



**HAL**  
open science

# Revisiting Tree Isomorphism: An Algorithmic Bric-à-Brac

Florian Ingels

► **To cite this version:**

| Florian Ingels. Revisiting Tree Isomorphism: An Algorithmic Bric-à-Brac. 2023. hal-04232137v2

**HAL Id: hal-04232137**

**<https://hal.science/hal-04232137v2>**

Preprint submitted on 13 Feb 2024

**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.

# Revisiting Tree Isomorphism: An Algorithmic Bric-à-Brac

Florian Ingels

`florian.ingels@univ-lille.fr`

Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille, France

## Abstract

The Aho, Hopcroft and Ullman (AHU) algorithm has been the state of the art since the 1970s for determining in linear time whether two unordered rooted trees are isomorphic or not. However, it has been criticized (by Campbell and Radford) for the way it is written, which requires several (re)readings to be understood, and does not facilitate its analysis. In this article, we propose a different, more intuitive formulation of the algorithm, as well as three propositions of implementation, two using sorting algorithms and one using prime multiplication. Although none of these three variants admits linear complexity, we show that in practice two of them are competitive with the original algorithm, while being straightforward to implement. Surprisingly, the algorithm that uses multiplications of prime numbers (which are also generated during the execution) is competitive with the fastest variants using sorts, despite having a worst theoretical complexity. We also adapt our formulation of AHU to tackle to compression of trees in directed acyclic graphs (DAGs). This algorithm is also available in three versions, two with sorting and one with prime number multiplication. Our experiments are carried out on trees of size at most  $10^6$ , consistent with the actual datasets we are aware of, and done in Python with the library `treex`, dedicated to tree algorithms.

**Keywords:** tree isomorphism, AHU algorithm, prime numbers multiplication, DAG compression

## 1 Introduction

### 1.1 Context

The Aho, Hopcroft and Ullman (AHU) algorithm, introduced in the 1970s [1, Example 3.2], establishes that the tree isomorphism problem can be solved in linear time, whereas the more general graph isomorphism problem is still an open problem today, where no proof of NP-completeness nor polynomial algorithm is known [41]. However, the problem is considered to be solved in practice; powerful heuristics exist, such as the quasi-polynomial algorithm from [10]; see also [36].

As far as we know, AHU remains the only state-of-the-art algorithm for determining, in practice, whether two trees are isomorphic. Recently, Liu [34] proposed to represent a tree by a polynomial of two variables, computable in linear time, and where two trees have the same polynomial if and only if they are isomorphic. Unfortunately, the existence of an algorithm to determine the equality of two polynomials in polynomial time is still an open question [40]. We should also mention [19], which proposes an alternating logarithmic time algorithm for tree isomorphism – under NC complexity class framework, that is, problems efficiently solvable on a parallel computer [11].

However, one criticism – emerging from Campbell and Radford in [20] – directed at the AHU algorithm is that it is presented in such a way that it is difficult to understand. We leave it to the reader to form their own opinion by reproducing the original text of the algorithm in Section 1.3, after a brief introduction of key background in Section 1.2. To the best of our knowledge, the remark from Campbell and Radford seems to have remained a dead letter in the community, and no alternative, clearer version of the algorithm seems ever to have been published – with the exception of Campbell and Radford themselves, which have nevertheless remained fairly close to the original text.

In this article, we propose to revisit the AHU algorithm by giving several alternative versions, all of them easier to understand and straightforward to implement. However, these variants have supra-linear complexity (which is also the case for the Campbell and Radford version). In practice, on trees of reasonable size ( $\leq 10^6$ ), with a Python implementation using the `treex` library [9], we find that two of the three proposed variants are faster than the original algorithm – one of them sorts lists of integers (like the original algorithm), while the other replaces this step by calculating the product of a list of primes. We also propose a direct adaptation of our variants to compute tree compression into directed acyclic graphs (DAGs) [25] – this time achieving state of the art complexity.

Section 1.2 introduces the notations and definitions useful for the rest of the paper; the original AHU algorithm is presented in Section 1.3, as well as the aim of the paper.

## 1.2 Tree isomorphisms

A rooted tree  $T$  is a connected directed graph without any undirected cycle such that (i) there exists a special node called the root and (ii) any node but the root has exactly one parent. The parent of a node  $u$  is denoted by  $\mathcal{P}(u)$ , whereas its children are denoted by  $\mathcal{C}(u)$ . The leaves  $\mathcal{L}(T)$  of  $T$  are the nodes without any children. Rooted trees are said to be unordered if the order among siblings is not significant; otherwise they are said to be ordered. This paper focuses only on unordered rooted trees, referred to simply as *trees* in the remainder of this article.

The degree of a node  $u$  is defined as  $\deg(u) = \#\mathcal{C}(u)$  and the degree of a tree  $T$  as  $\deg(T) = \max_{u \in T} \deg(u)$ . The depth  $\mathcal{D}(u)$  of a node  $u$  is the length of the path between  $u$  and the root. The depth  $\mathcal{D}(T)$  of  $T$  is the maximal depth among all nodes. The level of a node  $u$  is defined as  $\mathcal{D}(T) - \mathcal{D}(u)$ . The sets of nodes of level  $d$  in a tree  $T$  is denoted by  $T^d$ , and the mapping  $d \mapsto T^d$  can be constructed in linear time by a simple traversal of  $T$ .

**Definition 1.** *Two trees  $T_1$  and  $T_2$  are said to be isomorphic if there exists a bijective mapping  $\varphi : T_1 \rightarrow T_2$  so that (i) the roots are mapped together and (ii) for any  $u, v \in T_1, v \in \mathcal{C}(u) \iff \varphi(v) \in \mathcal{C}(\varphi(u))$ .*

Such a mapping  $\varphi$  is called a *tree isomorphism*. An example of isomorphic trees is provided in Figure 1. Whenever two trees  $T_1$  and  $T_2$  are isomorphic, we note  $T_1 \simeq T_2$ . It is well known that  $\simeq$  is an equivalence relation on the set of trees [46]. The *tree isomorphism problem* consists in deciding whether two trees are isomorphic or not.

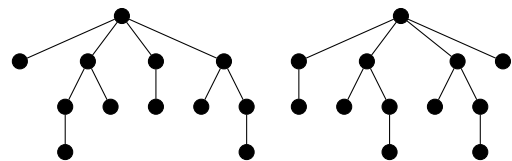


Figure 1: Two isomorphic trees.

For the broader *graph* isomorphism problem, it is not usual to explicitly construct the isomorphism  $\varphi$  – let us mention nonetheless [23, Section 3.3] and [30, 7] – but rather to compute a certificate of non-isomorphism. For instance, Weisfeiler-Lehman algorithms, also known as colour refinement algorithms [29, 32], colour the nodes of each graph according to certain rules, and the histograms of the colour distributions are then compared: if they diverge, the graphs are not isomorphic. This test is not complete in the sense that there are non-isomorphic graphs with the same colour histogram – even though the distinguishing power of these algorithms is constantly being improved [26]. While the graph isomorphism problem is not solved in the general case, it is solved for trees by virtue of the AHU algorithm, which is built on a colouring principle similar to that of Weisfeiler-Lehman.


### 1.3 The Aho, Hopcroft and Ullman algorithm

We reproduce below the original text of the algorithm, as introduced in 1974 by Aho, Hopcroft and Ullman in [1, Example 3.2] – only minor changes have been made to fit the notations used in this paper.

1. First, assign to all leaves in  $T_1$  and  $T_2$  the integer 0.
2. Assume by induction that all nodes at level  $d - 1$  of  $T_1$  and  $T_2$  have been assigned an integer. Let  $L_1$  (respectively  $L_2$ ) be the list of nodes in  $T_1$  (respectively  $T_2$ ) at level  $d - 1$  sorted by non-decreasing value of the assigned integers.
3. Assign to the nonleaves of  $T_1$  at level  $d$  a tuple of integers by scanning the list  $L_1$  from left to right and performing the following actions:
  - For each vertex on list  $L_1$  take the integer assigned to  $u$  to be the next component of the tuple associated with  $\mathcal{P}(u)$ .
  - On completion of this step, each nonleaf  $w$  of  $T_1$  at level  $d$  will have a tuple  $(i_1, i_2, \dots, i_k)$  associated with it, where  $i_1, \dots, i_k$  are the integers, in non-decreasing order, associated with the children of  $w$ .
  - Let  $S_1$  be the sequence of tuples created for the vertices of  $T_1$  on level  $d$ .
4. Repeat Step 3 for  $T_2$  and let  $S_2$  be the sequence of tuples created for the vertices of  $T_2$  on level  $d$ .
5. Sort  $S_1$  and  $S_2$  lexicographically. Let  $S'_1$  and  $S'_2$ , respectively, be the sorted sequence of tuples.
6. If  $S'_1$  and  $S'_2$  are not identical, then halt: the trees are not isomorphic. Otherwise, assign the integer 1 to those vertices of  $T_1$  on level  $d$  represented by the first distinct tuple on  $S'_1$ , assign the integer 2 to the vertices represented by the second distinct tuple, and so on. As these integers are assigned to the vertices of  $T_1$  on level  $d$ , replace  $L_1$  by the list of the vertices so assigned. Append the leaves of  $T_1$  on level  $d$  to the front of  $L_1$ . Do the same for  $L_2$ .  $L_1$  and  $L_2$  can now be used for the assignment of tuples to nodes at level  $d + 1$  by returning to Step 3.
7. If the roots of  $T_1$  and  $T_2$  are assigned the same integer,  $T_1$  and  $T_2$  are isomorphic.

Note that, in Step 5, the authors resort to a variant of radix sort [1, Algorithm 3.2]. Actually, the tree isomorphism problem and AHU algorithm are only introduced in the book as an application example of this sorting algorithm. This algorithm can sort  $n$  lists of varying lengths  $l_1, \dots, l_n$ , containing integers between 0 and  $m - 1$ , in complexity  $O(\sum_{i=1}^n l_i + m)$ . Expressed within our framework, the length of each list is exactly the degree of the associated node, and  $m = \#\{c(u) : u \in T^d\}$  – where  $c(u)$  designates the integer associated to node  $u$ .

**Theorem 1** (Aho, Hopcroft & Ullman). *AHU algorithm runs in  $O(n)$  where  $n = \#T_1 = \#T_2$ .*

**Proof.** See the proofs in [1, Example 3.2] for the whole algorithm and especially [1, Algorithm 3.2] for sorting lists  $S_1$  and  $S_2$  in Step 5. As stated above, Step 5 has complexity  $O(\sum_{u \in T^d} \deg(u) + \#\{c(u) : u \in T^d\})$ . Noticing that  $\sum_{u \in T^d} \deg(u) = \#T^{d-1}$  and that  $\#\{c(u) : u \in T^d\} \leq \#T^d$ , and summing over all levels indeed yields a linear complexity for all those sorts. 

**Remark 1.** *A point (very) briefly addressed by the authors of AHU algorithm specifies that the maximum integer  $m$  used in Step 5 must be not “too large” [1, Section 3.2, p.77]. Indeed, the sorting algorithm works if the integers can actually be considered as integers, and not as sequences of 0’s and 1’s, as pointed out by Radford and Campbell [20] – in which they show that there are large trees for which the algorithms runs in  $O(n \log n)$ .*

*How large are we talking? For the integers to **not** fit on one word of memory, we must assume that  $m > 2^k$  with a  $k$ -bit machine. The smallest tree  $T$  with  $m = 2^k$  has roughly  $k2^k$  nodes (see Appendix A). With  $k = 64$ , this would imply trees of size  $\approx 10^{21}$ . For most practical applications, this is unlikely to be a problem.*

Without any additional context for interpreting what the algorithm does, perhaps the reader will agree with this comment, arguing that the formulation of the algorithm is

*utterly opaque. Even on second or third reading. When an algorithm is written it should be clear, it should persuade, and it should lend itself to analysis.*

— Douglas M. Campbell and David Radford [20]

In Campbell and Radford view, the formulation of the algorithm is detrimental to understanding it, analyse it, and implement it. It is true that the theoretical contribution of the AHU algorithm is indisputable, since it establishes that the tree isomorphism problem is linear. On the other hand, and to support Campbell and Radford’s point of view, AHU would benefit from a formulation that is simpler to understand and implement. From a pedagogical point of view, it is likely that AHU is one of the first algorithms that people might want to study or implement when they learn about tree graph theory. AHU algorithm is also used by more advanced algorithms, such as the compression of trees into directed acyclic graphs (DAG) – see Section 2.3; or routinely for the construction of marked tree isomorphisms in [30, 7].

**Aim of the paper** In [20], Campbell and Radford provide a very clear, step-by-step exposition of the intuitions that lead to the AHU algorithm, and they even provide an algorithm similar to AHU that associates bitstrings to nodes instead of integers – with  $O(n \log n)$  time complexity. In this paper, we introduce yet another formulation for the AHU algorithm. This formulation assigns integers to the nodes, as does AHU and unlike Campbell and Radford’s version. Several possible implementations of our approach are studied, both from a theoretical and a practical point of view. In addition to clarifying the intuition behind the original algorithm, our variants are straightforward to implement – at the cost, however, of worse complexity. With respect to Remark 1, when used with trees of reasonable size in line with common use cases, they nonetheless perform better in practice. The outline of the paper is as follows:

- Section 2 introduces our intuition for the AHU algorithm, in three variants: two using list sorts, and one using multiplication of lists of primes instead. We also present an adaptation of AHU for the compression of trees into directed acyclic graphs (DAG), also in three variants.
- Section 3 tests these algorithms on simulated data of reasonable size, in competition with the original algorithm whenever possible (i.e. excluding DAG compression).

Although the theoretical complexities of the algorithms presented here exceed the linear complexity of the original algorithm, we show that in practice, with the exception of one of the sorting variants, the others are competitive with the original. In particular, the variant using prime number multiplication is competitive with the best variant using sorts, even though it also has to generate the primes on the fly in addition to multiplying them.

Finally, in Appendix A, we study a very specific class of trees, which allows us to prove results – namely Lemmas 1 and 2 – relating the size of trees to the number of distinct integers needed to assign classes level by level in the AHU algorithm (whatever its variant). To the best of our knowledge, this matter has never been addressed before.

## 2 Revisiting AHU algorithm

In this section, we present variants of the original AHU algorithm. First, Section 2.1 provides a new intuition of the algorithm, in the form of a colouring process. We propose two variant implementations, each using

a different sorting algorithm. In Section 2.2, we replace the sorting step by the multiplication of a list of primes, leading to a new variant of AHU. Finally, in Section 2.3, we focus on the compression of trees into directed acyclic graphs (DAGs), which can be achieved via a simple modification of our version of AHU – declined in three implementations: two with sorting, one with prime multiplication, as before.

## 2.1 An intuition for AHU and two variants

As already stated, the interested reader can find in [20] a step-by-step explanation of the concepts that work behind AHU algorithm. Here, we introduce another intuition for the AHU algorithm, presented as a colouring process, thus making the connection with Weisfeiler-Lehman algorithms for graph isomorphism already mentioned.

The core idea behind AHU algorithm is to provide each node in trees  $T_1$  and  $T_2$  a canonical representative of its equivalence class for  $\simeq$ , thus containing all the information about its descendants.

The nodes of both trees are simultaneously browsed in ascending levels. Suppose that each node  $u$  of level  $d - 1$  has been assigned a colour  $c(u)$ , representing its equivalence class for the relation  $\simeq$ . Each node  $u$  of level  $d$  is associated with a multiset  $\mathcal{C}_c(u) = \{\{c(v) : v \in \mathcal{C}(u)\}\}$  – if  $u$  is a leaf, this multiset is denoted  $\emptyset$ . Each distinct multiset is given a colour, which is assigned to the corresponding nodes. An illustration is provided in Figure 2. In the end, the trees are isomorphic if and only if their roots receive the same colour. Moreover, after processing level  $d$ , if the multiset of colours assigned to the nodes of level  $d$  differs from one tree to the other, we can immediately conclude that the trees are not isomorphic.

Consider the number of colours required by any version of AHU algorithm; this number is given by

$$\max_{d \in \llbracket 0, \mathcal{D}(T) \rrbracket} \#\{c(u) : u \in T^d\}. \quad (1)$$

We call it the *width* of  $T$  and denote it by  $\mathcal{W}(T)$ .

In practice, colours are represented by integers. To associate different integers with distinct multisets, we need to keep track of which ones we have already encountered. In order to check in constant time whether a multiset has already been seen (which is the case in the original algorithm: as the tuples are sorted in Step 5, it is enough to compare a tuple with its predecessor in the list  $S'_i$  to find out whether they are different or not), we need a perfect hash function that works on multisets. Obviously, this is a very strong assumption. Hash functions for multisets do exist – see for instance [21, 35] – but they involve advanced concepts, which would make implementation difficult for non-specialists. For the sake of this article, let us assume that we do not have access to such methods. Since we will be focusing on Python applications later on, we assume that the Python dictionary structure can be seen as a perfect hash table; it can hash integers, strings or tuples.

To get around multisets, a simple solution is to see them as lists, which we sort before hashing them as tuples. This approach, in particular, is used in Weisfeiler-Lehman algorithms, where the same problem arises – see, for example, [32, Algorithm 3.1]. The pseudocode for AHU as presented in this section, using prior sorting of multisets, is presented in Algorithm 1.

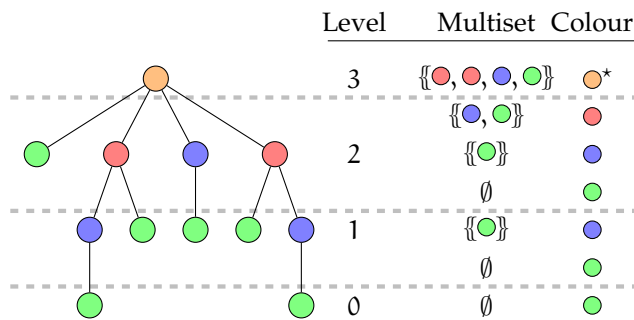


Figure 2: Assigning colours to nodes in AHU algorithm. \*: We could have used colour  $\text{green}$  because colours are assigned level by level and not globally. We have chosen to use another colour for the sake of clarity. In this example, the colours correspond exactly to the equivalence classes of the nodes.

Note that to compare the multisets of integers associated at current level  $d$ , on line 14, it is not necessary to hash but simply to sort the two lists and compare them term by term. This can be accomplished via pigeonhole sort [14]. Remember that pigeonhole sorting a list of  $k$  integers within the range 0 to  $m - 1$  is done in  $O(k+m)$ . Here, since colours are attributed at each level,  $m \leq \#T_i^d$ ; hence a complexity of  $O(\#T_i^d)$  for this step.

The overall complexity of Algorithm 1 depends on the sorting algorithm used in line 9 to sort  $\mathcal{C}_c(u)$ . It may be tempting to reuse the pigeonhole sort, already used in the algorithm, or to use a comparison sorting algorithm, such as timsort, Python's native algorithm [4]. We assume that  $T_1 = T_2 = T$  – this is the worst case, since, if  $T_1 \not\cong T_2$ , we do not visit all the levels.

---

**Algorithm 1: TREEISOMORPHISM**


---

```

Input:  $T_1, T_2$ 
Output:  $\top$  if and only if  $T_1 \cong T_2$ 
1 if  $\mathcal{D}(T_1) \neq \mathcal{D}(T_2)$  then
2   | return  $\perp$ 
3 else
4   for  $d$  from 0 to  $\mathcal{D}(T_1)$  do
5     |  $k \leftarrow 0$ 
6     | Let  $f : \emptyset \mapsto 0$ 
7     | for  $i \in \{1, 2\}$  and  $u \in T_i^d$  do
8       |  $\mathcal{C}_c(u) \leftarrow (c(v) : v \in \mathcal{C}(u))$ 
9       | Sort  $\mathcal{C}_c(u)$ 
10      | if  $f(\mathcal{C}_c(u))$  is not defined then
11        |  $k \leftarrow k + 1$ 
12        | Define  $f(\mathcal{C}_c(u)) = k$ 
13      |  $c(u) \leftarrow f(\mathcal{C}_c(u))$ 
14      | if  $\{\{c(u) : u \in T_1^d\} \neq \{c(u) : u \in T_2^d\}\}$  then
15        | return  $\perp$ 
16   | return  $\top$ 

```

---


**Proposition 1.** *Algorithm 1 runs*

- (i) in  $O\left(\frac{\#T^2}{\log \#T}\right)$  using pigeonhole sort;
- (ii) in  $O(\#T \log \deg(T))$  using timsort.

**Proof.** Fix a level  $d$  and a node  $u \in T_i^d$ . Building  $\mathcal{C}_c(u)$  requires  $O(\deg(u))$ . Then, sorting  $\mathcal{C}_c(u)$  depends on the algorithm used.

- (i) Pigeonhole sort is done in  $O(\deg(u) + \max(\mathcal{C}_c(u)))$ . Since the colours are attributed from 0 to  $\mathcal{W}(T) - 1$  – recall  $\mathcal{W}(T)$  from (1), we have  $\max(\mathcal{C}_c(u)) \leq \mathcal{W}(T)$  and therefore a complexity of  $O(\deg(u) + \mathcal{W}(T))$ .
- (ii) The worst-case complexity of timsort is  $O(\deg(u) \log \deg(u))$  [4].

Notice that  $\sum_{u \in T_i^d} \deg(u) = \#T_i^{d-1}$ . Recalling that line 14 is processed in  $O(\#T_i^d)$  via pigeonhole sort, the complexity of treating level  $d$  is  $O(\#T_i^d \mathcal{W}(T))$  for case (i), and  $O(\#T_i^d \log \deg(T))$  for case (ii) – using  $\log \deg(u) \leq \log \deg(T)$ . Summing over  $d$  leads to the result for (ii), and to  $O(\#T \mathcal{W}(T))$  for (i).

The results holds in case (i) by virtue of the following lemma, whose proof can be found in Appendix A. 

**Lemma 1.** *For any tree  $T$ ,  $\mathcal{W}(T) = O\left(\frac{\#T}{\log \#T}\right)$ .*

Neither version of Algorithm 1 is linear; we will see later in Section 3 how they behave in practice. In the next section, we will consider another approach, which does not resort to sorting, but instead replaces multiset hashing with prime number multiplication.



**Remark 2.** It should be noted that Algorithm 1 can be straightforwardly adapted to handle ordered and/or labelled trees (where each node carries a label). For ordered trees, it suffices to not sort  $\mathcal{C}_c(u)$ . For labelled trees, we have to assume that labels can be hashed. We replace  $\mathcal{C}_c(u)$  by the tuple  $(\text{label}(u), \mathcal{C}_c(u))$  and consider two tuples to be equal if both the label and the multiset are identical. Note that another way of handling labels exists, requiring not the equality of labels but rather the respect of label equivalence classes. This variant is known as marked tree isomorphism [16] and proved to be as hard as graph isomorphism – for which no polynomial algorithm is known, even though in practice very efficient algorithms exist; see [36, 10] for instance. Marked tree isomorphism is far beyond the scope of this article; but we refer the interested reader to [7, 30].

The authors of the AHU algorithm have provided in [1] a way of adapting their method to labeled trees - but only with labels that can be totally ordered. It suffices to add of label of node  $u$  as the first element of the tuple associated to it, before the lexicographical sort of Step 5. While not provided in [1], their algorithm can also be modified to account for ordered trees. In Step 3, the tuple associated to node  $u$  at level  $d$  is instead calculated as the tuple of integers associated with its children, in order. The list  $L_i$  is not longer necessary.

## 2.2 AHU with primes

In Algorithm 1, we need to associate a unique integer  $f(\mathcal{C}_c(u))$  to each distinct multiset  $\mathcal{C}_c(u)$  of integers encountered. There is a particularly simple and fundamental example where integers are associated with multisets: prime factorization. Indeed, via the fundamental theorem of arithmetic, there is a bijection between integers and multisets of primes. For example,  $360 = 2^3 \cdot 3^2 \cdot 5$  is associated to the multiset  $\{\{2, 2, 2, 3, 3, 5\}\}$ .

Note that this bijection is well known [15], and has already been successfully exploited in the literature for prime decomposition, but also usual operations such as product, division, gcd and lcd of numbers [45]. To the best of our knowledge, this link has never been exploited to replace multiset hashing, a fortiori in the context of graph isomorphism algorithms – such as Weisfeiler-Lehman, or AHU for trees. Note, however, that this approach has been used in the context of evaluating poker hands [43], where prime multiplication has been preferred to sorting cards by value in order to get a unique identifier for each distinct possible hand.

Since the previous versions of AHU we presented (both the original and our variants) sort lists of integers, the main challenge of this substitution concerns the potential additional complexity of multiplying lists of primes compared to sorting lists of integers.

---

### Algorithm 2: TREEISOMORPHISMWITHPRIMES

---

```

Input:  $T_1, T_2$ 
Output:  $\top$  if and only if  $T_1 \simeq T_2$ 
1 if  $\mathcal{D}(T_1) \neq \mathcal{D}(T_2)$  then
2   | return  $\perp$ 
3 else
4   |  $P \leftarrow [2, 3, 5, 7, 11, 13]$  and  $N_{\text{sieve}} \leftarrow 16$ 
5   | for  $d$  from 0 to  $\mathcal{D}(T_1)$  do
6     | Let  $f : 1 \mapsto 2$ 
7     |  $p \leftarrow 2$ 
8     | for  $i \in \{1, 2\}$  and  $u \in T_i^d$  do
9       |  $N(u) \leftarrow \prod_{v \in \mathcal{C}(u)} c(v)$ 
10      | if  $f(N(u))$  is not defined then
11        |  $N_{\text{sieve}}, P, p \leftarrow \text{NEXTPRIME}(N_{\text{sieve}}, P, p)$ 
12        | Define  $f(N(u)) = p$ 
13      |  $c(u) \leftarrow f(N(u))$ 
14      | if  $\{\{c(u) : u \in T_1^d\}\} \neq \{\{c(u) : u \in T_2^d\}\}$  then
15        | return  $\perp$ 
16 return  $\top$ 

```

---

Suppose that each node  $u$  at level  $d$  has received a prime number  $c(u)$ , assuming that all nodes at that level and of the same class of equivalence have received the same number. Then, to a node  $u$  at level  $d$ , instead of associating the multiset  $\mathcal{C}_c(u) = \{\{c(v) : v \in \mathcal{C}(u)\}\}$ , we associate the number  $N(u) = \prod_{v \in \mathcal{C}(u)} c(v)$ . The nodes of level  $d$  are then renumbered with prime numbers – where each distinct number  $N(u)$  gets a distinct prime. The fundamental theorem of arithmetic ensures that two identical multisets  $\mathcal{C}_c(\cdot)$  receive the same number  $N(\cdot)$ . The pseudocode for this new version of AHU is presented in Algorithm 2.



The subroutine NEXTPRIME, introduced in Algorithm 3, returns the next prime not already used at the current level; if there is no unassigned prime in the current prime list  $P$ , then new primes are generated using a segmented version of the sieve of Eratosthenes.

---

**Algorithm 3:** NEXTPRIME

---

**Input:**  $N_{\text{sieve}}$ , the largest number for which the sieve of Eratosthenes has already been performed, the list  $P$  of primes  $\leq N_{\text{sieve}}$  in ascending order, with  $\text{length}(P) \geq 6$ , and a prime  $p \in P$

- 1 **if**  $p$  is the last of element of  $P$  **then**
- 2      $N_{\text{sieve}}, P \leftarrow \text{SIEVEOFERATOSTHENES}(N_{\text{sieve}}, P)$   
       /\* At least one new prime has been added to  $P$  \*/
- 3 Let  $p'$  be the next prime after  $p$  in  $P$
- 4 **return**  $N_{\text{sieve}}, P, p'$

---

Let us denote  $p_n$  the  $n$ -th prime number. There are well known bounds on the value of  $p_n$  [22, 39] – with  $\ln$  denoting the natural logarithm and  $n \geq 6$ :


$$n(\ln n + \ln \ln n - 1) < p_n < n(\ln n + \ln \ln n). \quad (2)$$

Suppose we have the list of all primes  $P \leq N_{\text{sieve}}$ , where  $N_{\text{sieve}}$  is the largest integer sieved so far. With  $\#P = n - 1$ , to generate  $p_n$ , we simply resume the sieve up to the integer  $\lceil n(\ln n + \ln \ln n) \rceil$ , starting from  $\lceil n(\ln n + \ln \ln n - 1) \rceil$  or  $N_{\text{sieve}} + 1$ , whichever is greater – to make sure there is no overlap between two consecutive segments of the sieve. With this precaution in mind, the total complexity of the segmented sieve is the same as if we had directly performed the sieve in one go [12]; i.e.,  $O(N \log \log N)$  for a sieve performed up to integer  $N$ . Therefore, to generate the first  $n$  prime numbers, according to (2), we have  $N = n(\ln n + \ln \ln n) = O(n \log n)$  and the final complexity of the sieve can be evaluated as  $O(n \log n \log \log n)$ . See [37] for practical considerations on the implementation of the segmented sieve of Eratosthenes.

**Remark 3.** Note that other sieve algorithms exist, with better complexities – such as Atkin sieve [3] or the wheel sieve [38]; the sieve of Eratosthenes has the merit of being the simplest to implement and sufficient for our needs. Also, a better asymptotic complexity but with a worse constant can be counterproductive for producing small primes – which is rather our case since we generate the primes in order.

We now analyse the complexity of Algorithm 2, assuming that  $T_1 = T_2 = T$ . Following the previous discussion, we can consider separately the complexity for generating the primes numbers.


**Proposition 2.** Generating the primes required for Algorithm 2 can be done in  $O(\#T \log \log T)$ .

**Proof.** To generate the first  $n$  primes, the sieve must be carried out up to the integer  $n \cdot (\ln n + \ln \ln n)$ , for total complexity  $O(n \log n \log \log n)$ . As defined in (1), the number of primes needed by Algorithm 2 is equal to  $\mathcal{W}(T)$ . We result immediately follows by virtue of the upcoming lemma, whose proof can be found in Appendix A. 

**Lemma 2.** For any tree  $T$ ,  $\mathcal{W}(T) \ln \mathcal{W}(T) = O(\#T)$ .

Finally, we have the following result.

**Theorem 2.** Algorithm 2 runs in  $O(\#T \log \#T \cdot M(\deg(T) \log \#T))$  where  $M(n)$  varies from  $\log n$  to  $n$  depending on the multiplication algorithm used.

**Proof.** The proof can be found in Appendix B. 

This new variant is no more linear than the two versions introduced in Section 2.1. We shall see in Section 3 that, in practice, this version is competitive with its rivals.

## 2.3 Computing classes globally for DAG compression

In AHU algorithm as it has been presented so far, the colours assigned to the nodes are assigned level by level, and therefore make it possible to determine the equivalence class of the node relative to the level at which it is located. It is legitimate to ask whether the colours can be associated globally, so that the colour of a node is an exact reflection of its equivalence class in the tree – see Figure 2. In this way, two nodes located at different levels but having the same colour induce isomorphic subtrees.

The need to assign equivalence classes globally notably arises when considering the (lossless) compression of trees into directed acyclic graphs (DAG). Trees can have many redundancies in their structure, and the aim of DAG compression is precisely to eliminate these redundancies. There are many applications of DAG compression, some of which are: the representation of trees in computer graphics [44, 27], the simplification of queries on XML documents [18, 24] or again the computation of convolution kernels [2, 8].

The set of vertices of the DAG compression  $\mathfrak{R}(T)$  of a tree  $T$  corresponds to the set of equivalence classes of the subtrees of  $T$ . For any vertices  $a, b \in \mathfrak{R}(T)$ , the multiplicity of the arc  $a \rightarrow b$  corresponds to the number of children of class  $b$  of a subtree of class  $a$ . An example can be found in Figure 3. A more precise definition and an algorithm can be found in [25]. However, it should be noted that a simple adaptation of Algorithm 1 can also be used to construct the DAG compression of a tree, namely Algorithm 4.

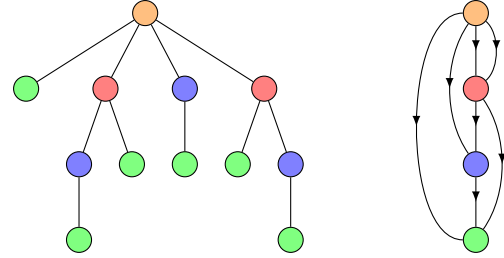


Figure 3: A tree  $T$  (left) and its DAG compression  $\mathfrak{R}(T)$  (right). Nodes are colored according to their equivalence class under  $\simeq$ .

**Remark 4.** *In light of Remark 2, the same adjustments can be applied to Algorithm 4 to take into account ordered and/or labelled trees. In addition, for labelled trees, the vertex in  $Q$  associated to the tuple  $(\text{label}(u), \mathcal{C}_c(u))$  is labelled with  $\text{label}(u)$ ;  $f$  must be initialized as an empty mapping, and  $Q$  as an empty graph. For ordered trees, the order of children of  $\mathcal{C}_c(u)$  must be respected in  $Q$ .*

As with Algorithm 1, the complexity of Algorithm 4 depends on the sorting algorithm used.

**Proposition 3.** *Algorithm 4 computes  $\mathfrak{R}(T)$*

- (i) *in  $O(\#T^2)$  using pigeonhole sort;*
- (ii) *in  $O(\#T \log \deg(T))$  using timsort.*

**Proof.** We assume that creating a new vertex can be done in  $O(\#\mathcal{C}_c(u)) = O(\deg(u))$ . Then, note that since we can determine whether  $f(\mathcal{C}_c(u))$  is defined in  $O(1)$  at line 8, the complexity of constructing the DAG is bounded by  $O(\sum_{u \in T} \deg(u)) = O(\#T)$  (the worst case would be when all nodes of  $T$  are of different equivalence class), and can be counted independently from other steps in the algorithm.

With timsort, the sorting complexity remains  $O(\deg(u) \log \deg(u))$  and the global complexity is the same as for Algorithm 1. However, for pigeonhole sort, things are different. By computing the classes globally,

---


### Algorithm 4: DAGCOMPRESSION

---

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

- 1  $k \leftarrow 0$
- 2 Let  $f : \emptyset \mapsto 0$
- 3 Let  $Q$  be a graph with a unique vertex 0
- 4 **for**  $d$  **from** 0 **to**  $\mathcal{D}(T)$  **do**
- 5     **for**  $u \in T^d$  **do**
- 6          $\mathcal{C}_c(u) \leftarrow (c(v) : v \in \mathcal{C}(u))$
- 7         Sort  $\mathcal{C}_c(u)$
- 8         **if**  $f(\mathcal{C}_c(u))$  is not defined **then**
- 9              $k \leftarrow k + 1$
- 10            Define  $f(\mathcal{C}_c(u)) = k$
- 11            Create a vertex  $k$  in  $Q$  with children  $\mathcal{C}_c(u)$
- 12          $c(u) \leftarrow f(\mathcal{C}_c(u))$
- 13 **return**  $Q$

---

the number of colours needed is not  $\mathcal{W}(T)$  but exactly  $\#\mathfrak{R}(T)$ , that can be roughly bounded by  $\#T$ ; therefore sorting  $\mathcal{C}_c(u)$  via pigeonhole has complexity  $O(\deg(u) + \#T)$  and leads to a global complexity of  $O(\#T^2)$ . 

The state-of-the-art complexity for computing  $\mathfrak{R}(T)$  is  $O(\#T \log \deg(T))$ <sup>1</sup> [25, Section 2.2.1], so our approach is consistent if we use timsort.

Algorithm 2 can also be adapted so that the DAG is constructed using multiplication of primes. As stated above, the largest colour (thus prime) used in the algorithm is no longer  $\mathcal{W}(T)$ , but  $\#\mathfrak{R}(T)$ . Using the same rough bound of  $\#\mathfrak{R}(T) \leq \#T$ , the complexity of Algorithm 2 would be modified to

$$O(\#T \log \#T \cdot (M(\deg(T) \log \#T) + \log \log \#T)),$$

the final complexity depending on the chosen multiplication algorithm and whether  $M(\deg(T) \log \#T)$  outweighs or not  $\log \log \#T$  – recalling that  $M(n)$  varies from  $\log n$  to  $n$ .

**Remark 5.** Concerning the bound  $\#\mathfrak{R}(T) \leq \#T$ , note that  $\#\mathfrak{R}(T) = \#T$  is achieved if and only if  $T$  is a chain; in which case all nodes have at most one child, and only one prime is actually needed. We could refine the bound with  $\#\mathfrak{R}(T) \leq \#T - \#\mathcal{L}(T) + 1$ , since all leaves of  $T$  have the same equivalence class. In general, if we choose a tree  $T$  uniformly at random among trees with  $n$  nodes, the expected value of  $\mathfrak{R}(T)$  is

$$\sqrt{\frac{\ln 4}{\pi}} \frac{n}{\sqrt{\ln n}} \left( 1 + O\left(\frac{1}{\ln n}\right) \right) \quad [17, Theorem 29].$$

The most compressible trees are called self-nested trees and achieve  $\#\mathfrak{R}(T) = \mathcal{D}(T)$  – on that matter, see [25, 5, 6].

Finally, note that AHU as stated in the original paper [1] cannot be extended to take into account this global assignment of colours.

### 3 Numerical experiments

Table 1 summarizes the different algorithms seen so far and whether or not they can be adapted to compute DAG compressions of trees. We implemented in Python those 7 different algorithms. All of them are written to be fully compatible with the library `treex` [9], dedicated to tree algorithms.

	Original AHU	AHU with pigeonhole sort	AHU with timsort	AHU with primes
Tree Isomorphism	$O(\#T)$	$O\left(\frac{\#T^2}{\log \#T}\right)$	$O(\#T \log \deg(T))$	$O(\#T \log \#T \cdot M(\deg(T) \log \#T))$
DAG Compression	$\times$	$O(\#T^2)$	$O(\#T \log \deg(T))$	$O(\#T \log \#T \cdot (M(\deg(T) \log \#T) + \log \log \#T))$

Table 1: Complexities of the various algorithms studied in this paper.

Most of auxiliary functions used in those algorithms have been implemented as well, such as the variant of radix sort used by AHU [1, Algorithm 3.2], pigeonhole sort or the prime variant of pigeonhole sort discussed in Appendix B. For the algorithm using primes, we have also implemented a variant that uses a pregenerated list of primes, large enough so that no additional sieving step is needed during execution – in this way we intend to study the impact of multiplication compared with sorting. The divide and conquer recursive multiplication strategy discussed in Appendix B as well as the sieve of Eratosthenes have been

<sup>1</sup>Actually, in [25], the authors announce a complexity of  $O(\#T \deg(T) \log \deg(T))$ ; but their proof does not exploit the fact that  $\sum_{u \in T} \deg(u) = \#T - 1$ . Taking this into account, we can simplify their result and obtain exactly  $O(\#T \log \deg(T))$ .

also implemented. Note that our implementation of the segmented sieve of Eratosthenes ignores multiples of 2 and 3, thus making the sieve 6 times faster. The noteworthy native Python functionalities we used are (i) the dictionary structures for hash tables, (ii) the `sort` method for lists – that use timsort algorithm [4] and (iii) the multiplication operator `*` – which use schoolbook multiplication for small integers, and Karatsuba for large ones.

All experiments have been conducted on a HP Elite Notebook with 32 Go of RAM and Intel Core i7-1365U processor.

### 3.1 Results for tree isomorphism

We generated 100 pairs of trees  $(T_1, T_2)$  of size  $n = 10^i$  for each  $i \in \llbracket 1, 6 \rrbracket$ , and for each case  $T_1 \simeq T_2$  or  $T_1 \not\simeq T_2$ , following the procedure below:

- for  $T_1 \simeq T_2$ , we generate a random recursive tree [47]  $T_1$  of size  $n$ , and generate  $T_2$  as a copy of  $T_1$ ;
- for  $T_1 \not\simeq T_2$ ,  $T_1$  and  $T_2$  are both random recursive trees of size  $n$ .

When processing a couple, we executed all five algorithms (original AHU, AHU with pigeonhole sort, AHU with timsort, AHU with primes and AHU with pregenerated primes) on that same couple, so that the results are comparable. The computation times we got are depicted in Figure 4.

As one might expect, in the case  $T_1 \simeq T_2$ , the behaviour of AHU with pigeonhole is supralinear. However, this is not the case for the other variants, which are in fact faster than the original algorithm. This can be explained by the fact that AHU has a large constant, scanning each list several times during its execution. Furthermore, there doesn't seem to be any significant difference between AHU with timsort, with primes or with pregenerated primes (for the same tree, we found that timsort is actually slightly faster in most of the cases). It's not surprising that the two versions with prime numbers are similar: it has already been established that the complexity of generating the prime numbers is negligible compared with the other steps in the algorithm. However, the proximity between timsort on the one hand, and prime numbers on the other, suggests that at this scale it is just as fast to sort lists as it is to multiply them.

Whenever  $T_1 \not\simeq T_2$ , all algorithms are roughly equally fast. This is likely due to the early stopping conditions: the distribution of random recursive trees generates trees that are too dissimilar for several levels to be visited during the execution.

**Remark 6.** *One could argue that  $10^6$  is not very large, as an upper limit to our simulations. However, most of the real tree databases we are aware of already do not have trees of this size. Among the 8 datasets studied in [8], the largest trees have a few thousand nodes; among the 5 studied in [42], a few hundred. If gigantic tree databases were to be built (for example, spanning trees from graphs with billions of nodes), it seems reasonable to imagine that they would be processed, in any case, with algorithms implemented in C, C++ or Rust rather than in Python (at the very least to be able to compute them efficiently, with the spanning trees example).*

### 3.2 Results for DAG compression

Here, we generated 100 random recursive trees  $T$  of size  $n = 10^i$  for each  $i \in \llbracket 1, 6 \rrbracket$ . The results are provided in Figure 5. One can see that the pigeonhole variant seems to confirm a quadratic behaviour, while the timsort variant confirms its slight advantage over the versions with prime numbers. This time, a difference can be seen between the version that uses pregenerated primes and the one that does not (as might be expected given the theoretical complexity).

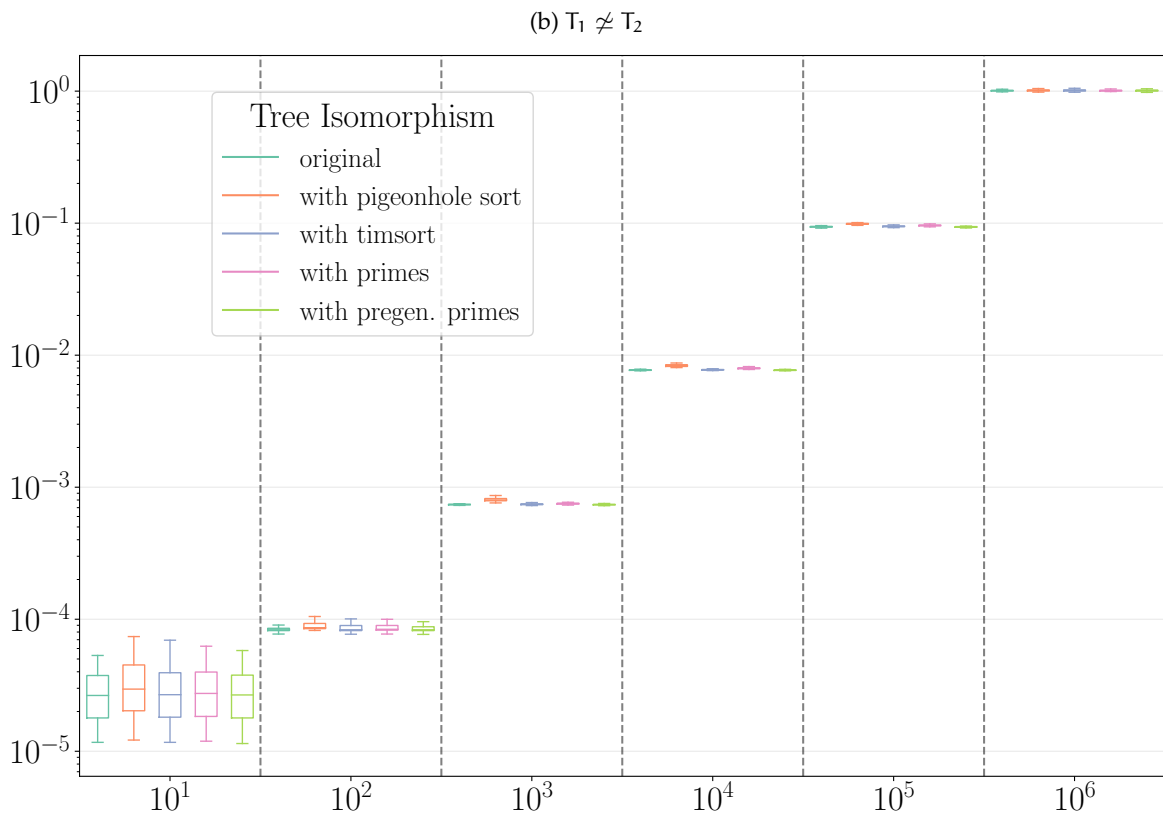
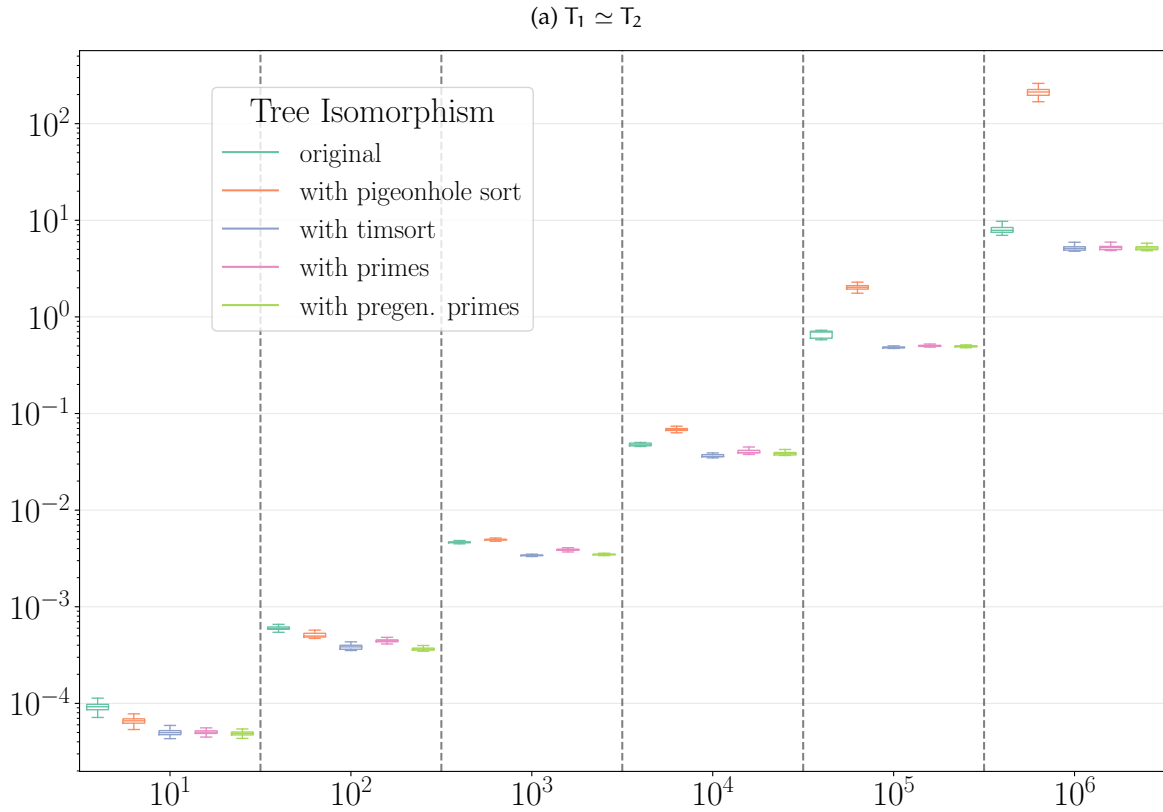


Figure 4: Computation times (in log scale) for tree isomorphism using different algorithms, according to the size of the trees, with 100 replicates for each size.

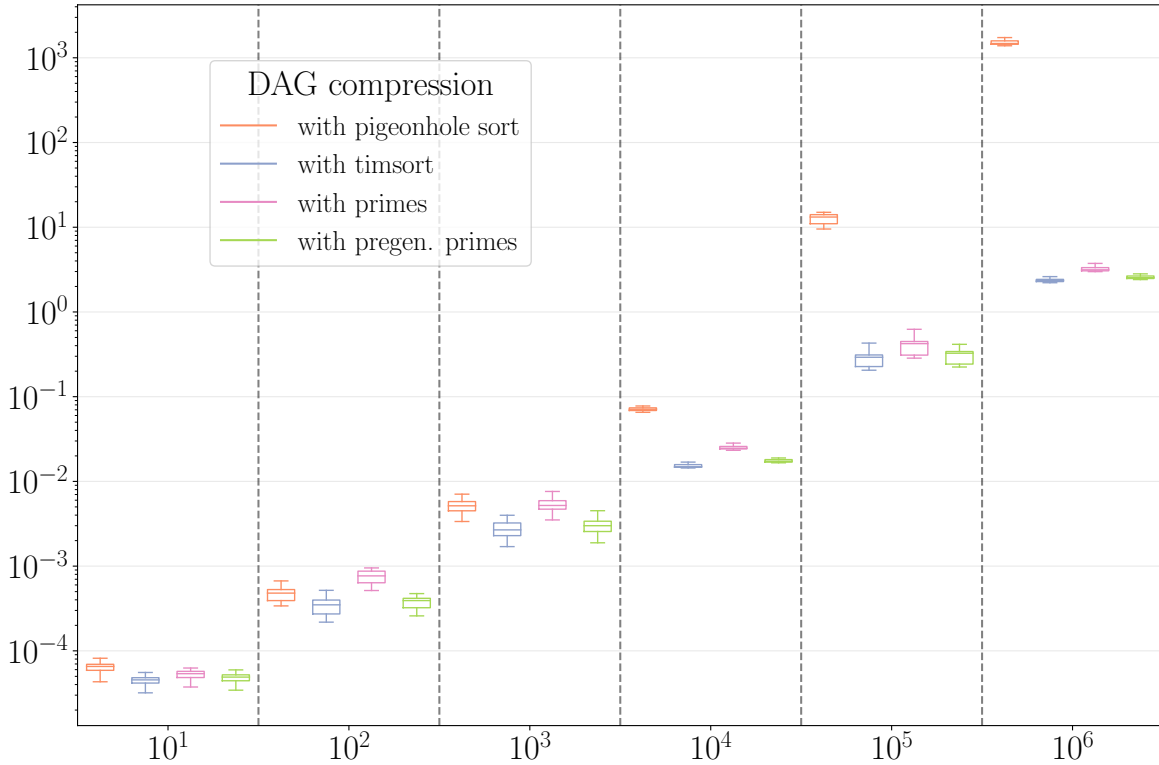


Figure 5: Computation times (in log scale) for DAG compression using different algorithms, according to the size of the trees, with 100 replicates for each size.

## Concluding remarks

In this article, we have provided a new intuition for understanding the AHU algorithm, as well as several algorithmic variants that are straightforward to implement, albeit at the cost of increased complexity. However, we have shown that on trees of reasonable size, with a Python implementation, some of these variants were faster than the original algorithm. We have shown that a simple adaptation of our algorithms can also be used to calculate the DAG compression of trees.

Perhaps counter-intuitively, we have shown that in this context we can multiply lists of primes almost as quickly as we can sort lists of integers. However, if one had to pick just one variant, the one with timsort would probably be the simplest to implement and the most effective in practice – bot for tree isomorphism and DAG compression. The version using prime numbers could possibly become competitive with timsort if the list multiplication and prime number generation operations were implemented in CPython (as is the case for timsort), but this is beyond the scope of this article.

One may also wonder whether our variant using prime numbers could be applied to other algorithms similar to AHU, such as (1-dimensional) Weisfeiler-Lehman algorithm for graph isomorphism. While this issue is outside the scope of this paper, and remains to be investigated, let us nonetheless mention two points that may prove challenging. First, the way Weisfeiler-Lehman operates can lead to processing as many colours as there are nodes in the graph, and therefore having to generate as many prime numbers – which relates to the DAG compression case studied in this paper, for which our variant performed slightly worse than the others. Next, we would multiply lists whose size depends on the degree of the current node; in a dense or complete graph, this means lists whose size is comparable to the number of nodes



in the graph. The complexity of performing these multiplications could prove far more expensive than for trees. Since Weisfeiler-Lehman can be implemented in  $O((\#V + \#E) \log \#V)$  for a graph  $G = (V, E)$ , it remains to be investigated to which extent the additional complexities mentioned above exceeds that of the original algorithm. See [32, Section 3.1] and references therein for a more precise description of the Weisfeiler-Lehman algorithms.

## Acknowledgements

The author would like to thank Dr. Romain Azaïs and Dr. Jean Dupuy for their helpful suggestions on the first drafts of the article; as well as the anonymous reviewers who provided feedback on a previous version of this article.

## References

- [1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.
- [2] 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.
- [3] Arthur Atkin and Daniel Bernstein. “Prime sieves using binary quadratic forms”. In: *Mathematics of Computation* 73.246 (2004), pp. 1023–1030.
- [4] Nicolas Auger et al. “On the worst-case complexity of TimSort”. In: *arXiv preprint arXiv:1805.08612* (2018).
- [5] Romain Azaïs. “Nearest embedded and embedding self-nested trees”. In: *Algorithms* 12.9 (2019), p. 180.
- [6] Romain Azaïs, Jean-Baptiste Durand, and Christophe Godin. “Approximation of trees by self-nested trees”. In: *2019 proceedings of the twenty-first workshop on algorithm engineering and experiments (alenex)*. SIAM. 2019, pp. 39–53.
- [7] Romain Azaïs and Florian Ingels. “Detection of common subtrees with identical label distribution”. In: *Theoretical Computer Science* (2023), p. 114366.
- [8] Romain Azaïs and Florian Ingels. “The weight function in the subtree kernel is decisive”. In: *Journal of Machine Learning Research* 21.67 (2020), pp. 1–36.
- [9] Romain Azaïs et al. “Treex: a Python package for manipulating rooted trees”. In: *Journal of Open Source Software* 4.38 (2019), p. 1351.
- [10] László Babai. “Graph isomorphism in quasipolynomial time”. In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 2016, pp. 684–697.
- [11] David A Barrington. “Bounded-width polynomial-size branching programs recognize exactly those languages in NC”. In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 1986, pp. 1–5.
- [12] Carter Bays and Richard H Hudson. “The segmented sieve of Eratosthenes and primes in arithmetic progressions to  $10^{12}$ ”. In: *BIT Numerical Mathematics* 17.2 (1977), pp. 121–127.
- [13] Jon Louis Bentley, Dorothea Haken, and James B Saxe. “A general method for solving divide-and-conquer recurrences”. In: *ACM SIGACT News* 12.3 (1980), pp. 36–44.
- [14] Paul E Black et al. “Dictionary of algorithms and data structures”. In: (1998).
- [15] Wayne D. Blizard et al. “Multiset theory”. In: *Notre Dame Journal of formal logic* 30.1 (1989), pp. 36–66.

- [16] Kellog S Booth and Charles J Colbourn. *Problems polynomially equivalent to graph isomorphism*. Computer Science Department, Univ., 1979.
- [17] Mireille Bousquet-Mélou et al. “XML compression via directed acyclic graphs”. In: *Theory of Computing Systems* 57 (2015), pp. 1322–1371.
- [18] Peter Buneman, Martin Grohe, and Christoph Koch. “Path queries on compressed XML”. In: *Proceedings 2003 VLDB Conference*. Elsevier. 2003, pp. 141–152.
- [19] Samuel R. Buss. “Alogtime algorithms for tree isomorphism, comparison, and canonization”. In: *Kurt Gödel Colloquium on Computational Logic and Proof Theory*. Springer. 1997, pp. 18–33.
- [20] Douglas M Campbell and David Radford. “Tree isomorphism algorithms: Speed vs. clarity”. In: *Mathematics Magazine* 64.4 (1991), pp. 252–261.
- [21] Dwaine Clarke et al. “Incremental multiset hash functions and their application to memory integrity checking”. In: *International conference on the theory and application of cryptology and information security*. Springer. 2003, pp. 188–207.
- [22] Pierre Dusart. “The  $k$ -th prime is greater than  $k(\ln k + \ln \ln k - 1)$  for  $k \geq 2$ ”. In: *Mathematics of computation* (1999), pp. 411–415.
- [23] Scott Fortin. “The graph isomorphism problem”. In: (1996).
- [24] 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.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] David Harvey and Joris Van Der Hoeven. “Integer multiplication in time  $O(n \log n)$ ”. In: *Annals of Mathematics* 193.2 (2021), pp. 563–617.
- [29] Ningyuan Teresa Huang and Soledad Villar. “A Short Tutorial on The Weisfeiler-Lehman Test And Its Variants”. In: *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2021, pp. 8533–8537.
- [30] Florian Ingels and Romain Azaïs. “Isomorphic unordered labeled trees up to substitution ciphering”. In: *International Workshop on Combinatorial Algorithms*. Springer. 2021, pp. 385–399.
- [31] Anatolii Alekseevich Karatsuba and Yu P Ofman. “Multiplication of many-digital numbers by automatic computers”. In: *Doklady Akademii Nauk*. Vol. 145. 2. Russian Academy of Sciences. 1962, pp. 293–294.
- [32] Sandra Kiefer. “Power and limits of the Weisfeiler-Leman algorithm”. PhD thesis. Dissertation, RWTH Aachen University, 2020.
- [33] Donald E Knuth. *The Art of Computer Programming: Fundamental Algorithms, volume 1*. Addison-Wesley Professional, 1997.
- [34] Pengyu Liu. “A tree distinguishing polynomial”. In: *Discrete Applied Mathematics* 288 (2021), pp. 1–8.
- [35] Jeremy Maitin-Shepard, Mehdi Tibouchi, and Diego F Aranha. “Elliptic curve multiset hash”. In: *The Computer Journal* 60.4 (2017), pp. 476–490.
- [36] Brendan D. McKay and Adolfo Piperno. “Practical graph isomorphism, II”. In: *Journal of symbolic computation* 60 (2014), pp. 94–112.

- [37] Melissa e O’neill. “The genuine sieve of Eratosthenes”. In: *Journal of Functional Programming* 19.1 (2009), pp. 95–106.
- [38] Paul Pritchard. “Linear prime-number sieves: A family tree”. In: *Science of computer programming* 9.1 (1987), pp. 17–35.
- [39] Barkley Rosser. “Explicit bounds for some functions of prime numbers”. In: *American Journal of Mathematics* 63.1 (1941), pp. 211–232.
- [40] Nitin Saxena. “Progress on Polynomial Identity Testing.” In: *Bull. EATCS* 99 (2009), pp. 49–79.
- [41] Uwe Schöning. “Graph isomorphism is in the low hierarchy”. In: *Journal of Computer and System Sciences* 37.3 (1988), pp. 312–323.
- [42] Kilho Shin and Tetsuji Kuboyama. “A comprehensive study of tree kernels”. In: *New Frontiers in Artificial Intelligence: JSAI-isAI 2013 Workshops, LENLS, JURISIN, MiMI, AAA, and DDS, Kanagawa, Japan, October 27–28, 2013, Revised Selected Papers* 5. Springer. 2014, pp. 337–351.
- [43] Kevin Suffecool. *Cactus kev’s poker hand evaluator*. 2005.
- [44] Ivan E Sutherland. “Sketchpad: A man-machine graphical communication system”. In: *Proceedings of the May 21-23, 1963, spring joint computer conference*. 1963, pp. 329–346.
- [45] Paul Tarau. “Towards a generic view of primality through multiset decompositions of natural numbers”. In: *Theoretical Computer Science* 537 (2014), pp. 105–124.
- [46] Gabriel Valiente. *Algorithms on trees and graphs*. Vol. 112. Springer, 2002.
- [47] Yazhe Zhang. “On the number of leaves in a random recursive tree”. In: (2015).

## A Proof of Lemma 1 and Lemma 2

We conduct the proof of Lemma 1 and Lemma 2 by first introducing a special tree, which for a given width has the smallest possible number of nodes, before observing how these two quantities are related.

### A.1 A special tree

Let  $k \geq 1$  be a fixed integer. A tree  $T$  such that  $\mathcal{W}(T) = k$  can be obtained by placing  $k$  trees  $T_i$ ,  $i \in \llbracket 1, k \rrbracket$ , under a common root, so that  $T_i \not\cong T_j$  for  $i \neq j$ . Note that this construction by no means encompasses all types of trees  $T$  with  $\mathcal{W}(T) = k$ . On the other hand, by cleverly choosing the  $T_i$ ’s, we can build a tree with the minimum number of nodes among all trees verifying  $\mathcal{W}(T) = k$ .

First,  $T_1$  would be the tree with a unique node. Then,  $T_2$  the only tree with two nodes. Then,  $T_3$  and  $T_4$  would be the two non-isomorphic trees with three nodes;  $T_5$  to  $T_8$  the four non-isomorphic trees with four nodes, and so on until we reach  $T_k$ . See Figure 6 for an example with  $k = 5$ . It should be clear that this construction ensures that  $\mathcal{W}(T) = k$  and  $\#T$  is minimal.

Following this construction, the total number of nodes in  $T$ , that we denote by  $t_k$ , is therefore closely related to the number of non-isomorphic trees and their cumulative sum. Let us denote  $a_n$  the number

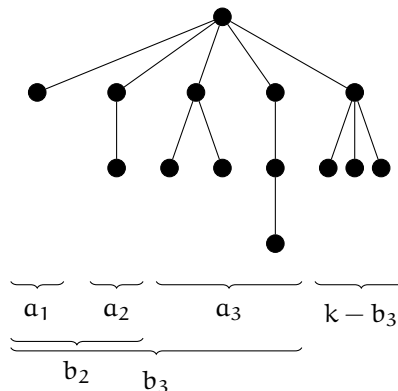


Figure 6: The smallest tree so that  $\mathcal{W}(T) = 5$ . We have  $b_3 < 5 \leq b_4$  and  $\#T = 1 + 1 \cdot a_1 + 2 \cdot a_2 + 3 \cdot a_3 + 4 \cdot (5 - b_3) = 14$ .

of non-isomorphic trees of size  $n$ , and  $b_n$  the number of non-isomorphic trees of size at most  $n$  – so that  $b_n = \sum_{i=1}^n a_i$ . Let  $n$  be the integer so that  $b_n < k \leq b_{n+1}$ . All trees with size up to  $n - 1$  are used in the construction, as well as  $k - b_n$  trees of size  $n + 1$  (no matter which ones).

Therefore,

$$t_k = 1 + \sum_{i=1}^n i \cdot a_i + (n+1)(k - b_n).$$

$n$	1	2	3	4	5	6	7	8
$a_n$	1	1	2	4	9	20	48	115
$b_n$	1	2	4	8	17	37	85	200
$t_n$	2	4	7	10	14	18	22	26

Table 2 provides the first values for  $a_n$ ,  $b_n$  and  $t_k$ . Following the previous discussion, we have the following result.

**Lemma 3.** For any tree  $T$ ,  $\#T \geq t_{\mathcal{W}(T)}$ .

Table 2: First values of  $a_n$ ,  $b_n$  and  $t_n$ .  $a_n$  is sequence [A000081](#) in OEIS, and  $b_n$  sequence [A087803](#). See OEIS Foundation Inc. (2023), The On-Line Encyclopedia of Integer Sequences, Published electronically at <https://oeis.org>.


## A.2 Relationships between $k$ and $t_k$

We require some preliminary results. We begin with the following lemma.

**Lemma 4.** Let  $(u_n)_{n \in \mathbb{N}}$  be a sequence so that  $u_n \underset{+\infty}{\sim} c \cdot d^n \cdot n^{-\alpha}$ , with  $c, \alpha \geq 0$  and  $d > 1$ . Then  $\sum_{k=1}^n u_k \underset{+\infty}{\sim} c \cdot \frac{d}{d-1} \cdot d^n \cdot n^{-\alpha}$ .

**Proof.** Obviously the sequence  $\sum d^n \cdot n^{-\alpha}$  diverges, and therefore we have  $\sum_{k=1}^n u_k \underset{+\infty}{\sim} c \sum_{k=1}^n d^k \cdot k^{-\alpha}$ . Then,

$$\frac{c \sum_{k=1}^n d^k \cdot k^{-\alpha}}{c \cdot \frac{d}{d-1} \cdot d^n \cdot n^{-\alpha}} = \frac{d-1}{d} \sum_{k=1}^n d^{k-n} \left(\frac{k}{n}\right)^{-\alpha} \underset{j=n-k}{=} \frac{d-1}{d} \sum_{j=1}^n \left(1 - \frac{j}{n}\right)^{-\alpha} d^{-j}.$$

With bounds  $1 \leq \left(1 - \frac{j}{n}\right)^{-\alpha} \leq \left(1 - \frac{1}{n}\right)^{-\alpha}$ , it is easy to see that the right-hand term goes to 1 as  $n \rightarrow \infty$ . 

From [33, Section 2.3.4.4], we have  $a_n \underset{+\infty}{\sim} c \cdot d^n \cdot n^{-3/2}$  with  $c \approx 0.439924$  and  $d \approx 2.955765$ . From Lemma 4, we immediately derive  $b_n \underset{+\infty}{\sim} c \cdot \frac{d}{d-1} \cdot d^n \cdot n^{-3/2}$ . Finally, noticing that  $i \cdot a_i \underset{+\infty}{\sim} c \cdot d^i \cdot i^{-1/2}$ , we derive from Lemma 4 that  $\sum_{i=1}^n i \cdot a_i \underset{+\infty}{\sim} c \cdot \frac{d}{d-1} \cdot d^n \cdot n^{-1/2}$ .

We now derive our main results.


**Lemma 5.**  $kn \underset{+\infty}{\sim} t_k$  – with  $n$  as defined in Subsection A.1, i.e. so that  $b_n < k \leq b_{n+1}$ .

**Proof.** We have

$$\frac{t_k}{kn} = \frac{1 + \sum_{i=1}^n i \cdot a_i + (n+1)(k - b_n)}{kn} = \frac{1 + (n+1)k}{nk} + \frac{\sum_{i=1}^n i \cdot a_i - (n+1)b_n}{nk}.$$


First, the left-hand term tends to 1 as  $k \rightarrow \infty$ . We now prove that the right-hand term tends to 0 as  $k \rightarrow \infty$ . Since  $b_n < k$ , we have

$$\left| \frac{\sum_{i=1}^n i \cdot a_i - (n+1)b_n}{nk} \right| < \left| \frac{\sum_{i=1}^n i \cdot a_i - nb_n}{nb_n} \right| + \frac{b_n}{nb_n}.$$

Notice that  $\sum_{i=1}^n i \cdot a_i \sim nb_n$ . By definition of  $\sim$  and  $o(\cdot)$  notations, for any (positive) sequences  $u_n$  and  $v_n$ ,  $u_n \sim v_n \iff u_n - v_n = o(v_n) \iff \frac{u_n - v_n}{v_n} \rightarrow 0$ , hence the result. 

**Lemma 6.**  $\ln k \underset{+\infty}{\sim} \ln t_k \underset{+\infty}{\sim} n \ln d$  with  $d \approx 2.955765$ .

**Proof.** By definition,  $b_n < k \leq b_{n+1}$  and  $\sum_{i=1}^n i \cdot a_i < t_k \leq 1 + \sum_{i=1}^{n+1} i \cdot a_i$ . Hence,  $k \sim b_n$  and  $t_k \sim \sum_{i=1}^n i \cdot a_i$ .

Taking the logarithm of the asymptotic equivalents provided earlier on both equations yields the result.  Combining the two previous lemmas, we derive the following two corollaries.

**Corollary 1.**  $k = O\left(\frac{t_k}{\ln(t_k)}\right)$

From Lemma 3, and since  $x \mapsto \frac{x}{\log x}$  is increasing (for  $x \geq 3$ ), we get that for any tree  $T$ ,  $\mathcal{W}(T) = O\left(\frac{\#T}{\log \#T}\right)$  – hence Lemma 1 holds.

**Corollary 2.**  $k \ln k = O(t_k)$ .

From Lemma 3, we get that for any tree  $T$ ,  $\mathcal{W}(T) \ln \mathcal{W}(T) = O(\#T)$ , which ends the proof of Lemma 2.

## B Proof of Theorem 2

Following Proposition 2, the generation of primes is done in  $O(\#T \log \log \#T)$ . This term vanishes in the final complexity due to the upcoming term in  $\#T \log \#T$ . Let us denote by  $p_{\mathcal{W}(T)}$  the largest prime needed by the algorithm. Fix  $d \in \llbracket 0, \mathcal{D}(T) \rrbracket$  and  $u \in T^d$ .

**Complexity of multiplication** The complexity for multiplying two  $n$ -bits numbers varies from  $O(n^2)$  for usual schoolbook algorithm, to  $O(n \log n)$  [28] – even if this result is primarily theoretical, by the authors’ own admission. Karatsuba algorithm, which is widely used, runs in  $O(n^{\log 3})$  [31]. This algorithm is actually used in Python when the numbers get large, and schoolbook otherwise. Let us denote the complexity of multiplication as  $n \cdot M(n)$ , with  $M(n)$  varying from  $\log n$  to  $n$  depending on the algorithm used.

Multiplying two  $n$ -bits numbers together yields a  $2n$ -bits number. To compute the product of  $m$  numbers of  $n$  bits, we adopt a divide and conquer approach and multiply two numbers which themselves are the recursive product of  $m/2$  numbers. This strategy leads to a complexity of  $O(mn \cdot M(mn))$  by virtue of the Master Theorem [13]. Since computing  $N(u)$  implies multiplying  $\deg(u)$  primes with at most  $\log p_{\mathcal{W}(T)}$  bits, this lead to a complexity of  $O(\deg(u) \log p_{\mathcal{W}(T)} \cdot M(\deg(u) \log p_{\mathcal{W}(T)}))$ . Using (2) we have  $p_{\mathcal{W}(T)} < \mathcal{W}(T) (\ln \mathcal{W}(T) + \ln \ln \mathcal{W}(T))$ . With Lemma 2, we have  $p_{\mathcal{W}(T)} = O(\#T)$ ; thus a final complexity of  $O(\deg(u) \log \#T \cdot M(\deg(u) \log \#T))$  for computing  $N(u)$ .

**Other points** Testing whether or not  $f(N(u))$  is defined in line 12 can be done in  $O(1)$  since  $N(u)$  is an integer, as per our assumption of perfect hash tables working with integers, strings and tuples.

For comparing the multisets in line 16, we resort to pigeonhole sort as for Algorithm 1. Classic pigeonhole would have complexity  $O(\#T^d + p_n)$ , where  $p_n$  is the biggest prime in the list; but many holes will be unnecessary (as  $c(u)$  is necessarily prime). Using a perfect hash table that associate to the  $i$ -th prime number the integer  $i$ , one can use only  $n$  holes, one for each prime number, which reduces the complexity to  $O(\#T^d + n)$ . Since the primes are reallocated at each level, at level  $d$  we need as many primes as there are different equivalence classes at that level – i.e.  $\#\{c(u) : u \in T^d\}$ . This number is  $\leq \#T^d$ , therefore the complexity of the sort collapses to  $O(\#T^d)$ .

**Conclusion** Processing level  $d$  thus requires

$$O\left(\sum_{u \in T^d} [\deg(u) \cdot \log \#T \cdot M(\deg(u) \cdot \log \#T)] + \#T^d\right).$$

First, notice that  $\sum_{u \in T^d} \deg(u) = \#T^{d-1}$ . Bounding other occurrences of  $\deg(u)$  by  $\deg(T)$  and summing over  $d$  leads to the claim.