $x = \overline{w}$ and $y = \overline{\psi(w)}$. As $\psi = \phi$ on $T_1[u]$, this implies $y = \overline{\phi(w)}$ and therefore $x R_\phi y$. ✍

Proof of Proposition 3.3

Let $T_1 \xrightarrow{\phi} T_2$, and $u, v \in T_1$ such that $T_1[u] \xrightarrow{\psi} T_1[v]$. Let ϕ_u (respectively, ϕ_v) be the restriction of ϕ to the nodes of $T_1[u]$ (respectively, $T_1[v]$). The proof follows directly from the following commutative diagram:

$$\begin{array}{ccc}
 T_1[u] & \xrightarrow[\text{Assumption}]{\psi} & T_1[v] \\
 \uparrow \phi_u^{-1} & & \downarrow \phi_v \\
 T_2[\phi(u)] & \xrightarrow[\text{Proposition 3.2}]{\phi_v \circ \psi \circ \phi_u^{-1}} & T_2[\phi(v)]
 \end{array}$$

Proposition 3.2 is indicated on the left and right sides of the diagram.

A.3 Proof of Theorem 3.6

Assume $0 \leq h \leq \mathcal{H}(D)$ and $q \in D^h$ are fixed, where $D = \mathfrak{R}(T)$.

- ▶ From lines 7 to 14, we test each node $u \in T(q)$ against each representative $s(P)$ for $P \in \mathfrak{P}$. We assume constant operation costs for lines 9, 12, 13 and 14. Since all nodes $u \in T(q)$ have the same equivalence class, for a fixed u , the cost of testing it against the partition is of the order of $\#\mathfrak{P} \times C_-(q)$. Storing the nodes of $\mathcal{D}(u)$ in N has complexity $O(\#\mathcal{D}(u))$ per $u \in T(q)$ (except those chosen to represent their part). Actually, the total number $\#N$ of descendants can not exceed $\#T$. In addition, since $\#\mathfrak{P}$ is smaller than or equal to the number of nodes u already processed, the overall cost of this step is $O(\#T(q)^2 C_-(q) + \#T)$.
- ▶ We then delete from $T(q')$, for $q' \in \mathcal{D}(q)$, the elements stored in N . Since the elements of N are the same for each vertex q' , an efficient approach is to place the nodes of N in a hash table [59] with complexity $O(\#N)$ and test whether each $v \in T(q')$ is also in N with complexity $O(1)$. The overall complexity for this step is therefore of $O(\#N + \sum_{q' \in \mathcal{D}(q)} \#T(q'))$, assuming a perfect hash function [114]. Since $\#N \leq \#T$ and $\sum_{q' \in \mathcal{D}(q)} \#T(q') \leq \#T$ (with equality when q is the root), the complexity can be simplified to $O(\#T)$.
- ▶ From lines 18 to 23, we create the new vertices in the DAG. Let us suppose $P \in \mathfrak{P}$ and $u \in P$ fixed. We assume that f_ϕ was obtained, and stored, when checking $T[u] \sim T[s(P)]$. Copying it on the new arc costs $\#\mathcal{A}(T[s(P)])$ – that we bound by $\#\mathcal{A}(T)$. Combining all this and summing over $P \in \mathfrak{P}$ and $u \in P$, and also noticing that $\sum_{P \in \mathfrak{P}} \sum_{u \in P} 1 = \#T(q)$, the overall complexity of this step is therefore $O(\#T(q) \#\mathcal{A}(T))$.

[59]: Knuth (1973), ‘The art of computer programming. Vol. 3, Sorting and Searching’

[114]: Lu et al. (2006), ‘Perfect hashing for network applications’

Summing over h and $q \in D^h$, the overall complexity of the algorithm is of the order of

$$\sum_{h=\mathcal{H}(D)}^0 \sum_{q \in D^h} (\#T(q)^2 C_{\sim}(q) + \#T(q) \#A(T) + 2\#T).$$

We break this sum into pieces. Note that $\sum_{h=\mathcal{H}(D)}^0 \sum_{q \in D^h} \#T(q) = \#T$ and that $\sum_{h=\mathcal{H}(D)}^0 \sum_{q \in D^h} 1 = \#D$.

- Assuming that $T' \in \mathcal{S}(T) \implies C_{\sim}([T']) = O(C_{\sim}([T]))$, and since $\sum_n n^2 \leq (\sum_n n)^2$ for positive n 's,

$$\sum_{h=\mathcal{H}(D)}^0 \sum_{q \in D^h} \#T(q)^2 C_{\sim}(q) = O(C_{\sim}([T]) \#T^2).$$

- In addition,

$$\sum_{h=\mathcal{H}(D)}^0 \sum_{q \in D^h} (\#T(q) \#A(T) + 2\#T) = (\#A(T) + 2\#D) \#T.$$

Finally, since $O(1)$ is negligible compared to $O(C_{\sim}([T]))$, and since $\#A(T) \leq \#T$ and $\#D \leq \#T$, the final complexity of [Algorithm 6](#) is $O(\#T^2 C_{\sim}([T]))$.

Remark A.1 If we reject the hypothesis of a perfect hash function, the worst case complexity becomes $O(n)$ instead of $O(1)$. The complexity associated with the concerned step becomes $O(\#T^2)$, which is reflected in the final complexity to yield $O(\#T^2(C_{\sim}([T]) + \#T))$.

Since $C_{\sim}([T])$ measures the complexity of attesting for the existence of an isomorphism up to a ciphering between two trees, this implies attesting to at least the existence of a topological isomorphism, and thus this complexity is at least as high as for the AHU algorithm – which is at least $O(\#T)$, see [The Aho, Hopcroft & Ullman algorithm](#) (p. 13). Therefore, $\#T = O(C_{\sim}([T]))$ and the claimed complexity remains.

A.4 Proofs of Section 4.4

Whether it is bags or collections, backtracking explores the possible associations between their elements (nodes or set of nodes), building permutations.

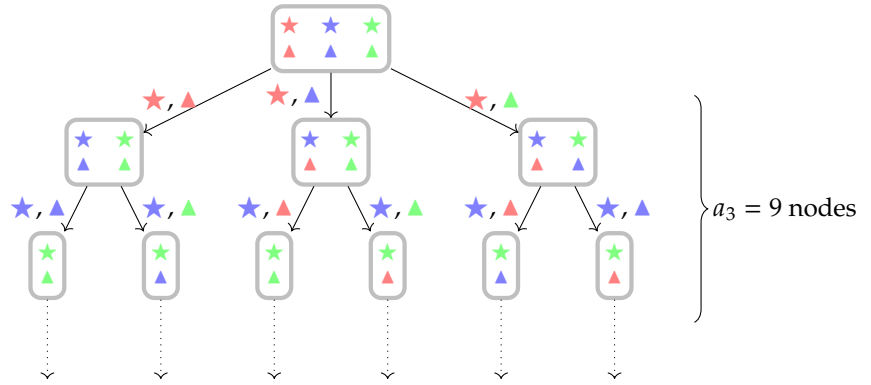
Backtracking tree Given two sets of $n \geq 2$ objects, there are $n!$ ways to map them together, which correspond to the leaves of the backtracking search tree. The size of the tree itself is directly related to the number of operations a_n required to create all those permutations of n elements [75] – see also OEIS sequence A038156⁴⁷:

$$a_n = n! \sum_{i=1}^{n-1} \frac{1}{i!}. \quad (\text{A.1})$$

[75]: Knuth (2005), *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations* (*Art of Computer Programming*)

47: OEIS Foundation Inc. (2022), The On-Line Encyclopedia of Integer Sequences, <https://oeis.org/A038156>

Figure A.1: Backtracking tree for mapping two sets of $n = 3$ objects. The mappings made at each step are indicated on the arrows. Note that the last mappings are not shown, but would be decided, within the context of bags and collections, respectively by [Deduction Rule 4.1](#) and [Deduction Rule 4.4](#); and therefore they can be excluded from the backtracking tree.



We illustrate the backtracking tree with $n = 3$ in [Figure A.1](#).

If you have to process several bags in a row, you have to duplicate the backtracking tree of the second bag under each of the leaves of the first, then the tree of the third under each of the leaves of the second, etc. Simply with two bags, of cardinality p and q , processed in this order, the total size of the backtracking tree would be given by $1 + a_p + p!a_q$.

The question naturally arises of the order in which the bags should be processed to obtain as small a tree as possible. Similar questions emerge regarding the order in which to process collections, which to focus on if there is a choice between bags and collections, etc. Note that in the sequel, we ignore recursive mappings (from parents) or bag/collection splits (from `SPLITCHILDREN`). In other words, this is a pessimistic scenario where the entire search tree is fixed by the order of processing of bags and collections, without any further modification.

A variadic function We introduce here an abstract variadic function whose value corresponds to the size of the backtracking tree when evaluated with the cardinalities of bags and collections – as specified in the next paragraph.

Given a finite sequence of integer tuples $(n_i, \alpha_i), i \in \llbracket 1, p \rrbracket$ with $n_i \geq 2$ and $\alpha_i \geq 1$, we define the following variadic function

$$f((n_1, \alpha_1), \dots, (n_p, \alpha_p)) = \begin{cases} \alpha_1 a_{n_1} & \text{if } p = 1; \\ \alpha_1 [a_{n_1} + n_1! f((n_2, \alpha_2), \dots, (n_p, \alpha_p))] & \text{otherwise.} \end{cases} \quad (\text{A.2})$$

The goal is twofold: connect the bags and collections to the tuples (n_i, α_i) , and study the behavior of f – where our goal is to minimize f with regard to the order of the tuples, and therefore the order in which bags and collections are processed.

Link between f , bags and collections Concerning bags, we already saw that if we have two bags of cardinality n and m , treating them in this order induces a tree of size $1 + a_n + n!a_m$. If we now have p bags of respective sizes n_1, \dots, n_p , and that we process them successively in this order, the resulting tree has $1 + f((n_1, 1), \dots, (n_p, 1))$ nodes – i.e. a bag of cardinality n corresponds to the tuple $(n, 1)$.

Concerning collections, we can in fact imagine two strategies to deal with them. Let C be a collection and n so that $\#C(n) > 0$:

- ▶ either we create the $(\#C(n))!$ possible bags by associating the sets of $C(n)$, and then process these bags one after the other;
- ▶ or we create a bag, we process it entirely, then another bag on the collection which now contains $\#C(n) - 1$ sets, we process it entirely, etc.

The first strategy amounts to processing a tuple $(\#C(n), 1)$ first, then $\#C(n)$ times a tuple $(n, 1)$. The second strategy amounts to processing the tuple $(n, \#C(n))$ first⁴⁸, then the tuple $(n, \#C(n) - 1)$, etc., up to the tuple $(n, 1)$.

The second strategy is better to minimize f , as per the following lemma.

Lemma A.5 For any $n, \alpha \geq 2$,

$$f((n, \alpha), (n, \alpha - 1), \dots, (n, 1)) < f((\alpha, 1), \underbrace{(n, 1), \dots, (n, 1)}_{\alpha})$$

Proof. First, we compute each of the two terms involved.

- ▶ A proof by induction on α without difficulty allows to show that

$$f((n, \alpha), (n, \alpha - 1), \dots, (n, 1)) = a_n \sum_{k=0}^{\alpha-1} \frac{\alpha!}{(\alpha - 1 - k)!} (n!)^k.$$

- ▶ Again, an easy calculation by induction on α allows to show that

$$f(\underbrace{(n, 1), \dots, (n, 1)}_{\alpha}) = a_n \sum_{k=0}^{\alpha-1} (n!)^k; \text{ and thus the right-hand term equals } a_\alpha + \alpha! a_n \sum_{k=0}^{\alpha-1} (n!)^k.$$

The conclusion is immediate when comparing term by term each of the two sums. 

Therefore, $C(n)$ correspond to the tuples $(n, \#C(n)), (n, \#C(n) - 1)$, up to $(n, 1)$; the size of the backtracking tree generated by $C(n)$ is therefore expressed by $1 + f((n, \#C(n)), (n, \#C(n) - 1), \dots, (n, 1))$.

Proof of Theorem 4.3

Given $p \geq 2$ tuples (n_i, α_i) , and denoting the first two (m, α) and (n, β) , we are interested in the conditions on m, n, α, β under which we would have

$$\Delta f = f((m, \alpha), (n, \beta), \dots, (n_p, \alpha_p)) - f((n, \beta), (m, \alpha), \dots, (n_p, \alpha_p)) \leq 0.$$

If $\Delta f \leq 0$, we process the tuples (m, α) and (n, β) in this order; on the other hand, if $\Delta f \geq 0$, we must swap them to minimize f . In other words, we want to define a strategy for when to swap the first two tuples.

Remark A.2 Note that, by virtue of the bubble sort principle [115], it is enough to know when to swap the first two arguments to minimize globally f . Indeed, we can then recursively permute the subsequent

48: Indeed, we branch on $\#C(n)$ possibilities (mapping the first set of $C_1(n)$ with each of the others from $C_2(n)$), and then process the tree generated by a bag of cardinality n .

[115]: Astrachan (2003), 'Bubble sort: an archaeological algorithmic analysis'

[60]: Skiena (2012), ‘Sorting and searching’

elements, and obtain a permutation that minimizes globally f in $\mathcal{O}(p^2)$ swaps at most. Bubble sort has a bad complexity – compared to merge sort in $\mathcal{O}(p \log p)$ [60], but it is only used here as a theoretical argument. The goal is to find a condition on the tuples to optimize f .

After some elementary manipulations using the definition of f in Equation A.2, Δf can be rewritten as

$$\Delta f = \beta a_n(\alpha m! - 1) - \alpha a_m(\beta n! - 1). \tag{A.3}$$

In the following we assume $m \geq n$ and we examine the implications on α, β .


Case of a tie If $m = n$, then the tuple with $\min(\alpha, \beta)$ must be placed first to minimize f , by virtue of the following lemma.

Lemma A.6 *With $m = n \geq 2$, and $\alpha, \beta \geq 1$, we have*

$$\Delta f = a_n(\alpha - \beta) \geq 0 \iff \alpha \geq \beta.$$

Case of $\beta \geq 2$ When $m > n$, if $\beta \geq 2$, then the tuple (m, α) must be processed first to minimize f , no matter the value of α .

Lemma A.7 *With $m > n \geq 2$, $\alpha \geq 1$ and $\beta \geq 2$, we have $\Delta f < 0$.*

Proof. The proof is deferred to [Remaining proofs \(p. 141\)](#). 

Case of $\beta = 1$ If $m > n$ and $\beta = 1$, the tuple $(n, 1)$ must be processed before (m, α) to minimize f .

Lemma A.8 *With $m > n \geq 2$, $\alpha \geq 1$ and $\beta = 1$, we have $\Delta f > 0$.*

Proof. The proof is also deferred to [Remaining proofs \(p. 141\)](#). 

Summary The consequences of [Lemma A.6](#), [Lemma A.7](#) and [Lemma A.8](#) are summarized in [Table A.1](#).

A strategy for dealing with bags and collections can be derived from this.

- ▶ When you have bags and collections, since bags are represented by tuples $(n, 1)$ and collections by tuples (m, α) with $\alpha \geq 2$, [Table A.1](#) shows that starting by treating bags minimizes the size of the backtracking tree;
- ▶ if there are only bags left, i.e. $\alpha_i = 1$ for all i , [Table A.1](#) shows that processing the bags by increasing cardinality ensures that f is minimal;

	$\beta = 1$	$\beta \geq 2$
$\alpha = 1$	$\min(n, m)$	m
$\alpha \geq 2$	n	$\max(n, m)$ *

Table A.1: Strategy for deciding the first tuple to process between (m, α) and (n, β) to minimize f . ★: If $m = n$, take $\min(\alpha, \beta)$.

- if there are only collections left, say $C(n)$ and $C'(m)$ – with $\#C(n) \geq 2$ and $\#C'(m) \geq 2$, Table A.1 shows that the size of the tree is minimized by starting with $\max(n, m)$ – and $\min(\#C(n), \#C'(m))$ if $n = m$.

Remaining proofs

Preliminaries From Equation A.1, we define $b_n = \sum_{i=1}^{n-1} \frac{1}{i!}$ so that $a_n = n!b_n$. b_n converges to $e - 1 \approx 1.718$ so $n! \leq a_n \leq (e - 1)n!$. We denote by r_n the remainder of the series (b_n) , i.e. $r_n = \sum_{i=n}^{\infty} \frac{1}{i!}$, so that $e - 1 = b_n + r_n$.

Finally, given $k \geq 0$, we define $s_{n,k} = b_{n+k} - b_n = \sum_{i=n}^{n+k-1} \frac{1}{i!}$.

From Equation A.3, with the notations we have just introduced, we rewrite Δf as

$$\Delta f = \alpha \beta b_n n! m! \left[1 - \frac{1}{\alpha m!} - \frac{b_m}{b_n} \left(1 - \frac{1}{\beta n!} \right) \right].$$

Since we suppose moreover that $m = n + k$, we rewrite $m = n + k$, with $k \geq 0$, and with $b_{n+k} = b_n + s_{n,k}$, after simplifications,

$$\Delta f = -\alpha \beta b_n n! (n + k)! \left(\frac{1}{\alpha (n + k)!} + \frac{s_{n,k}(\beta n! - 1) - b_n}{b_n \beta n!} \right). \quad (\text{A.4})$$

Proof of Lemma A.7 With Equation A.4, $\Delta f < 0$ is equivalent to

$$\frac{1}{\alpha (n + k)!} + \frac{s_{n,k}(\beta n! - 1) - b_n}{b_n \beta n!} > 0.$$

It suffices to show that the right-hand term is non-negative, i.e. that $s_{n,k}(\beta n! - 1) \geq b_n$. If we show that this is true for $k = 1$, since $s_{n,k}$ is increasing in k , the inequality will hold for any $k \geq 1$. Since $s_{n,1} = 1/n!$, the case $k = 1$ is equivalent to

$$\beta - \frac{1}{n!} \geq b_n \iff \beta \geq b_n + \frac{1}{n!} = b_{n+1}$$

which holds since $\beta \geq 2 \geq e - 1 \geq b_{n+1}$.

Proof of Lemma A.8 With Equation A.4, $\Delta f > 0$ is equivalent to

$$\frac{1}{\alpha (n + k)!} + \frac{s_{n,k}(n! - 1) - b_n}{b_n n!} < 0;$$

which, in turn, after some elementary manipulations, can be rewritten as

$$\frac{1}{s_{n,k}} \left(1 - \frac{n!}{\alpha (n + k)!} \right) > \frac{n! - 1}{b_n}. \quad (\text{A.5})$$

To show this result, we proceed by disjunction of cases on n .


Lemma A.9 Equation A.5 holds for $n = 2$, and $k, \alpha \geq 1$.

Proof. We rewrite Equation A.5 as

$$\frac{1}{s_{2,k}} \left(1 - \frac{2!}{\alpha(2+k)!} \right) > \frac{2! - 1}{b_2} = 1.$$

If the inequality holds for $\alpha = 1$, it will hold for any value of α . Let $\alpha = 1$. Multiplying each side of the inequality by $s_{2,k}$, we obtain

$$\sum_{i=2}^{k+1} \frac{1}{i!} < 1 - \frac{2}{(k+2)!} \iff \sum_{i=2}^{k+2} \frac{1}{i!} < 1 - \frac{1}{(k+2)!}.$$

The left-hand term on the right is increasing in k and tends to $e - 2 \approx 0.71$; whereas the right-hand term is also increasing and equals $1 - \frac{1}{3!} = \frac{5}{6} \approx 0.83$ when $k = 1$. Therefore $\sum_{i=2}^{k+2} \frac{1}{i!} < e - 2 < \frac{5}{6} < 1 - \frac{1}{(k+2)!}$. 

Since $\alpha \geq 1$ and $k \geq 1$,

$$\frac{1}{s_{n,k}} \left(1 - \frac{n!}{\alpha(n+k)!} \right) > \frac{1}{s_{n,k}} \left(1 - \frac{1}{n+1} \right);$$

so Equation A.5 holds if the following holds:

$$\frac{1}{s_{n,k}} \frac{n}{n+1} > \frac{n! - 1}{b_n}.$$

Note that the left-hand term is a (strictly) decreasing function of k ; therefore the inequality holds if we can prove that it holds when $k \rightarrow \infty$. In light of $\lim_{k \rightarrow \infty} s_{n,k} = r_n$, and with $b_n = (e - 1) - r_n > 0$, this amounts to prove that

$$(n! - 1) \left(1 + \frac{1}{n} \right) \frac{r_n}{e - 1 - r_n} < 1. \quad (\text{A.6})$$

We end the disjunction of cases with the following lemmas.

Lemma A.10 Equation A.6 holds for $n = 3$.

Proof. With $n = 3$, $r_3 = e - 5/2$ and


$$(3! - 1) \left(1 + \frac{1}{3} \right) \frac{r_3}{e - 1 - r_3} \approx 0.97 < 1.$$



Lemma A.11 Equation A.6 holds for $n \geq 4$.

Proof. With $n! - 1 < n!$, we bound the left-hand term of Equation A.6 by

$$\left(1 + \frac{1}{n}\right) \frac{n!r_n}{e - 1 - r_n}$$

Note that each of the terms composing this product is a decreasing function⁴⁹ of n , so the product is decreasing as well. It turns out that starting from $n = 4$, this term is smaller than 1 and so it is for $n \geq 4$. Indeed, we have $r_4 = e - 8/3$ and the product is evaluated as $\approx 0.92 < 1$. 

49: Indeed, r_n is a decreasing function of n , and we can show that $n!r_n$ is too, at the expense of a tedious but direct calculation.


Proof of Proposition 4.4

Using the definition of f given by Equation A.2, we have the following.

$$\textbf{Lemma A.12} \quad f((n_1, \alpha_1), \dots, (n_p, \alpha_p)) \leq (e - 1) \prod_{i=1}^p \alpha_i n_i! \sum_{i=0}^{p-1} \frac{1}{2^i}$$

Proof. By induction on p . $f((n, \alpha)) = \alpha a_n \leq (e - 1)\alpha n!$. Let $p \geq 2$ be fixed. We have

$$\begin{aligned} f((n_1, \alpha_1), \dots, (n_p, \alpha_p)) &= \alpha_1 [a_{n_1} + n_1! f((n_2, \alpha_2), \dots, (n_p, \alpha_p))] \\ &\leq (e - 1)\alpha_1 n_1! \left(1 + \prod_{i=2}^p \alpha_i n_i! \sum_{i=0}^{p-2} \frac{1}{2^i}\right) \\ &= (e - 1) \prod_{i=1}^p \alpha_i n_i! \left(\frac{1}{\prod_{i=2}^p \alpha_i n_i!} + \sum_{i=0}^{p-2} \frac{1}{2^i}\right). \end{aligned}$$

and since $\alpha_i \geq 1, n_i \geq 2$, we have $\prod_{i=2}^p \alpha_i n_i! \geq 2^{p-1}$. 

Reconnecting with the bags and collections:

- ▶ A bag $B \in \mathbb{B}$ contributes a tuple $(\#B, 1)$ and therefore to $\#B!$ in the product;
- ▶ A collection $C \in \mathbb{C}$ and an integer $n \in \mathbb{N}$ so that $\#C(n) > 0$ contributes the tuples $(n, \#C(n)), (n, \#C(n) - 1), \dots, (n, 1)$ so a total combined of $\#C(n)!n!^{\#C(n)}$.

We retrieve exactly $N(\mathbb{B}, \mathbb{C})$ as in Equation 4.2. Bounding the sum by 2 leads to $2(e - 1)N(\mathbb{B}, \mathbb{C})$.

Note that this result is true whether or not we optimize the order of tuples to minimize f . The bound is rather high since a lot of information is lost by using $n! \geq 2$.

By sorting the tuples, one can imagine the gain on the bound in light of Lemma A.13 – showing what can be gained (or lost) if we optimize (or not) the order of the tuples.

$$\textbf{Lemma A.13} \quad |\Delta f| \underset{k \rightarrow \infty}{\sim} (e - 1)\alpha |1 - \beta c_n| (n + k)!, \text{ where } c_n \in (0, 1) \text{ depends only on } n \text{ and } \Delta f \text{ is defined in Equation A.3}$$

Proof. We have

$$|\Delta f| = \alpha \beta n!(n+k)! \left| b_n \left(1 - \frac{1}{\alpha(n+k)!} \right) - b_{n+k} \left(1 - \frac{1}{\beta n!} \right) \right|$$

$$\underset{k \rightarrow \infty}{\sim} \alpha \beta n!(n+k)! \left| b_n - (e-1) \left(1 - \frac{1}{\beta n!} \right) \right|.$$

Since $b_n = (e-1) - r_n$, the term between $|\cdot|$ can be simplified to

$$\frac{e-1}{\beta n!} - r_n = \frac{e-1}{\beta n!} (1 - \beta c_n),$$

where $c_n = \frac{n!r_n}{e-1} \in (0, 1)$. Indeed, since $r_1 = e_1$ and $n!r_n$ is strictly decreasing in n , $n!r_n \in (0, e-1)$

Finally, $|\Delta f| \underset{k \rightarrow \infty}{\sim} (e-1)\alpha |1 - \beta c_n| (n+k)!$ 🍃

A.5 Proof of Proposition 7.3

[39]: Azaïs et al. (2020), ‘The weight function in the subtree kernel is decisive’

This proof is reproduced from [39] and is mainly based on the following technical lemma, whose statement requires the following notation. If u is a vertex of a tree T , $\mathfrak{F}(u)$ denotes the family of u , i.e., the set composed of the ascendants of u , u , and the descendants of u in T . We recall that $\mathcal{D}(u)$ stands for the latter.

Lemma A.14 *Let $u, v \in T_i$, $i \in \{1, 2\}$. One has*

$$K(T_i^u, T_i^v) = K(T_i, T_i) - \sum_{x \in \mathcal{B}_{u,v}} \omega_{T_i[x]} + K(\tau_{\mathcal{H}(u)}, \tau_{\mathcal{H}(v)}),$$

where

$$\mathcal{B}_{u,v} = \begin{cases} \mathcal{D}(u) \cup \{u\} & \text{if } u = v, \\ \mathfrak{F}(u) \cup \mathfrak{F}(v) & \text{otherwise.} \end{cases} \quad (\text{A.7})$$

Let $u \in T_1$ and $v \in T_2$. Then,

$$K(T_1^u, T_2^v) = K(\tau_{\mathcal{H}(u)}, \tau_{\mathcal{H}(v)}).$$

Proof. We begin with the case $u \neq v$. The result relies on the following decomposition, which is valid under the assumptions made on T_i and the sequence (τ_h) ,

$$\mathcal{S}(T_i^u) \cap \mathcal{S}(T_i^v) = [\mathcal{S}(T_i) \setminus \{T_i[z] : z \in \mathfrak{F}(u) \cup \mathfrak{F}(v)\}] \cup [\mathcal{S}(\tau_{\mathcal{H}(u)}) \cap \mathcal{S}(\tau_{\mathcal{H}(v)})].$$

Together with Equation 7.4,

$$K(T_i^u, T_i^v) = \sum_{\theta \in \mathcal{S}(T_i) \setminus \{T_i[z] : z \in \mathfrak{F}(u) \cup \mathfrak{F}(v)\}} w_\theta \mathbf{N}_\theta(T_i^u) \mathbf{N}_\theta(T_i^v)$$

$$+ \sum_{\theta \in \mathcal{S}(\tau_{\mathcal{H}(u)}) \cap \mathcal{S}(\tau_{\mathcal{H}(v)})} w_\theta \mathbf{N}_\theta(T_i^u) \mathbf{N}_\theta(T_i^v).$$

If $\theta \in \mathcal{S}(\tau_{\mathcal{H}(u)}) \cap \mathcal{S}(\tau_{\mathcal{H}(v)})$, then $N_\theta(T_i^z) = N_\theta(\tau_{\mathcal{H}(z)})$, $z \in \{u, v\}$, because, for any $h > 0$, τ_h is not a subtree of T_0 or T_1 by assumption. Thus,


$$\begin{aligned} \sum_{\theta \in \mathcal{S}(\tau_{\mathcal{H}(u)}) \cap \mathcal{S}(\tau_{\mathcal{H}(v)})} w_\theta N_\theta(T_i^u) N_\theta(T_i^v) &= \sum_{\theta \in \mathcal{S}(\tau_{\mathcal{H}(u)}) \cap \mathcal{S}(\tau_{\mathcal{H}(v)})} w_\theta N_\theta(\tau_{\mathcal{H}(u)}) N_\theta(\tau_{\mathcal{H}(v)}) \\ &= K(\tau_{\mathcal{H}(u)}, \tau_{\mathcal{H}(v)}), \end{aligned} \quad (\text{A.8})$$

in light of Equation 7.4 again. Furthermore, if $\theta \in \mathcal{S}(T_i) \setminus \{T_i[z] : z \in \mathfrak{F}(u) \cup \mathfrak{F}(v)\}$, then $N_\theta(T_i^z) = N_\theta(T_i)$, $z \in \{u, v\}$, and

$$\begin{aligned} \sum_{\theta \in \mathcal{S}(T_i) \setminus \{T_i[z] : z \in \mathfrak{F}(u) \cup \mathfrak{F}(v)\}} w_\theta N_\theta(T_i^u) N_\theta(T_i^v) &= \sum_{\theta \in \mathcal{S}(T_i) \setminus \{T_i[z] : z \in \mathfrak{F}(u) \cup \mathfrak{F}(v)\}} w_\theta N_\theta(T_i) N_\theta(T_i) \\ &= \sum_{\theta \in \mathcal{S}(T_i)} w_\theta N_\theta(T_i) N_\theta(T_i) \\ &\quad - \sum_{\theta \in \{T_i[u] : u \in \mathfrak{F}(v) \cup \mathfrak{F}(w)\}} w_\theta N_\theta(T_i) N_\theta(T_i) \\ &= K(T_i, T_i) - \sum_{\theta \in \{T_i[z] : z \in \mathfrak{F}(u) \cup \mathfrak{F}(v)\}} w_\theta, \end{aligned} \quad (\text{A.9})$$

since $N_\theta(T_i) = 1$ because of the first assumption on T_i . Equation A.8 and Equation A.9 state the first result. When $u = v$, the decomposition is slightly different,

$$\mathcal{S}(T_i^u) = [\mathcal{S}(T_i) \setminus \{T_i[z] : z \in \{u\} \cup \mathcal{D}(u)\}] \cup \mathcal{S}(\tau_{\mathcal{H}(u)}),$$

but the rest of the proof is similar. Finally, the formula for $K(T_1^u, T_2^v)$ is a direct consequence of the third assumption on T_1, T_2 and the sequence (τ_h) . 


By virtue of the previous lemma, one can derive the following result on the quantity Δ_x^i defined by Equation 7.5.

Lemma A.15 *Let $x \in T_i$, $i \in \{1, 2\}$. One has*

$$\Delta_x^i = K(T_i, T_i) - \mathbb{E}_u \left[\sum_{z \in \mathcal{B}_{x,u}} w_{T_i[z]} \right].$$

Proof. In light of Lemma A.14, one has

$$\Delta_x^i = K(T_i, T_i) - \mathbb{E}_u \left[\sum_{z \in \mathcal{B}_{x,u}} w_{T_i[z]} \right] + \mathbb{E}_u [K(\tau_{\mathcal{H}(x)}, \tau_{\mathcal{H}(u)})] - \mathbb{E}_v [K(\tau_{\mathcal{H}(x)}, \tau_{\mathcal{H}(v)})].$$

By assumption on the stochastic model of random trees, $\mathcal{H}(u)$ and $\mathcal{H}(v)$ have the same distribution; thus $\mathbb{E}_u [K(\tau_{\mathcal{H}(x)}, \tau_{\mathcal{H}(u)})] = \mathbb{E}_v [K(\tau_{\mathcal{H}(x)}, \tau_{\mathcal{H}(v)})]$, which states the expected result. 

The next decomposition is useful to prove the result of interest. If c_h^i denotes the number of subtrees of height h appearing in T_i , $h \geq 0$, then

the probability of picking a particular vertex u is $P_\rho(\mathcal{H}(u))/c_{\mathcal{H}(u)}^i$ and thus

$$\mathbb{E}_u \left[\sum_{z \in \mathcal{B}_{x,u}} w_{T_i[z]} \right] = \frac{P_\rho(\mathcal{H}(x))}{c_{\mathcal{H}(x)}^i} \sum_{z \in \{x\} \cup \mathcal{D}(x)} w_{T_i[z]} + \sum_{u \in T_i \setminus \{x\}} \frac{P_\rho(\mathcal{H}(u))}{c_{\mathcal{H}(u)}^i} \sum_{z \in \mathcal{B}_{x,u}} w_{T_i[z]}.$$

In addition, for $u \in T_i \setminus \{x\}$,

$$\sum_{z \in \{x\} \cup \mathcal{D}(x)} \omega_{T_i[z]} = K(T_i[x], T_i[x]), \quad (\text{A.10})$$

$$\sum_{z \in \mathcal{B}_{x,u}} \omega_{T_i[z]} = K(T_i, T_i) - \sum_{z \notin \mathcal{B}(x) \cup \mathcal{B}(u)} \omega_{T_i[z]}. \quad (\text{A.11})$$

Equation A.10 and Equation A.11 together with Lemma A.15 show that

$$\Delta_x^i = \frac{P_\rho(\mathcal{H}(x))}{c_{\mathcal{H}(x)}^i} (K(T_i, T_i) - K(T_i[x], T_i[x])) + \sum_{u \in T_i \setminus \{x\}} \frac{P_\rho(\mathcal{H}(u))}{c_{\mathcal{H}(u)}^i} \sum_{z \notin \mathcal{B}(x) \cup \mathcal{B}(u)} \omega_{T_i[z]}.$$

The left-hand term (and the right-hand term when $w_{T_i} > 0$) is null if and only if $x = \mathcal{R}(T_i)$, which shows the first result. In addition,

$$\Delta_x^i \geq \frac{P_\rho(\mathcal{H}(x))}{c_{\mathcal{H}(x)}^i} (K(T_i, T_i) - K(T_i[x], T_i[x])),$$

which states the expected Equation 7.6 with $P_\rho(0) \leq P_\rho(\mathcal{H}(x))$ (true if $\rho > H/2$) and $c_{\mathcal{H}(x)}^i \leq \#\mathcal{L}(T_i)$. The conclusion comes from the fact that the probability of drawing a vertex x of height greater than h is $G_\rho(h)$.

A bijection between FDAGs and row-Fishburn matrices

B

It must be October, the trees are falling away and showing their true colors.

Charmaine J Forde

This (short) chapter is dedicated to the proof of [Theorem 6.5](#), reproduced from [32], which is in two steps. First, we recall the natural bijection between FDAGs and their adjacency matrices; the latter are then put into bijection with the row-Fishburn matrices.

- [B.1 Equivalence between FDAGs and reduced adjacency matrices . . . 147](#)
- [B.2 Reduced adjacency matrix to incremental adjacency matrix . . . 147](#)
- [B.3 Incremental adjacency matrix to reduced adjacency matrix 148](#)

[32]: Ingels et al. (2022), ‘Enumeration of Irredundant Forests’

B.1 Equivalence between FDAGs and reduced adjacency matrices

Let $D = (v_0, \dots, v_n)$ be a FDAG constructed in k steps from D_0 in the enumeration tree defined in [Section 6.1](#). The adjacency matrix of D is defined as $A = (A_{i,j})_{i,j \in \llbracket n,0 \rrbracket^2}$ where, if m is the multiplicity of v_j in $C(v_i)$, then $A_{i,j} = m$ – possibly 0 if $v_j \notin C(v_i)$. By construction of D , as v_n is the last inserted vertex, it has no parents, so $A_{n,\cdot}$ is a column of zeros; and as v_0 is a leaf, it has no children, so $A_{\cdot,0}$ is a row of zeros. We define the *reduced* adjacency matrix M as the matrix A deprived of this column and this row. Therefore, $M = (A_{i,j})_{i \in \llbracket n,1 \rrbracket, j \in \llbracket n-1,0 \rrbracket}$. Naturally, one can reconstruct the adjacency matrix (and thus D) from the reduced matrix by adding a row and a column of zeros.

As a vertex can not be a parent to any vertex introduced after it, we have $A_{i,j} = 0$ for all $i \leq j$ – so that M is an upper-triangular matrix. In addition, as all vertices except v_0 have at least one child, there is at least one non-zero entry in each row of M . Therefore, M is a row-Fishburn matrix. However, we have no guarantee that this matrix verifies $\text{size}(M) = k$.

B.2 Reduced adjacency matrix to incremental adjacency matrix

Let $D = (v_0, \dots, v_n)$ be a FDAG, and M its reduced adjacency matrix. Let M_i be the row of M corresponding to $C_\psi(v_i)$. The incremental adjacency matrix \hat{M} is defined as:

$$\begin{cases} \hat{M}_1 = M_1 \\ \hat{M}_{i+1} = M_{i+1} \ominus M_i \end{cases}$$

where the \ominus operation is defined as follows: given two rows $a_0 \cdots a_n$ and $b_0 \cdots b_n$, then denoting $j = \min\{i : a_i \neq b_i\}$, and $c = a_j - b_j$,

$$\begin{array}{r} a_0 \cdots a_{j-1} \quad a_j \quad a_{j+1} \cdots a_n \\ \ominus b_0 \cdots b_{j-1} \quad b_j \quad b_{j+1} \cdots b_n \\ \hline = 0 \cdots 0 \quad c \quad a_{j+1} \cdots a_n \end{array} .$$

We claim that this new matrix \hat{M} is a row-Fishburn matrix of size k , if $D \in E_k$. Actually, since M was already a row-Fishburn matrix, we just have to check that the size is correct. Let us consider v_i and v_{i+1} . The vertex v_{i+1} has been constructed from v_i by using either (\uparrow) or (\nearrow) , and potentially several (\curvearrowright) after that – let us say $p \geq 0$ times. Therefore, if the claim is correct, the sum over \hat{M}_{i+1} should be exactly $p + 1$. Consider the operation by which v_{i+1} was added in the first place:

- (\uparrow) $C_\psi(v_{i+1})$ is reduced to a single element a , such that $a >_{\text{lex}} C_\psi(v_i)$. Therefore, the index j of the first non-zero coefficient of M_{i+1} is ahead of the one of M_i so that the coefficient c of \ominus is equal to the j -th coefficient of M_{i+1} minus zero. Since the (\curvearrowright) rule adds children to respect decreasing words, the p extra coefficients are added to the right of the j -th coefficient (including it) and therefore they are kept unchanged by the \ominus operation. Eventually, the sum over M_{i+1} is $p + 1$ and so is the sum over \hat{M}_{i+1} .
- (\nearrow) $C_\psi(v_{i+1})$ is built from $C_\psi(v_i)$ with [Algorithm 13](#), and therefore they (i) share a common prefix, possibly empty and (ii) then differ by a single letter. The index of that letter in M_{i+1} corresponds to the index j defined in \ominus . Therefore, the coefficient c is – before any (\curvearrowright) – equal to one. The argument of (\curvearrowright) letters being added to the right of j still holds and therefore the sum over \hat{M}_{i+1} is also $p + 1$.

To conclude the proof, we have to exhibit the inverse function of the mapping we just defined. This will prove that this mapping is indeed a bijection, and then the theorem holds.

B.3 Incremental adjacency matrix to reduced adjacency matrix

Let M and \hat{M} be constructed as before. From \hat{M} , we can define a matrix M' as:

$$\begin{cases} M'_1 = \hat{M}_1 \\ M'_{i+1} = M'_i \oplus \hat{M}_{i+1} \end{cases}$$

where the \oplus operation is defined as follows: given two words $a_0 \cdots a_n$ and $b_0 \cdots b_n$, then denoting $j = \min\{i : b_i \neq 0\}$, and $c = a_j + b_j$,

$$\begin{array}{r} a_0 \cdots a_{j-1} \quad a_j \quad a_{j+1} \cdots a_n \\ \oplus b_0 \cdots b_{j-1} \quad b_j \quad b_{j+1} \cdots b_n \\ \hline = a_0 \cdots a_{j-1} \quad c \quad b_{j+1} \cdots b_n \end{array} .$$

By construction, \oplus is the inverse operation of \ominus , so that we have the following lemma:

Lemma B.1 *The following properties hold:*

- ▶ $M_i \oplus (M_{i+1} \ominus M_i) = M_{i+1}$
- ▶ $(M_i \oplus \hat{M}_{i+1}) \ominus M_i = \hat{M}_{i+1}$

Therefore, $M = M'$.

The FDAG of Figure 5.7 is reproduced below to illustrate the stages of the proof. This FDAG is constructed in 7 steps, that are (in this order): (\uparrow) , (\nearrow) , (\nwarrow) , (\uparrow) , (\nearrow) , (\ominus) and (\ominus) . The matrices A , M and \hat{M} are given in Figure B.1. One can see that \hat{M} is of size 7, as expected.

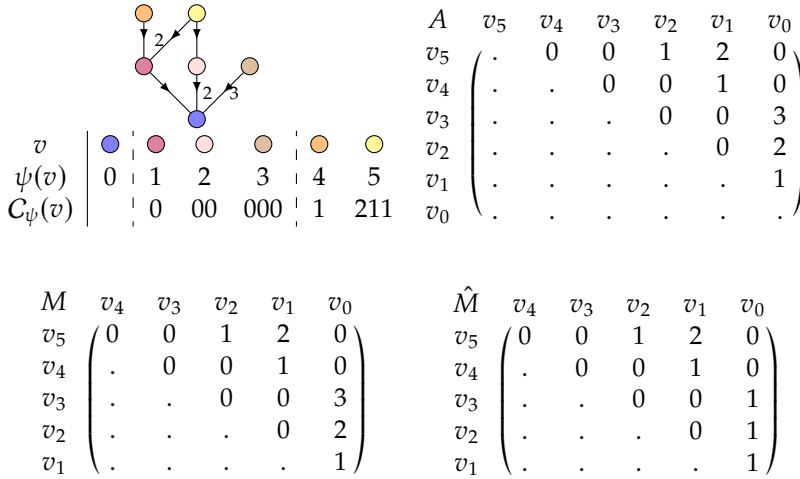


Figure B.1: The FDAG of Figure 5.7 reproduced (top left), its adjacency matrix A (top right), its reduced adjacency matrix M (bottom left) and its incremental adjacency matrix \hat{M} (bottom right). Dots represent zeros corresponding to $A_{i,j}$ elements with $i \leq j$.

Remark B.1 It should be noted that (general) Fishburn matrices, with at least one non-zero entry on each row and column, are in bijection with FDAGs compressing forests made of a unique tree. Indeed, via the bijection above, as such FDAG has a unique root, it must be the last inserted vertex, and therefore, each column admits at least one non-zero entry (otherwise it would be another root).

Remark B.2 It is possible to enumerate row-Fishburn matrices by using the previous bijection and the FDAGs enumeration tree together. Nevertheless, things are a little simpler in this case and the equivalent of the operations (\uparrow) and (\nearrow) can be merged, giving two rules for matrix expansion:

- (R1) Increase one coefficient to the (inclusive) right of the rightmost nonzero coefficient of the top row by 1.
- (R2) Increase the dimension of the matrix by 1 (to the left and top), all new coefficients set to zero. Set one coefficient of the top row to 1.

Bibliography

Here are the references in citation order.

- [1] Stanley Fields and Mark Johnston. 'Whither model organism research?' In: *Science* 307.5717 (2005), pp. 1885–1886 (cited on page ix).
- [2] Fabrice Besnard et al. 'Cytokinin signalling inhibitory fields provide robustness to phyllotaxis'. In: *Nature* 505.7483 (2014), pp. 417–421 (cited on page x).
- [3] Yann Guédon et al. 'Pattern identification and characterization reveal permutations of organs as a key genetically controlled property of post-meristematic phyllotaxis'. In: *Journal of theoretical biology* 338 (2013), pp. 94–110 (cited on page x).
- [4] Ayan Chaudhury and Christophe Godin. 'Skeletonization of plant point cloud data using stochastic optimization framework'. In: *Frontiers in Plant Science* 11 (2020), p. 773 (cited on page x).
- [5] Katia Mirande, Franck Hétyroy-wheeler, and Christophe Godin. 'High-precision 3D segmentation of plants combining spectral clustering and quotient graph techniques: a multi-level approach'. In: *9th International Conference on Functional-Structural Plant Models*. En ligne, France, 2020 (cited on page x).
- [6] Aristid Lindenmayer. 'Mathematical models for cellular interactions in development I. Filaments with one-sided inputs'. In: *Journal of theoretical biology* 18.3 (1968), pp. 280–299 (cited on page xi).
- [7] Aristid Lindenmayer. 'Developmental systems without cellular interactions, their languages and grammars'. In: *Journal of Theoretical Biology* 30.3 (1971), pp. 455–484 (cited on page xi).
- [8] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012 (cited on page xi).
- [9] Frédéric Boudon et al. 'L-Py: an L-system simulation framework for modeling plant architecture development based on a dynamic language'. In: *Frontiers in plant science* 3 (2012), p. 76 (cited on page xi).
- [10] Helin Dutagaci et al. 'ROSE-X: an annotated data set for evaluation of 3D plant organ segmentation methods'. In: *Plant methods* 16.1 (2020), pp. 1–14 (cited on page xi).
- [11] Ayan Chaudhury et al. 'Transferring PointNet++ Segmentation from Virtual to Real Plants'. In: *CVPPA-ICCV*. 2021 (cited on page xi).
- [12] Ayan Chaudhury, Frédéric Boudon, and Christophe Godin. '3D plant phenotyping: All you need is labelled point cloud data'. In: *European Conference on Computer Vision*. Springer. 2020, pp. 244–260 (cited on page xi).
- [13] Shu-Yun Le, Ruth Nussinov, and Jacob V. Maizel. 'Tree graphs of RNA secondary structures and their comparisons'. In: *Computers and Biomedical Research* 22.5 (1989), pp. 461–473. doi: [https://doi.org/10.1016/0010-4809\(89\)90039-6](https://doi.org/10.1016/0010-4809(89)90039-6) (cited on page 2).
- [14] Gianni Costa et al. 'A Tree-Based Approach to Clustering XML Documents by Structure'. In: *Knowledge Discovery in Databases: PKDD 2004*. Ed. by Jean-François Boulicaut et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 137–148 (cited on page 2).
- [15] M. A. Martín-Delgado, J. Rodríguez-Laguna, and G. Sierra. 'Density-matrix renormalization-group study of excitons in dendrimers'. In: *Phys. Rev. B* 65 (15 2002), p. 155116. doi: [10.1103/PhysRevB.65.155116](https://doi.org/10.1103/PhysRevB.65.155116) (cited on page 2).
- [16] Donald Meagher. 'Geometric modeling using octree encoding'. In: *Computer graphics and image processing* 19.2 (1982), pp. 129–147 (cited on page 2).
- [17] Leo Breiman et al. *Classification and regression trees*. Routledge, 2017 (cited on page 2).
- [18] Leo Breiman. 'Random forests'. In: *Machine learning* 45.1 (2001), pp. 5–32 (cited on page 2).

- [19] Marek Kubale. *Graph colorings*. Vol. 352. American Mathematical Soc., 2004 (cited on page 2).
- [20] Radia Perlman. ‘An algorithm for distributed computation of a spanning tree in an extended LAN’. In: *ACM SIGCOMM Computer Communication Review* 15.4 (1985), pp. 44–53 (cited on page 2).
- [21] Marthe Bonamy. ‘A small report on graph and tree isomorphism’. In: *Lecture note* (2010) (cited on pages 3, 13).
- [22] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. ‘The design and analysis of computer algorithms’. In: *Reading* (1974) (cited on pages 3, 13, 119).
- [23] Pierre-Antoine Champin and Christine Solnon. ‘Measuring the similarity of labeled graphs’. In: *International Conference on Case-Based Reasoning*. Springer. 2003, pp. 80–95 (cited on pages 3, 13, 25).
- [24] Kellogg S Booth and Charles J Colbourn. *Problems polynomially equivalent to graph isomorphism*. Computer Science Department, Univ., 1979 (cited on pages 3, 25, 37, 38).
- [25] Brendan D McKay and Adolfo Piperno. ‘Practical graph isomorphism, II’. In: *Journal of symbolic computation* 60 (2014), pp. 94–112 (cited on pages 3, 25, 38, 121, 127).
- [26] Florian Ingels and Romain Azaïs. ‘Isomorphic Unordered Labeled Trees up to Substitution Ciphering’. In: *Combinatorial Algorithms*. Ed. by Paola Flocchini and Lucia Moura. Cham: Springer International Publishing, 2021, pp. 385–399 (cited on pages 3, 6, 23, 37, 134).
- [27] Tatsuya Asai et al. ‘Discovering frequent substructures in large unordered trees’. In: *International Conference on Discovery Science*. Springer. 2003, pp. 47–61 (cited on pages 4, 64, 65).
- [28] David Avis and Komei Fukuda. ‘Reverse search for enumeration’. In: *Discrete applied mathematics* 65.1-3 (1996), pp. 21–46 (cited on pages 4, 60, 68).
- [29] Shin-ichi Nakano. ‘Efficient generation of plane trees’. In: *Information Processing Letters* 84.3 (2002), pp. 167–172 (cited on pages 4, 63).
- [30] Shin-ichi Nakano and Takeaki Uno. ‘Efficient generation of rooted trees’. In: *National Institute for Informatics (Japan), Tech. Rep. NII-2003-005E* 8 (2003) (cited on pages 4, 63–65, 67, 84, 85).
- [31] Florian Ingels and Romain Azaïs. ‘A Reverse Search Method for the Enumeration of Unordered Forests using DAG Compression’. In: *Fourth International Workshop on Enumeration Problems and Applications*. 2020 (cited on page 4).
- [32] Florian Ingels and Romain Azaïs. ‘Enumeration of Irredundant Forests’. In: *Theoretical Computer Science* (2022). doi: <https://doi.org/10.1016/j.tcs.2022.04.033> (cited on pages 4, 6, 59, 72, 147).
- [33] Nello Cristianini, John Shawe-Taylor, et al. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000 (cited on pages 5, 94, 103).
- [34] David Haussler. *Convolution kernels on discrete structures*. Tech. rep. Technical report, Department of Computer Science, University of California . . . , 1999 (cited on pages 5, 95, 129).
- [35] Michael Collins and Nigel Duffy. ‘Convolution kernels for natural language’. In: *Advances in neural information processing systems* 14 (2001) (cited on pages 5, 85, 95, 99, 102, 129).
- [36] Giovanni Da San Martino. ‘Kernel methods for tree structured data’. In: (2009) (cited on pages 5, 95, 99, 102, 107, 129).
- [37] SVN Vishwanathan, Alexander Johannes Smola, et al. ‘Fast kernels for string and tree matching’. In: *Kernel methods in computational biology* 15 (2004), pp. 113–130 (cited on pages 5, 85, 95, 99, 101, 104, 110, 117).
- [38] Florian Ingels and Romain Azaïs. ‘De l’importance de la fonction de poids dans le noyau des sous-arbres’. In: *JdS 2019-51èmes Journées de Statistique*. 2019, pp. 1–6 (cited on page 5).
- [39] Romain Azaïs and Florian Ingels. ‘The weight function in the subtree kernel is decisive’. In: *Journal of Machine Learning Research* 21 (2020), pp. 1–36 (cited on pages 5, 6, 17, 93, 101, 133, 144).
- [40] Arthur Cayley. *On the Analytical Forms Called Trees: With Application to the Theory of Chemical Combinations*. se, 1875 (cited on page 5).

- [41] Philip Bille et al. ‘Tree compression with top trees’. In: *Information and Computation* 243 (2015), pp. 166–177 (cited on page 6).
- [42] Ivan E Sutherland. ‘Sketchpad a man-machine graphical communication system’. In: *Simulation* 2.5 (1964), R–3 (cited on pages 6, 14).
- [43] John C Hart and Thomas A DeFanti. ‘Efficient antialiased rendering of 3-D linear fractals’. In: *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*. 1991, pp. 91–100 (cited on pages 6, 14).
- [44] Peter Buneman, Martin Grohe, and Christoph Koch. ‘Path queries on compressed XML’. In: *Proceedings 2003 VLDB Conference*. Elsevier. 2003, pp. 141–152 (cited on pages 6, 14).
- [45] Markus Frick, Martin Grohe, and Christoph Koch. ‘Query evaluation on compressed trees’. In: *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings*. IEEE. 2003, pp. 188–197 (cited on pages 6, 14).
- [46] Romain Azaïs et al. ‘Treex: a Python package for manipulating rooted trees’. In: *Journal of Open Source Software* 4.38 (2019), p. 1351 (cited on page 7).
- [47] Edward A Bender and S Gill Williamson. *Lists, decisions and graphs*. S. Gill Williamson, 2010 (cited on page 11).
- [48] Heinz Prüfer. ‘Neuer beweis eines satzes über permutationen’. In: *Arch. Math. Phys* 27.1918 (1918), pp. 742–744 (cited on page 11).
- [49] Donald E Knuth. *Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees—History of Combinatorial Generation*. Addison-Wesley Professional, 2013 (cited on page 11).
- [50] David Aldous. ‘The continuum random tree III’. In: *The Annals of Probability* (1993), pp. 248–289 (cited on page 11).
- [51] Romain Azaïs, Alexandre Genadot, and Benoit Henry. ‘Inference for conditioned Galton-Watson trees from their Harris path’. In: *ALEA* 16 (2019), pp. 561–604 (cited on pages 11, 12).
- [52] Béla Bollobás. *Random Graphs*. 2nd ed. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2001 (cited on page 12).
- [53] Jacques Neveu. ‘Arbres et processus de Galton-Watson’. In: *Annales de l’IHP Probabilités et statistiques*. Vol. 22. 2. 1986, pp. 199–207 (cited on page 12).
- [54] Albert Nijenhuis and Herbert S Wilf. *Combinatorial algorithms: for computers and calculators*. Elsevier, 2014 (cited on page 12).
- [55] L Alonso, R Schott, and INRIA-Lorraine CRIN. ‘Random Unlabelled Rooted Trees Revisited’. In: *Proc. ICCI*. Vol. 94. Citeseer. 1994, pp. 1352–1367 (cited on page 12).
- [56] Yazhe Zhang. ‘On the number of leaves in a random recursive tree’. In: *Brazilian Journal of Probability and Statistics* 29.4 (2015), pp. 897–908 (cited on page 12).
- [57] Gabriel Valiente. *Algorithms on trees and graphs*. Springer Science & Business Media, 2013 (cited on page 12).
- [58] Douglas M Campbell and David Radford. ‘Tree isomorphism algorithms: Speed vs. clarity’. In: *Mathematics Magazine* 64.4 (1991), pp. 252–261 (cited on page 13).
- [59] Donald E Knuth. ‘The art of computer programming. Vol. 3, Sorting and Searching’. In: (1973) (cited on pages 14, 136).
- [60] Steven S Skiena. ‘Sorting and searching’. In: *The Algorithm Design Manual*. Springer, 2012, pp. 103–144 (cited on pages 14, 133, 140).
- [61] Christophe Godin and Pascal Ferraro. ‘Quantifying the degree of self-nestedness of trees: application to the structural analysis of plants’. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 7.4 (2009), pp. 688–703 (cited on pages 15, 16, 83).
- [62] Peter J Downey, Ravi Sethi, and Robert Endre Tarjan. ‘Variations on the common subexpression problem’. In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 758–771 (cited on page 16).

- [63] Martin Gardner. *Codes, ciphers and secret writing*. Courier Corporation, 1984 (cited on page 23).
- [64] Mark S Mayzner and Margaret Elizabeth Tresselt. 'Tables of single-letter and digram frequency counts for various word-length and letter-position combinations.' In: *Psychonomic monograph supplements* (1965) (cited on page 23).
- [65] Chris Savarese and Brian Hart. 'The Caesar Cipher'. In: *Historical Cryptography Web Site* (1999) (cited on page 23).
- [66] Viktor N Zemlyachenko, Nickolay M Korneenko, and Regina I Tyshkevich. 'Graph isomorphism problem'. In: *Journal of Soviet Mathematics* 29.4 (1985), pp. 1426–1481 (cited on pages 25, 127).
- [67] Uwe Schöning. 'Graph isomorphism is in the low hierarchy'. In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 1987, pp. 114–124 (cited on pages 25, 38, 119, 127).
- [68] Boris Weisfeiler and Andrei Leman. 'The reduction of a graph to canonical form and the algebra which appears therein'. In: *NTI, Series 2.9* (1968), pp. 12–16 (cited on pages 25, 120, 121).
- [69] Martin Grohe, Pascal Schweitzer, and Daniel Wiebking. 'Deep Weisfeiler Leman'. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 2600–2614 (cited on page 25).
- [70] László Babai. 'Monte-Carlo algorithms in graph isomorphism testing'. In: *Université tde Montréal Technical Report, DMS 79-10* (1979) (cited on page 38).
- [71] Maria M Klawe, Derek G Corneil, and Andrzej Proskurowski. 'Isomorphism testing in hookup classes'. In: *SIAM Journal on Algebraic Discrete Methods* 3.2 (1982), pp. 260–274 (cited on page 38).
- [72] Stefan Canzar et al. 'On tree-constrained matchings and generalizations'. In: *Algorithmica* 71.1 (2015), pp. 98–119 (cited on page 38).
- [73] Monaldo Mastrolilli and Georgios Stamoulis. 'Constrained matching problems in bipartite graphs'. In: *International Symposium on Combinatorial Optimization*. Springer. 2012, pp. 344–355 (cited on page 38).
- [74] Rina Dechter and Daniel Frost. 'Backtracking algorithms for constraint satisfaction problems; a survey'. In: *Constraints, International Journal, to appear* (1998) (cited on page 39).
- [75] Donald E Knuth. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming)*. Addison-Wesley Professional, 2005 (cited on pages 51, 137).
- [76] Ailsa H Land and Alison G Doig. 'An automatic method for solving discrete programming problems'. In: *50 Years of Integer Programming 1958-2008*. Springer, 2010, pp. 105–132 (cited on page 60).
- [77] Xifeng Yan and Jiawei Han. 'gspan: Graph-based substructure pattern mining'. In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings*. IEEE. 2002, pp. 721–724 (cited on pages 60, 62).
- [78] Kazuaki Yamazaki et al. 'Enumeration of nonisomorphic interval graphs and nonisomorphic permutation graphs'. In: *Theoretical Computer Science* 806 (2020), pp. 310–322 (cited on page 60).
- [79] Sebastian Nowozin. 'Learning with structured data: applications to computer vision.' PhD thesis. Berlin Institute of Technology, 2009 (cited on pages 60, 62).
- [80] David S Johnson, Mihalis Yannakakis, and Christos H Papadimitriou. 'On generating all maximal independent sets'. In: *Information Processing Letters* 27.3 (1988), pp. 119–123 (cited on pages 60, 64, 78, 81).
- [81] Giuseppe Di Battista and Roberto Tamassia. 'Algorithms for plane representations of acyclic digraphs'. In: *Theoretical Computer Science* 61.2-3 (1988), pp. 175–198 (cited on page 61).
- [82] Ralph Freese. 'Automated lattice drawing'. In: *International Conference on Formal Concept Analysis*. Springer. 2004, pp. 112–127 (cited on page 61).
- [83] Katsuhisa Yamanaka. 'Permutation Enumeration'. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. New York, NY: Springer New York, 2016, pp. 1559–1564. doi: 10.1007/978-1-4939-2864-4_735 (cited on page 61).

- [84] Mukund Deshpande, Michihiro Kuramochi, and George Karypis. 'Mining Chemical Compounds'. In: *Data Mining in Bioinformatics*. Ed. by Xindong Wu et al. London: Springer London, 2005, pp. 189–215. doi: [10.1007/1-84628-059-1_9](https://doi.org/10.1007/1-84628-059-1_9) (cited on page 61).
- [85] Victor Parque and Tomoyuki Miyashita. 'An Efficient Scheme for the Generation of Ordered Trees in Constant Amortized Time'. In: *2021 15th International Conference on Ubiquitous Information Management and Communication (IMCOM)*. IEEE. 2021, pp. 1–8 (cited on page 63).
- [86] Benno Schwikowski and Ewald Speckenmeyer. 'On enumerating all minimal solutions of feedback problems'. In: *Discrete Applied Mathematics* 117.1-3 (2002), pp. 253–265 (cited on page 64).
- [87] Arthur B Kahn. 'Topological sorting of large networks'. In: *Communications of the ACM* 5.11 (1962), pp. 558–562 (cited on page 68).
- [88] Hsien-Kuei Hwang and Emma Yu Jin. 'Asymptotics and statistics on Fishburn matrices and their generalizations'. In: *arXiv preprint arXiv:1911.06690* (2019) (cited on page 78).
- [89] Vít Jelínek. 'Counting general and self-dual interval orders'. In: *Journal of Combinatorial Theory, Series A* 119.3 (2012), pp. 599–614 (cited on page 78).
- [90] Kathrin Bringmann, Yingkun Li, and Robert C Rhoades. 'Asymptotics for the number of row-Fishburn matrices'. In: *European Journal of Combinatorics* 41 (2014), pp. 183–196 (cited on page 78).
- [91] Charles H Jones. 'Generalized Hockey Stick Identities and N-dimensional Blockwalking'. In: (1994) (cited on page 83).
- [92] Raymond Greenlaw. 'Subtree isomorphism is in DLOG for nested trees'. In: *International Journal of Foundations of Computer Science* 7.02 (1996), pp. 161–167 (cited on page 83).
- [93] Romain Azaïs, Jean-Baptiste Durand, and Christophe Godin. 'Approximation of trees by self-nested trees'. In: *ALENEX 2019 - Algorithm Engineering and Experiments*. San Diego, United States, Jan. 2019, pp. 1–24. doi: [10.10860](https://doi.org/10.10860) (cited on pages 83, 84).
- [94] William Feller. *An introduction to probability theory and its applications, vol 2*. John Wiley & Sons, 2008 (cited on page 85).
- [95] Roger A Horn and Charles R Johnson. *Matrix analysis*. Cambridge university press, 2012 (cited on page 94).
- [96] James Mercer. 'XVI. Functions of positive and negative type, and their connection the theory of integral equations'. In: *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character* 209.441-458 (1909), pp. 415–446 (cited on page 94).
- [97] Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2001 (cited on pages 94, 103).
- [98] John Shawe-Taylor, Nello Cristianini, et al. *Kernel methods for pattern analysis*. Cambridge university press, 2004 (cited on page 94).
- [99] Trevor Hastie et al. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer, 2009 (cited on page 94).
- [100] Daisuke Kimura et al. 'A subpath kernel for rooted unordered trees'. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2011, pp. 62–74 (cited on pages 95, 99, 104).
- [101] Kilho Shin and Tetsuji Kuboyama. 'A Generalization of Haussler's Convolution Kernel—Mapping Kernel and Its Application to Tree Kernels'. In: *Journal of Computer Science and Technology* 25.5 (2010), pp. 1040–1054 (cited on page 95).
- [102] Fabio Aioli et al. 'Fast on-line kernel learning for trees'. In: *Sixth International Conference on Data Mining (ICDM'06)*. IEEE. 2006, pp. 787–791 (cited on pages 95, 99, 102, 107).
- [103] Maria-Florina Balcan, Avrim Blum, and Nathan Srebro. 'A theory of learning with similarity functions'. In: *Machine Learning* 72.1-2 (2008), pp. 89–112. doi: [10.1007/s10994-008-5059-5](https://doi.org/10.1007/s10994-008-5059-5) (cited on pages 97, 98).

- [104] David S. Ebert and F. Kenton Musgrave. *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003 (cited on page 105).
- [105] Ludovic Denoyer and Patrick Gallinari. 'Report on the XML mining track at INEX 2005 and INEX 2006: categorization and clustering of XML documents'. In: *SIGIR Forum*. Vol. 41. 2007, pp. 79–90 (cited on page 111).
- [106] Ghassan Hamarneh and Preet Jassi. 'VascuSynth: Simulating Vascular Trees for Generating Volumetric Image data with Ground Truth Segmentation and Tree Analysis'. In: *Computerized Medical Imaging and Graphics* 34.8 (2010), pp. 605–616. doi: 10.1016/j.compmedimag.2010.06.002 (cited on page 113).
- [107] Preet Jassi and Ghassan Hamarneh. 'VascuSynth: Vascular Tree Synthesis Software'. In: *Insight Journal* January-June (2011), pp. 1–12. doi: 10380/3260 (cited on page 113).
- [108] Damien G Hicks et al. 'Maps of variability in cell lineage trees'. In: *PLoS computational biology* 15.2 (2019), e1006745 (cited on page 114).
- [109] E Faure et al. 'An algorithmic workflow for the automated processing of 3D+ time microscopy images of developing organisms and the reconstruction of their cell lineage'. In: *Nat. Commun* (2015) (cited on page 114).
- [110] Martin Grohe et al. 'Color refinement and its applications'. In: (2017) (cited on pages 120, 121).
- [111] Cameron B Browne et al. 'A survey of Monte Carlo tree search methods'. In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43 (cited on page 121).
- [112] Frank Harary. 'The number of linear, directed, rooted, and connected graphs'. In: *Transactions of the American Mathematical Society* 78.2 (1955), pp. 445–463 (cited on page 125).
- [113] Alfred V Aho and Neil JA Sloane. 'Some doubly exponential sequences'. In: *Fibonacci Quart* 11.4 (1973), pp. 429–437 (cited on pages 129, 130).
- [114] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. 'Perfect hashing for network applications'. In: *2006 IEEE International Symposium on Information Theory*. IEEE. 2006, pp. 2774–2778 (cited on page 136).
- [115] Owen Astrachan. 'Bubble sort: an archaeological algorithmic analysis'. In: *ACM Sigcse Bulletin* 35.1 (2003), pp. 1–5 (cited on page 139).

Index of frequent notations

Miscellaneous

$\#G$	Number of vertices of graph G
$\#S$	Number of elements of set S
$O(f(n))$	Algorithm worst case time complexity is bounded by $f(n)$, up to a multiplicative constant – where n is the size of the input
\perp, \top	Respectively, logical FALSE and logical TRUE
id	Identity function
ϕ	Isomorphism between two graphs
$\{x : P(x)\}$	Set of elements x that satisfy property $P(x)$
$\mathbb{1}_{P(x)}$	1 if $P(x) = \top$, 0 otherwise
$\text{Isom}(G_1, G_2)$	Set of all isomorphisms between graphs G_1 and G_2
\wedge, \vee	Respectively, logical AND and logical OR
E/R	Quotient set of set E by the equivalence relation R
$G_1 \simeq G_2$	G_1 and G_2 are isomorphic

Nodes (trees & DAGs)

$[v]$	Class of equivalence of node v for the tree isomorphism equivalence relation
$C(v)$	Children of node v

$\text{depth}(v)$	Depth of node v
$\mathcal{D}(v)$	Descendants of node v – i.e., children of v , their children, and so on
$\text{o}(v)$	Origin of node v (in a DAG D compressing a forest); the set of trees in the forest in which $\mathfrak{R}^{-1}(D[v])$ appears as a subtree
$\text{deg}(v)$	Degree of node v
\bar{v}	Label of node v
$\mathcal{P}(v)$	Parent(s) of node v
$\pi_i(v)$	Number of times the subtree $\mathfrak{R}^{-1}(D[v])$ appears in the tree T_i , where D is the DAG compression of the forest containing T_i
$\mathcal{H}(v)$	Height of node v
$T[v]$	Subtree of T rooted in node $v \in T$

Trees

$\mathcal{A}(T)$	Set of labels of tree T – also called alphabet
$\mathcal{L}(T)$	Leaves of tree T
$\mathfrak{R}(T)$	DAG compression of tree T , \mathfrak{R}^{-1} stands for the inverse operator
$\mathcal{S}(T)$	Set of subtrees of tree T
$\mathcal{R}(T), \mathcal{R}(D)$	Root of tree T ; set of roots of DAG D
\mathcal{T}	Set of trees
T^h	Set of nodes of T with height h