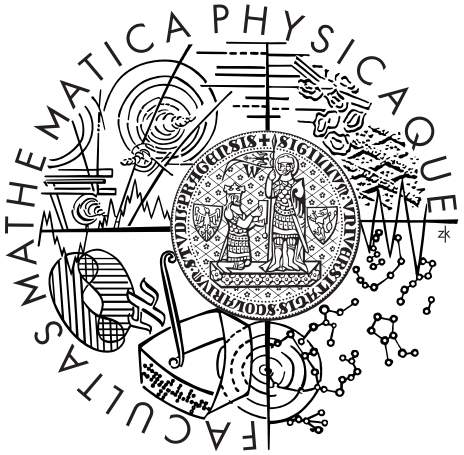
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Böhm

Graph labeling

Department of Applied Mathematics, MFF UK

Supervisor of the bachelor thesis: Mgr. Martin Mareš, PhD.
Study programme: Informatika
Specialization: Obecná informatika

Prague 2011

I thank my advisor and my family for their unending patience and support.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, August 5th, 2011 Martin Böhm

Název práce: Graph labeling

Autor: Martin Böhm

Katedra: Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš, PhD., Katedra aplikované matematiky, MFF UK

Abstrakt: Práce představuje výsledky v oblasti schémat pro značkování grafů, která kódují sousednost vrcholů. Tato schémata mají praktické aplikace v oblasti paralelních algoritmů, souvisí však i s teorií univezálních grafů. Práce se soustředí na moderní metodu Traversal and Jumping, jejíž důkaz správnosti je zjednodušen a opraven. Také se zabýváme hledáním malých univerzálních grafů hrubou silou.

Klíčová slova: teorie grafů, komprese, univerzální struktury

Title: Graph labeling

Author: Martin Böhm

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, PhD., Department of Applied Mathematics, MFF UK

Abstract: We introduce the concept of adjacency labeling schemes and recent results in the area. These schemes have practical application in parallel algorithm design and they relate to the theory of universal graphs. We concentrate on the modern technique of "Traversal and Jumping". We present a less technical proof of its correctness as well as correcting some errors in the original proof. We also apply brute-force algorithms to find small induced-universal graphs.

Keywords: graph theory, compression, universal structures

Contents

Introduction

In many applications of graph theory we encounter the problem that the graph which we are traversing is too large to be completely stored in the memory of a computer. This is of crucial importance for distributed algorithms, which often need to work on large graphs. Search engines manipulating large XML trees also find use of this technique.

For manipulating such immense data, it is necessary to encode the graph structure in such a way that we can decode the local structure of the graph only from a fraction of the complete encoded structure. We concentrate on *adjacency labeling schemes*, which are methods for encoding only the most basic graph structure – adjacency – in local manner.

There is also a close link between adjacency labeling schemes and a class of *universal graphs*, which arise in the field of graph theory.

Our work focuses on the adjacency labeling schemes for the class of *trees*, which are often used to store data without cyclic dependencies (XML trees, for example).

In the first part of the thesis, we introduce the concept of labeling and list some landmark results of the area. In the second part of our thesis, we concentrate on a seminal proof regarding a specific subclass of trees – using a method called Traversal and Jumping – and reprove the original result while fixing the shortcomings of the original proof. In the final part of the thesis, we use algorithmic means to search for small induced-universal graphs.

1. Preliminaries

1.1 Graph basics

We assume basic knowledge in the field of graph theory – for definitions of graphs, trees, orientations, please see a textbook on Graph Theory.

In general graphs, we say vertices are connected to each other, or are neighbours. This symmetric relation is called **adjacency**. In rooted trees (trees with edges oriented away from the root), given two adjacent vertices u and v, we do prefer an assymetric relation and say that u is a **parent** of v if u and v are adjacent, but u is closer to the root than v.

We define **depth** of a vertex in a rooted tree in this way: the root has depth 0 and every child of a vertex of depth k has depth $k + 1$.

If we take the reflexive and transitive closure of the "being a parent" relation, we get the **ancestry relation**. The reader can think of ancestry in family trees, as it is the same.

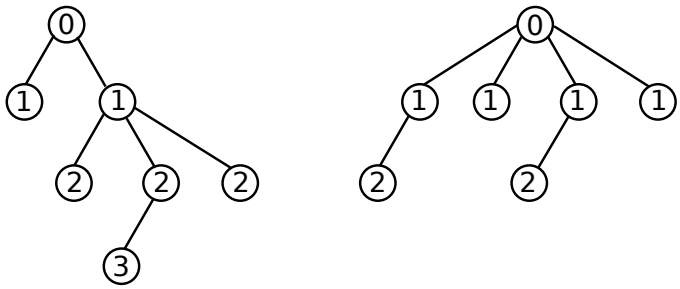


Figure 1. Two rooted trees with depth inside the vertices. Note that the two trees are isomorphic if viewed as unrooted trees, but the choice of the root changes the parent relation.

A **caterpillar** is a tree C with two types of vertices: S and L. The vertices of S (the **spine**) induce a path in graph C, while every vertex of L (these are called **legs**) is connected with exactly one vertex $s \in S$.

We can observe that if the spine vertices at the end of the induced spine path carry no leg vertices, they can be assigned as leg vertices to their spine neighbours. Therefore, we may assume that every spine vertex at the end of the path has at least one associated leg.

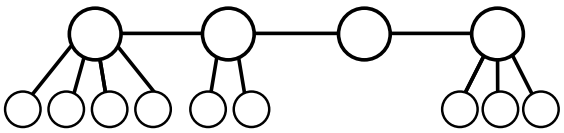


Figure 2. A caterpillar with 4 spine vertices and 9 leg vertices. We can create an isomorphic caterpillar with 5 spine vertices and 8 legs, or an isomorphic one with 6 spine vertices and 7 legs.

For a non-negative integer x, $\mathrm{bin}(x)$ will denote its binary representation. When working with binary strings, we will denote their concatenation as $x \circ y$.

In most contexts, the $\log n$ will denote the **binary logarithm** of n, usually restricted only to non-negative numbers. We often use logarithms for estimation, and so, when dealing with $\log 0$, we define it to be 0. Also, in chapter 2 of the

thesis, we often assume n is a power of 2, so that we do not have to differentiate between $\lceil \log n \rceil$ and $\log n$ – in estimations, the difference is usually purely technical. (Note that this constant can be often hidden into $\mathcal{O}(1)$.)

The **iterated logarithm** $\log^* n$, a function growing asymptotically slower than any chain of logarithms with a fixed length, is defined thus: $\log^* n = s$ if and only if we need to iterate the logarithm operation s times, starting from n, until we get a negative number. To show a few examples, $\log^* 223 = 5$ and $\log^* 1048576 = 6$. The iterated logarithm is an inverse function to the tower function $T(k) = 2^{2^{2^{\cdots}}}\Big\}k$ times.

We will also use $\ulcorner p \urcorner$ as p rounded up to the nearest power of two. (We still employ $\lceil q \rceil$ as rounding up to the nearest integer.)

We employ the Iverson notation for conveniently using predicates inside formulas. If $P(x)$ is a Boolean predicate dependent on a variable x, we define $[P(x)]$ as 1 if $P(x)$ is true, and zero otherwise.

1.2 Adjacency labeling scheme

Our goal is to search for functions that encode the local structure of a graph into computer-readable numbers.

Being precise, we want to label vertices of a graph G with binary strings – elements of the set $\{0,1\}^l$ for a suitable l. A **labeling** is thus a function $\lambda : V \to \{0,1\}^l$. However, not every labeling is of interest to us, we concentrate on adjacency labeling functions, which are defined as follows:

Definition 1. An **adjacency-labeling function** is a function $\lambda : V \to \{0,1\}^l$ for which the following conditions hold:

- The labeling λ is injective. (The labels are unique.)
- There exists a decoding function $\delta : \{0,1\}^l \times \{0,1\}^l \to \{0,1\}$ such that $\delta(\lambda(x), \lambda(y)) = 1$ if and only if the vertices x and y are adjacent in the original graph.

It is not possible to label all possible graphs, as the label length l would then be unbounded. Therefore, we often restrict ourselves to a graph subclass $\mathbb{G}$, usually with a fixed limit on the number of vertices n.

Definition 2. The triple $(\mathbb{G}, \lambda, \delta)$ is a **adjacency labeling scheme** if $\mathbb{G}$ is a graph class and for every graph $G \in \mathbb{G}$ is λ an adjacency labeling function with decoding function δ.

We can see from the definition that while λ can encode the graph using global information about it, the decoding function δ can work only with the bit strings without knowing which graph from $\mathbb{G}$ has been encoded.

In general practice it is often expected that the δ has worst-case time complexity $\mathcal{O}(1)$, and that λ has worst-case time complexity $\mathcal{O}(|V(G)| + |E(G)|)$.

1.3 Universal graphs

Given a set of graphs $\mathbb{G}$, a graph U is called **universal** if it contains every graph from $\mathbb{G}$ as a subgraph. A subtype of universal graphs that is more of interest to us are *induced-universal* graphs: A graph U' is induced-universal for a set $\mathbb{G}'$ if every graph in $\mathbb{G}'$ is an induced subgraph of U.

Note that for every graph class with size at most n, there is a simple universal graph of size n – the complete graph K_n. Our work concentrates on the induced-universal graphs and therefore, we will call induced-universal graphs simply universal graphs, unless noted otherwise.

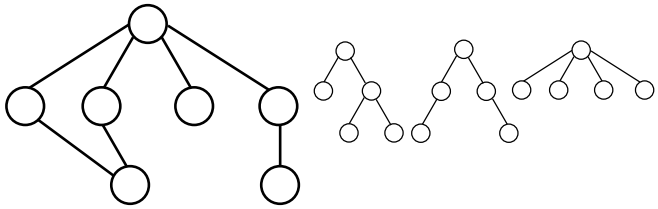


Figure 3. An induced-universal graph for trees of size 5 and the list of non-isomorphic trees of that size.

The following lemma connects the universal graphs, which have been originally studied in the field of mathematics, with adjacency-labeling schemes which come from computer science:

Lemma 1. *Given an adjacency labeling scheme for a graph class $\mathbb{G}$ which is b bits long, we can construct a universal graph of size 2^b.*

Second, given a universal graph of size 2^b for a graph class $\mathbb{G}$, we can construct an adjacency labeling scheme with labels of lengths b for such class.

Proof. We can create a "labeling graph" L where we consider every possible label of b bits as a vertex, while edges are defined by our adjacency decoding function δ. If the labeling is correct, the algorithm must not make mistakes, so when given a graph $G \in \mathbb{G}$, we can use the encoding function λ which actually assigns to every vertex $v \in V(G)$ a vertex $\lambda(v) \in L$, and see that the image of $\Lambda(G)$ is an induced subgraph of L isomorphic to G.

For the second part of the lemma, assume we have been given a universal graph L'. It has 2^b vertices, so we assign every vertex a unique non-negative integer from the interval $[0, 2^b - 1]$, which we will call an identifier. Now, since L' is fixed, of limited size and universal, we can for every $G \in \mathbb{G}$ (say by going through all possibilities) find an induced subgraph in L' isomorphic to G. We label the vertices of G by the bit representation of the identifiers belonging to its copy in L'.

The decoding function δ simply looks at the universal graph L', treats given bit strings as identifiers in L' and answers adjacency based on edges in the universal graph. □

For some classes of graphs, adjacency labeling of length $\mathcal{O}(\log n)$ can never exist, and therefore, the universal graph has to have at least $2^{\Omega(\log n)} = \Omega(n)$ vertices. This holds because every two different graphs with labels of length $\mathcal{O}(\log n)$ must differ at least in one bit in their labels, and so there can be at

most $2^{\mathcal{O}(\log n)}$ such graphs. For example, bipartite graphs of size n cannot have any universal graph of size $\mathcal{O}(n)$ [2].

Universal graphs are often studied in combinatorics, most often their infinite versions, but the finite ones have also received some attention [11]. The universal graphs have been studied even before the notion of adjacency labeling was known [10].

1.4 Suffix codes

The "Traversal and Jumping" method, which is discussed in the third part of the thesis, is based on encoding numbers into binary strings with fixed prefixes so that they can be later extracted without knowing what the original number was. The technique of encoding all non-negative integers into such binary strings is called suffix (or prefix) coding, and the resulting code is called a **suffix code**.

The name "prefix code" is more common in literature, but it refers to the opposite property than we use. Every suffix code can be made into a prefix code simply by reversing it. To avoid confusion, we will henceforth speak only about suffix codes.

Our goal is to encode every non-negative number x into a binary string $C(x)$ so that when we prepend other binary strings in front of $C(x)$, we can still decode x. We can state the requirement like this: no encoded number can be a suffix of another encoded number.

Suffix codes have a valuable property: that chaining a constant number of suffix codes creates another suffix code. This also means that we can encode and decode pairs of integers without needing any form of delimeters. Traversal and Jumping uses suffix codes in precisely this manner.

As it would be expected, it is impossible to use a suffix encoding of all non-negative integers and not expect a non-trivial increase in size compared to their standard binary encoding. We always need at least $\log n + \Omega(\log \log n)$ space for numbers from 1 to n.

Traversal and Jumping, as implemented in [1] and in our thesis, uses the following set of recursive prefix codes, which encode all non-negative integers x:

$$\text{code}_0(x) = 1 \circ 0^{x \text{ times}}$$

$$\text{code}_i(x) = \text{bin}(x) \circ \text{code}_{i-1}(|\text{bin}(x)| - 1) \text{ when } i > 0.$$

In this thesis, we are only interested in the first two codes in this recursion, code_0 and code_1, which we'll call c_l (the long code) and c_s (the short code), respectively. To illustrate the codes, $c_s(4)$ is equal to `100 100` and $c_l(4)$ is `1000`.

We will need the following lemma on the label size of the short and long code:

Lemma 2. *For a non-negative integer x, the length of $c_l(x)$ is $x + 1$ and the length of $c_s(x)$ is $2\lfloor \log x \rfloor + 2$.*

Proof. The length of c_s can be seen immediately from the definition of it, and c_l is composed of the binary representation of x (of length $\lfloor \log x \rfloor + 1$) and the length of such representation in c_l form, minus one, which is again precisely $\lfloor \log x \rfloor + 1$. □

2. Known results

2.1 Trivial bounds

There is a very simple adjacency labeling scheme for all trees of size n with labels of size $2 \log n$. We root the tree and then traverse it, assigning a unique non-negative identifier to each vertex. Then, for every vertex, we construct the labeling in this manner: inside the first $\log n$ bits we store the identifier of the vertex itself and in the following $\log n$ bits we store the identifier of its parent in the tree. For two vertices x, y we need only to compare the second number of x with the first number of y and vice versa. If any two numbers are the same, we consider the vertices adjacent. For the root, we just store the same number twice, so we can detect root immediately.

This test is clearly correct and every vertex was given a single number, therefore we have found a $2 \log n$ adjacency labeling scheme. This was first published by the seminal work of Kannan et al [2]. We can view this scheme as an upper bound of the size of the optimal adjacency labeling scheme for trees.

A simple lower bound can also be found: we can always view the binary labels as numbers, and since every vertex of a tree has a unique label, we need n different non-negative integers and so at least $\log n$ bits. There is currently no (asymptotically) better lower bound for trees. In fact, it is believed that this is the asymptotically correct size of the optimal adjacency labeling scheme for trees of size at most n.

2.2 Microtree/macrotree decomposition

In 2002, Alstrup and Rauhe introduced a simple adjacency labeling scheme for trees which uses $\log n + \mathcal{O}(\log \log n)$ bits [3]. They achieve this by using a preorder traversal of the tree (assigning increasing identifiers to vertices so that the root gets the smallest number and then we recursively traverse its children from left to right) and storing its identifier (the $\log n$ part) combined with a modified version of the **heavy/light decomposition**.

In their approach, the **heavy** edge is the edge from the root to the largest subtree induced by its children, and then defined recursively for all other vertices in the tree (ignoring its ancestors). All non-heavy edges are **light**. If there is more than one candidate for the heavy edge, we select one arbitrarily.

In [3], they use a labeling scheme storing for each v its traversal number along with three integers from the range $[0, \log n]$ – the number of light edges on the path from the root to v, the logarithm of the difference between the identifier of v and its parent and the logarithm of the difference between the identifier of v and its heavy child. This leads to an adjacency labeling scheme of the requested size.

The scheme works because if the light depth changes by one, the difference between the identifiers for non-adjacent vertices is noticable even on the logarithmic scale, mostly because of the increase by the heavy subtree. If the light depth does not change, the remaining logarithmic numbers must be equal in order for the vertices to be adjacent. For the precise inequalities see [3].

In the same paper, the authors also show how to use the aforementioned labeling along with a microtree/macrotree decomposition (for definition see [3]) to create a labeling for general trees which only uses $\log n + \mathcal{O}(\log^* n)$ bits. Even though it is very close to the optimum, it is expected that microtree/macrotree decomposition itself is not sufficient to reach the optimum value, as factors of $\mathcal{O}(\log^* n)$ are common in proofs using these decompositions.

It is important to mention that the $\log n + \mathcal{O}(\log^* n)$ scheme uses asymptotically more than a constant time for decoding, as we have to recurse into roughly $\mathcal{O}(\log^* n)$ subtrees.

2.3 Traversal and Jumping

In 2007, Bonichon, Gavoille and Labourel published a technique named "Traversal and Jumping" [1] which gives $\log n + \mathcal{O}(1)$ label lengths for several classes of trees, most notably caterpillars and binary trees. This is especially noteworthy as these are some of the few non-trivial classes of trees for which $\log n + \mathcal{O}(1)$ bound is known.

It has also been claimed that Traversal and Jumping can also be used on all bounded degree trees, which seems quite plausible, as one would follow a similar technique as was used for binary trees. However, the proof of this claim has not yet been published as of 2011.

2.4 Trees with bounded depth

Another class of trees for which a $\log n + \mathcal{O}(1)$ adjacency labeling scheme is known are trees with bounded depth d. In fact, this is a corollary of a theorem by Fraigniaud and Korman on ancestry labeling schemes [6]. An **ancestry labeling scheme** is very similiar to adjacency labeling in that it shares the necessity of injectivity and an encoding and decoding function, but the functions decode not adjacency, but the more general ancestry.

The result of Fraigniaud and Korman is an ancestry labeling scheme for all trees (and forests) of depth d with labels of size $\log n + 2\log d + \mathcal{O}(1)$. We can translate it to a $\log n + 3\log d + \mathcal{O}(1)$ adjacency labeling scheme simply by storing the depth itself inside the label. A parent can be defined then as an ancestor which has depth one less than the child.

It is important to mention that it has been shown that it is not possible to get an $\log n + \mathcal{O}(1)$ ancestry labeling scheme for all trees – in fact, if you require even parent and sibling queries for trees (so you can recognize which is which), you need labels of size at least $\log n + \Theta(\log \log n)$. Both of the results mentioned above are proven in [4].

2.5 Planar graphs

Another class of graphs for which an adjacency labeling scheme is often sought are **planar** graphs, graphs which can be drawn without crossings on the plane.

Since it is known [7] that every planar graph can be decomposed into a union of three forests (disjoint sets of trees), we can make use of this decomposition,

apply the straightforward labeling method and get the upper bound of $6 \log n + \mathcal{O}(1)$. If we make use of the fact that we can store only one identifier for all three trees, as opposed to having different ones for every tree, we get to the bound of $4 \log n + \mathcal{O}(1)$.

We can employ another theorem [8], which states that we can find a decomposition into three forests where one of the forests has bounded maximum degree by a constant, and get label lengths of $3 \log n + \mathcal{O}(1)$.

In [9], Gavoille et al. suggested a different labeling, which produces labels of size $2 \log n + \mathcal{O}(\log \log n)$. Their result is actually a specific case of a more general result on graphs with bounded treewidth – planar graphs do not have fixed treewidth, but we can decompose them into two graphs which are limited. Using the labels together as in the previous example, we get the coeficient 2 in $2 \log n$.

3. Traversal and Jumping

3.1 General method

"Traversal and Jumping" is a method of developing adjacency labeling schemes for several subclasses of trees. It is noteworthy because it is the first and (at the time) only technique for producing $\log n + \mathcal{O}(1)$ adjacency labeling schemes for caterpillars and bounded degree trees.

This method was developed by Bonichon, Gavoille and Labourel [1] in 2007. In general, we can describe this method in the following steps:

1. Set a fixed root and orientation away from the root.
2. For vertices v of the graph, define and compute intervals which contain labels for the children of v. Often, the actual interval sizes depend on interval sizes of the children, so we may have to traverse the graph in order to compute the interval sizes.
3. Create a suffix code C that encodes the interval sizes.
4. Traverse the graph again and assign to every vertex v a different positive number chosen from the right interval so that $C(v)$ can be decoded from this number.

The method treats the label assigned to v as an integer or as a bit string, whichever is better at the moment. We will therefore mix and match these two terms as well.

As we can see from Step 4 of the procedure, the choice of the code C may influence the interval sizes, so we will define the code first and compute the interval sizes afterwards.

In the following section, we will describe this technique on the class of caterpillars. We skip the case of binary (or bounded degree) trees, as it is much more technical, while the general ideas remain the same.

3.2 Caterpillars

As we recall from the introduction, a **caterpillar** is a simple tree consisting of a path (the **spine**) and vertices connected each to one element of the spine, called **legs**.

In this section, we will prove the original result of Gavoille and Labourel concerning $\lceil \log n \rceil + 6$ bounds on label sizes of caterpillars of size at most n. Our proof is slightly different and arguably less technical than the original, although it employs the same encoding techniques and basic ideas. Also, we fix some errors present in the original paper.

Formally, we will prove the following:

Theorem 1. *There exists an adjacency labeling scheme for the set of all caterpillars of size at most n, which uses $\lceil \log n \rceil + 6$ bits for the label length. Also, we can construct such labeling in linear time and decode the labels in time $\mathcal{O}(1)$.*

3.2.1 Step 1: Orienting the caterpillar

For caterpillars, we orient the spine as a path. The root will be one of the spine vertices that has only one spine neighbor. We name these vertices s_1 to s_k along the oriented path. Also, the edge between a spine vertex and a leg vertex is always oriented towards the leg vertex. Most of the information for adjacency will be stored in the spine vertices.

The leg vertices will be denoted $l_{i,j}$ for a j-th leg vertex of the spine vertex s_i. We will also denote d_i the number of leg vertices for a given s_i.

We will need to decide quickly whether a given label of a vertex is a leaf or not. Therefore, we spend 1 bit of the label (the first one) to encode whether a vertex is a leg vertex (then it has 0 set) or a spine vertex (then it has 1).

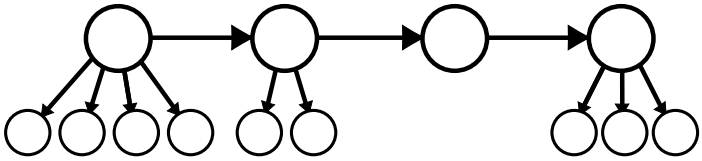


Figure 4. A caterpillar showing the suggested orientation.

3.2.2 Step 2: Defining the intervals

We will define two intervals for every spine vertex s_i, both of which together will hold all children of s_i, with respect to the orientation. The sizes of these two intervals will be stored inside the label of s_i. The intervals themselves will immediately follow s_i and these two intervals do not overlap each other, so we can easily check, given a label of another vertex, if such vertex is inside either of these intervals or completely outside. Intervals for different s_i may overlap each other, which we describe later.

The first interval will be called a **leg interval** of s_i and it will host all legs that are associated with s_i. Labels that will be associated with these leg vertices will be simply picked in the increasing order from this interval, and otherwise they will hold no information on their own.

On the other hand, the following spine vertex s_{i+1} must be stored in a larger interval by itself, because we need enough candidates for the label of s_{i+1}, so that the right information (in our case, the interval sizes of the following intervals) can be decoded from the label.

Therefore, we impose that in the second type of interval, called the **location interval** of s_{i+1}, only one spine vertex will be located. That does not mean that there are no other labels – there will be, as the following spine vertex also has the associated leg interval right after its own label.

In order to fulfill our requirement on the number of spine vertices present in one location interval, we must make sure that the following interval (which is a leg interval for s_{i+1}) ends after the end of the location interval of s_{i+1}. We solve this by actually making the size of the location and the leg interval of s_i exactly the same.

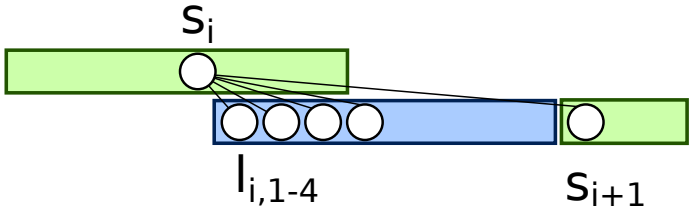


Figure 5. The location interval and the leg interval for s_i and a location interval for s_{i+1}. Note that the leg interval of s_i starts immediately after the set label for s_i and that it is of the same length as the location interval for s_i.

3.2.3 Step 3: The suffix code

We shall devise the suffix code so that for every spine vertex s_i, we can store inside this code two numbers: the size of the leg interval of s_i and the size of the location interval for s_{i+1} (the next spine vertex on the oriented path).

It would be very hard to encode the exact sizes of the intervals, therefore we will be more generous and keep only a fraction of the information. Precisely said, we will set both intervals to be as large as some power of 2, while storing only the logarithm of the interval size.

Also, since we use a suffix code, we cannot expect to have both efficient encodings for very small numbers and efficient encodings for larger numbers. Roughly said, the shorter codes (in terms of bit length) we use for the smaller numbers, the longer the code gets when the integers increase.

With this in mind, we will choose a granularity for the interval sizes. The intervals for s_i will be of size 2^{t_i+3}, where $t_i \geq 0$. Therefore, the smallest interval we can allocate will be of size $8 = 2^3$. As we described before, we will need to store two numbers in the label for every s_i: the size of the leg interval for s_i and the size of the location interval for s_{i+1}.

The reader should remember that, when we defined the intervals in Step 2, we noted that we will set the size of the location interval of s_i and the leg interval of s_i to be the same size, so we will not get into trouble with decoding the adjacency of the spine vertices. This means that t_i will denote the size of two intervals: the leg interval of s_i and the location interval of s_i.

However, these two interval sizes are stored in different labels: the size of the leg interval of s_i is stored inside the label of s_i, while the location interval for s_i must be stored within the label of s_{i-1}. This means that every spine vertex will store two integers: (t_i, t_{i+1}).

Now that we have ascertained that we need to store the pair (t_i, t_{i+1}) for every spine vertex s_i, we need to figure out which suffix code to use. Since the length of the location interval for s_i is equal to 2^{t_i+3}, the code should use at most $t_i + 3$ bits, as we cannot store more information using a suffix code inside an interval of this size.

Recall that we defined two codes c_l and c_s, the latter of which is a suffix code that uses $2\lfloor \log x \rfloor + 2$ bits for encoding a number x, which should be more than enough for our needs – we have $t_i + 3$ bits, while c_s uses only two times the logarithm of t_i plus two. The authors of the original article suggest that we can pad the rest of the code with the longer code (minus the length of the shorter

code), so that the length of the complete code is exactly $t_i + 3$:

$$C(s_i) = c_l(t_i + 3 - |c_s(t_{i+1})|) \circ c_s(t_{i+1}).$$

We will need to work closely with variables t_i in the proof, so we restate the conditions we have on t_i: the c_l code above can encode only non-negative numbers, t_i must be large enough to store all leg vertices of s_i and it has to be at least 0:

$$t_i = \max\{|c_s(t_{i+1})| - 3, \lceil \log d_i \rceil - 3, 0\}$$

3.2.4 Problem with the code

However, at this point we encounter a problem which is not treated in the referenced article and leads to the incorrectness of the proof given there. If we look closely at the size of $C(s_i)$, we find out that it is actually $t_i + 4$ bits long, because of the leading 1 in the code c_s. The authors of the article claim that we can find a number representing $C(s_i)$ in an interval of size 2^{t_i+3}, but that is not always the case.

A straightforward approach would then be to actually make the intervals one bit larger and increase the final calculations (which we will make later) by a factor of 2. This is possible, but it leads to a labeling of size $\lceil \log n \rceil + 7$, not $\lceil \log n \rceil + 6$ as originally claimed.

We will however fix the code in the following way, which will keep the length $\lceil \log n \rceil + 6$: if we decode as t_i any number besides 0, we will add 1 to the t_i which was decoded (so we will expect larger intervals), but if we decode 0 as t_i, we treat it the same way it was before. The numbers t_i and the code will be the same as before, but we have changed the interval calculation: instead of 2^{t_i+3}, we will now count $2^{t_i+3+[t_i \geq 1]}$.

The reason for this change is because we must be careful with the spine vertices which have 0 leg vertices attached. Both in the new code and in the original code, the pairs $(0, 0)$ and $(0, 1)$ are encoded using only 3 bits, which will be important later on.

3.2.5 Encoding and decoding

The following section deals with the correctness and complexity of encoding and decoding the pair (t_i, t_{i+1}) to/from an integer that is assigned as a label to s_i.

We will assume that we are working in a Word-RAM computational model. This means we can access word-sized memory register in constant time and apply "traditional" arithmetic and binary operations on them. We also assume all of our numbers fit into one word or a constant number of words. For a precise definition of the RAM model, see e.g. [5].

The following lemma shows that it is possible to decode and encode the prefix code $C(s_i)$ into a location interval, even in constant time:

Lemma 3. *Given two integers (t_i, t_{i+1}) and an integer x, we can decode and encode these numbers using the code $C(s_i)$ from/into an interval $[x, x+2^{t_i+3+[t_i \geq 1]})$ in $\mathcal{O}(1)$ time.*

Proof. We will split the proof into two parts: encoding and decoding.

Encoding. To encode the pair, we will first transform them into the code $C(s_i)$. First, we calculate the value of $|\text{bin}(t_{i+1})| - 1$, which we can do in $\mathcal{O}(1)$ (we are given t_{i+1} in binary as input, so we know its length, or we can calculate it using MSB). Then, we take t_{i+1} in its binary representation, append 1 and shift the entire number to the left by $|\text{bin}(t_{i+1})| - 1$ places, thus completing the encoding of $c_s(t_{i+1})$.

To encode the $c_l(t_i + 3 - |c_s(t_{i+1})|)$, we need to add one more 1 to the correct place, the computation of which takes again only constant time. Adding the one, we have successfully encoded (t_i, t_{i+1}) into the code $C(s_i)$. We now also know the length of $C(s_i)$: $t_i + 4$ or $t_i + 3$, depending on t_i. We will denote the length of the code by l.

We can also see that if t_i equals zero, the only pairs (t_i, t_{i+1}) where the first number is zero are $(0, 0)$ and $(0, 1)$, as any higher t_{i+1} would cause the t_i to be at least 1. The code for $(0, 0)$ is `110` and the code for $(0, 1)$ is `111`, so these cases can be safely encoded inside the interval of 2^3 numbers.

In order to encode the code into a positive number from the interval, we simply find the smallest larger number to x which contains $C(s_i)$ as a suffix. This number is surely within the $[x, x + 2^l)$ interval, because in this sequence we can see all possible binary suffixes of length l.

The smallest number which contains $C(s_i)$ as a suffix can be computed again very fast, as we already know the length l. We replace the l least significant bits with our code $C(s_i)$. Using this operation, we may arrive at a number smaller than x. If this is the case, we need to put it back into our interval, so we simply add 1 to the partpreceding our encoded suffix. Since we were originally below x, by this operation we could not have gone above $x + 2^l$.

Decoding. To decode the string, we will first use LSB (least significant bit operation) to decode the value $|\text{bin}(x)| - 1$. After we decode this number, we can easily take the preceding $|\text{bin}(x)|$ bits and decode t_{i+1}. To decode t_i, we shift the entire c_s part of the code to the right and then apply LSB again. The position of the first 1 from the left encodes t_i again according to the definition of c_l.

Besides MSB and LSB, we have used only basic arithmetic and shifts, which all can be done in $\mathcal{O}(1)$ time. The fact that LSB and MSB can be done on Word-RAM in $\mathcal{O}(1)$ is a non trivial one, for the proof see [5]. □

The complete decoding process, given two labels label(u) and label(v) can be described in this manner:

1. Strip the first bit from both labels. If both bits are equal to zero, the vertices are not adjacent. Otherwise, continue the check. Assume now that $u = s_i$.

2. Decode t_{i+1} and t_i from label(s_i) as explained in Lemma 1. Do the same for v, if it is a spine vertex as well.

3. Calculate the interval sizes from t_i using the formula $2^{t_i+3+[t_i \geq 1]}$.

4. If v lies in the interval $(\text{label}(u), \text{label}(u) + 12^{t_i+3+[t_i \geq 1]}]$ and v is a leg vertex, declare u and v to be adjacent.

5. If v is a spine vertex and either v lies in $[\text{label}(u) + 2^{t_i+3+[t_i\geq 1]}, \text{label}(u) + 2^{t_i+3+[t_i\geq 1]} + 2^{t_{i+1}+3+[t_{i+1}\geq 1]})$ or vice versa, declare u and v to be adjacent.

6. Otherwise, declare them to be non-adjacent.

3.2.6 Step 4: Computing the labels

Now that we have labels and set the dependencies, we can move on to computing the integer labels, given a specific caterpillar. To do this, we observe that the last spine vertex s_k does not need to store any location interval at all, and thus will have t_{k+1} set to zero. The size of its own leg interval therefore depends only on the number of legs, and we can see that the rest of the interval sizes will be defined by induction, going backwards along the oriented path of the spine.

To calculate the labels, we will traverse the caterpillar twice – the first traversal will work backwards along the oriented path, calculating the values of t_i. In the second traversal, we will go along the path of the spine, and "laying down" the intervals one by one, starting from 0, always picking the label of the vertex s_i based on Lemma 1.

For example, if we calculated t_1 to be 0 and t_2 to be 1, we will first start with the interval $[0, 8)$ and pick the right integer encoding $C(s_1)$ as described in the lemma. Then, we move on to the interval $[8, 16)$ and put all the leg vertices of s_1 one by one into that interval (so the $l_{1,1}$ will have label 8, $l_{1,2}$ label 9 and so on). Afterwards, we read t_2 and look at the interval $[16, 58)$. This is the location interval for s_2, so we choose the right number for encoding $C(s_2)$ and continue onwards.

3.2.7 Label length

In order to conclude our proof of Theorem 1, we need to estimate the maximum size of the labels for a caterpillar of size n.

Because of the way Traversal and Jumping assigns intervals to every spine vertex and intervals to a set of leg vertices, we will estimate the maximum label length by the label of the last leg vertex.

Since the intervals are without gaps, and the label of the last vertex is inside the last interval, we can estimate the label length by (the logarithm of) the sum of the interval sizes, provided we count every interval indicated by t_i twice – the first time it is a leg interval for s_i, then a location interval for s_{i+1}. Also note that we need a location interval even for the first vertex, s_1. We also must not forget about the extra bit of information we use on non-zero intervals and about the bit we use for deciding whether the vertex is in the spine or not.

If we assume the last (leg) vertex is named $l_{k,m}$, we have:

$$\text{label}(l_{k,m}) \leq \sum_{i=0}^{k} 2 \cdot 2^{t_i+3+[t_i\geq 1]}.$$

The accounting method

The logical step now is to bound the size of $2^{t_i+3+[t_i\geq 1]}$. However, it will not be as simple. If we look at the definition of t_i as stated above:

$$t_i = \max\{|c_s(t_{i+1})| - 3, \lceil \log d_i \rceil - 3, 0\},$$

we can see that sometimes the size of t_i is set by the size of t_{i+1}, and we cannot bound it at all by the number of the associated leg vertices, d_i. The following picture illustrates the idea:

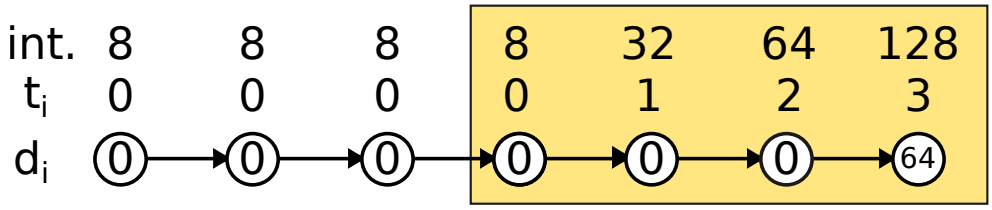


Figure 6. A potential situation which needs to be solved using the accounting method. Inside the vertices are the leg counts d_i. Above them are the numbers t_i for each vertex. On top are the interval sizes for each s_i. We calculate the interval sizes using the formula $2^{t_i+3+[t_i\geq 1]}$. Note that the small vertices inside the rectangle could not estimate their lengths by their d_i, but summed together, the total length of the intervals is less than 4 times the d_i of the last one.

Therefore, we have to somehow "redistribute" the size of t_i to the later spine vertices, which actually caused the increase in size for t_i. We will borrow some terms from economy: we say that a vertex s_j **pays** for vertex s_l, $l \leq j$, if we attribute the interval length of $2^{t_l+3+[t_l\geq 1]}$ to the vertex s_j. The idea is that s_j could have enough leg vertices, so that it can cover not only its own intervals, but also the intervals of other vertices. This will be shown to be true.

It is interesting to note that this method of proving inequalities and other mathematical statements comes from algorithm theory, namely amortized complexity.

Our goal will be the following lemma:

Lemma 4. *With t_i defined as before, the following inequality holds:*

$$\sum_{i=0}^{k} 2^{t_i+3+[t_i\geq 1]} \leq 8 \sum_{i=0}^{k} \ulcorner d_i + 1 \urcorner.$$

Proof. We will try to estimate $2^{t_i+3+[t_i\geq 1]}$ for all vertices. We will split the proof into three cases, based on which part of the definition is being the maximum at the moment. First, we solve two cases which are able to pay for their own intervals:

Case 1: the spine vertex has at most 8 legs and the previous location interval does not need many bits. Formally, $t_i = 0$ and $t_{i+1} \leq 1$. The interval size is then $2^3 = 8$, which is less than $8\ulcorner d_i + 1 \urcorner$. Notice that we had to keep these intervals only 3 bits in length, otherwise the inequality would break for vertices with no legs.

Case 2: the spine vertex has enough legs, so that $\lceil \log d_i \rceil - 3$ defines the interval size. Formally, in this case $d_i \geq 8$ and $|c_s(t_{i+1})| < \lceil \log d_i \rceil - 3$.

The interval size is therefore bounded by $2^{\ulcorner d_i \urcorner}$, which is always smaller than $2^{\ulcorner d_i + 1 \urcorner}$. The coefficient 2 appears because of our $[t_i \geq 1]$ factor which we introduced in order to correct the original proof.

To sum up, we have that in this case, 2^{t_i+3+1} is bounded by $2^{\ulcorner d_i + 1 \urcorner}$.

Case 3: the location interval for s_{i+1} needs more information than the legs of s_i, and the size of t_i becomes dominated by that information. Formally, $t_{i+1} \geq 2$ and $|c_s(t_{i+1})| > \lceil \log d_i \rceil - 3$, which leads to $t_i = |c_s(t_{i+1})| - 3$.

In this case, we need to use the accounting method as described earlier. We will move the cost of this t_i to the first s_k (with k larger than i) which belongs to a case other than 3. We can deduce that the method is correct from the following observations:

- **There is always a vertex which can be used for the transfer.** The following vertex is either of case 2, which is what we look for, or of case 3, in which case we use induction to prove this claim. The last vertex is always of type 1 or 2, so the induction argument is correct.

- **The vertices of case 1 or case 3 are never recipients of the cost.** If the vertex preceding a vertex of case 1 enters case 3 itself, it has to be because the preceding label length was too large – which means the vertex had to be of case 3 or case 2.

- **The increased cost is at most twice the interval size of the receiving vertex.** We will look at the increased cost at the point where all the vertices have finished the accounting. We can see that if a vertex enters case 3, the following inequality holds:

$$2^{t_i+3+[t_i \geq 1]} = 2^{|c_s(t_{i+1})|+1} = 2^{2(\lceil \log(t_{i+1}+1) \rceil)+1} \leq 2^{t_{i+1}+3+1}/2.$$

  This inequality holds for every $t_i \geq 1$, because it can be rewritten as $2\lceil \log(t_i + 1) \rceil \leq t_i + 2$, which holds for small t_i by manual verification and for larger ones by asymptotics of the left and right sides.

  That means the interval size is halved from the following vertex by and if the vertex that pays the cost is l vertices away, the interval size is divided by 2^l. Since $\sum_{i=1}^{r} p/2^i \leq 2 \cdot p$, we have increased the amount allocated to the recipient by a multiple of 2, and since the recipient's interval estimate was shown to be $2^{\ulcorner d_i + 1 \urcorner}$, we will only double this amount and get the estimate for vertices of Case 2 as $4^{\ulcorner d_i + 1 \urcorner}$.

By this calculation we get all the possible bounds: the small vertices have 2^{0+3} bounded at most $8^{\ulcorner d_i + 1 \urcorner}$, spine vertices with enough legs pay have both their cost and the cost of the preceding smaller vertices bounded by $4^{\ulcorner d_i + 1 \urcorner}$ and the vertices which get dominated by larger vertices are all absorbed in the costs of larger ones. Thus:

$$\sum_{i=0}^{k} 2^{t_i+3+[t_i \geq 1]} \leq 8 \sum_{i=0}^{k} {}^{\ulcorner} d_i + 1 {}^{\urcorner}.$$

□

We conclude our original argument, that the label length of the last vertex can be bounded by:

$$\text{label}(l_{k,m}) \leq \sum_{i=0}^{k} 2 \cdot 2^{t_i+3+[t_i \geq 1]} \leq 2 \sum_{i=0}^{k} 8 \cdot \ulcorner d_i + 1 \urcorner \leq 16 \sum_{i=0}^{k} \ulcorner d_i + 1 \urcorner \leq 32n = 2^5 n.$$

We add one bit to decide whether the vertex is a spine vertex or a leg vertex, and we get that we have a $\lceil \log n \rceil + 6$ adjacency labeling scheme for caterpillars of size at most n, which is what we set out to prove.

3.2.8 Improving the bound

Originally, the goal of our work was to use this accounting method to improve the bound of the article by at least one. This seemed to be possible, as in the original proof it could be seen that the large vertices only pay $2\ulcorner d_i + 1 \urcorner$, which could be increased by a multiple of 2 and keep within the desired bound $4\ulcorner d_i + 1 \urcorner$.

The problematic spine vertices for which the bound is $8\ulcorner d_i + 1 \urcorner$ and not just 4 are the ones with no associated legs at all. In the code C, we use the codes `110` and `111` for these, which require location intervals of size 2^3. Since there can be caterpillars which consist of many such vertices with no vertices for transferring the cost, we would need to create a better suffix code.

One of the ideas would be to change the granularity and encode the new $(0, 0)$ (only for spine vertices which have up to 3 legs) as `10` and $(0, 1)$ as `11` and then deal with the increase in code length for the larger vertices which can afford it. However, this technique is probably impossible with the proof given above, as there are no larger vertices which could bear the increased cost of the suffix code – every vertex that is not of Case 3 is paying for the full amount that would be allowed were the 8 constant changed to 4: $4\ulcorner d_i + 1 \urcorner$.

We have been unable to improve the constant mainly because we had to spend the possible increased cost of the larger vertices on the correction of the original proof.

4. Small universal graphs for trees

As we noted in the first part of the thesis, every adjacency labeling scheme can be thought of as a special case of an induced-universal graph. If the scheme produces labels of size l, the resulting universal graph will be of size 2^l. While the conjecture of an existence of $\log n + \mathcal{O}(1)$ labeling is still open, we have employed an algorithm that generates induced-universal graphs for small values of n, the number of vertices of the trees.

Using this method a lower bound for the constant in the hypothesised $\log n + \mathcal{O}(1)$ could be found, provided we show that there exists an n such that all universal trees for such n have size larger than $2^{\log n} = n$. However, the sheer amount of graphs required to check makes searching for the precise constant very difficult.

In order to skip at least some of the graphs, we check only connected graphs of size n. It can be easily seen that every disconnected universal graph (for a class of connected graphs) can be extended with additional edges until it is connected, while preserving the universal property.

4.1 Description of the algorithm

At its heart the algorithm is an exhaustive search from the set of all possible graphs of size n. Nonetheless, the algorithm has to employ non-trivial methods to provide fast operations that are needed – namely, it has to quickly decide whether the given graph is universal.

Instead of checking that every tree of size k is an induced subgraph in this graph, we will do the converse operation – we will generate all trees of size k from a given universal candidate C, then quickly sort these and remove isomorphic ones. In the end, we will compare the number of non-isomorphic trees induced in C with the number of non-isomorphic trees of size k and return the boolean result.

This has the advantage that the candidate graphs themselves can be generated in advance and the checking itself can be parallelized on several machines if need be.

The algorithm goes through the following steps:

1. We generate all non-isomorphic graphs C for a given n.
2. For a fixed tree size k, we check all k-vertex-subsets of a given graph C.
3. We filter these subsets and keep only those that define a tree.
4. We save all the generated trees for one C.
5. We centralize the trees in C, separating them into two groups. This is done so the following step is more straightforward.

6. We use bucket sort to calculate the number of isomorphism classes of the trees.

7. We add the two numbers and compare them to the number of all non-isomorphic trees of size k.

4.1.1 Generating trees

Verification

The first step is to generate all induced subgraphs/trees of size k from a given candidate C. Usually those are called induced **subtrees**. There can be exponentially many non-isomorphic induced subtrees in some candidates, so we do not focus much attention at the finest points of the generation. Nonetheless, we do try to make it reasonably fast.

We choose every k-tuple of vertices of C and for the specific tuple we only consider the induced graph in C. For convenience reasons, we will copy the k-tuple and the edges it induces as a separate graph. We will also gain faster tree checking in the following steps. The copying step will however take $\mathcal{O}(k^2)$ time, as we have to create new adjacency lists.

It is wiser to consider k-tuples of vertices instead of every $(k-1)$-tuple of edges, as the candidate C can have quadratically many edges.

Next, we must verify that the current tuple is a tree (we will call the current induced subgraph I). This verification can be done using a depth-first search. This search simply traverses the vertices of I, starting from an arbitrary one, and marks the ones it has already visited. Two different routes to a vertex imply a cycle inside the induced subgraph, while the number of visited vertices at the end can be used to check connectivity. A depth-first search will take time $\mathcal{O}(k)$, even though the k-tuple may not be a tree. This holds because after we checked $(k-1)$ edges we know whether the k-tuple induced a tree or not.

Centralization

If I is verified to be a tree, a copy is created and exported for the next phase of the algorithm. However, the next phase needs to process trees which are "standard" in some sense – namely, we need to have a rooted tree without arbitrarily choosing the root vertex. For this the *centralization* of a tree is applied.

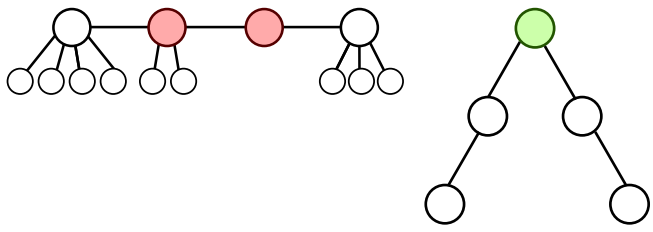


Figure 7. The caterpillar on the left has no uniquely defined center, but the tree on the right has.

Centralization is a simple, linear time algorithm to find a vertex (called a **center**) with a property that the longest path from it to some leaf is shortest among all possible centers. Centers of trees are the best choices for rooting such

trees, as they are often uniquely defined and so we have an easier time when working with isomorphism classes.

A center is "almost always" uniquely defined, as the lemma states:

Lemma 5. *The center of a tree is uniquely defined unless there is an edge such that both vertices on that edge have the same longest distance from them to a leaf.*

Proof. If the graph is only a single vertex, the center is uniquely defined, if it is only one edge, the two vertices have the same longest distance to a leaf (to each other). So we may assume the tree has at least some non-leaf vertices.

Now, we observe than when considering a leaf connected to a non-leaf vertex, the leaf is never the right choice for a center, because every path must cross the inner vertex through which the leaf is attached.

We can therefore remove all the leaf vertices and if we find a center of the new, reduced tree, we can add all the leaf vertices back and notice that every path to the leaf has now been increased by one, therefore the center of the reduced tree is the best candidate for a new center.

By removing all the leaves, we could either end with a single remaining vertex (which must be the center) or with an empty graph. In the second case, before this step we removed two leaves connected to each other – but after returning all the removed vertices we note (by induction) that they have the same maximum distance to a leaf, which means that we cannot choose the center uniquely. □

As we noted in the proof of the lemma, the only case where the center (our new root) is not uniquely defined is where the two candidates share an edge. Since the next step does not need to count all the trees in one pass, we can split the exported trees into two groups: with a unique center and without it. The trees without it can then be converted to trees with a center using subdivision of the problematic edge. We lose the information about the tree structure, but we will not need it in the following step.

Centralization can be also easily achieved by a linear-time algorithm, inspired by the proof of the lemma: We mark every vertex with a number that corresponds to the step at which it would be removed from the graph using the reduction algorithm.

- First, we mark every leaf with the number 1 and add their neighbours to a queue.
- Then, we go through the queue and look for all the new leaves. Some elements of the queue are not yet leaves, but we know that every new leaf must be created by removing a previous leaf, so all new leaves are somewhere in the queue.
- We mark every new leaf by the number 2 and treat them as removed vertices, which means adding their neighbors to the queue.
- We repeat the process until all the vertices are marked with a number.
- Then, either two vertices have the same number (and we subdivide the edge between them), or there is a vertex with the highest number, which must be the center because of the correctness of the lemma.

The running time is $\mathcal{O}(k)$ for a tree of size k, as we can observe that every edge triggers at most one addition to the queue, and the number of edges is $k-1$.

We have therefore centralized/rooted the trees and sorted them into two groups, one with trees of size k and one with trees of size $k+1$.

4.1.2 Sorting trees

In the second part, we take the set of the trees of size k (or $k+1$), many of which are isomorphic, and decide how many isomorphism classes there are in the set.

We will achieve this by using a special version of the sorting algorithm named "Bucket sort". We will assign every non-isomorphic subtree a special number, while keeping track of how many numbers we have assigned. At the end of the algorithm, the number of different numbers we have assigned to the roots of the trees will equal the number of non-isomorphic trees in that set.

There are many algorithms for canonically assigning a positive number to every isomorphism class of trees. We will, however, make use of the fact that we do not need to store or decode the numbers – all we care about is the number of isomorphism classes. We will, therefore pick numbers in the increasing order, only making sure that every member of an isomorphism class gets the same number.

The bucket sort is executed in several stages. First, we sort the subtrees by depth. This can be done by doing a depth-first search on a tree and then putting the subtrees into buckets once their depth becomes known. For the rest of the algorithm, we work only with trees with (roots of) the same depth d, going from the shallowest (depth 0) to the deepest.

For depth 0, we can assign every vertex with the same identifier, as one-vertex subtrees are all isomorphic. So we can suppose the depth is now 1 or more.

At the beginning of the stage, we take all subtrees of depth d and put them in a list. Next, we can look at every subtree from that list as a tree where every vertex has been already assigned a correct identifier, not counting the root. Our task is to assign a number to the root such that every two isomorphic trees from our list get the same number.

It is important to note that subtrees can be isomorphic only if they have the same depth, so we do not miss any classes by working in phases by depth.

Next, for every tree on the list, we sort (using bucket sort in $\mathcal{O}(k)$ or even a slower sort, as k is very small) the children of the root according to their already assigned identifiers (they have lower depth than the root by definition, so we already assigned identifiers to their subtrees). It is not important in which order we sort these subtrees, as the numbers actually mean very little to us.

Since the children of every root are sorted by their assigned identifiers, we will consider the tuple of the identifiers and apply the bucket sort iteration for every position in the tuple.

The bucket sort therefore takes the list L, goes (at step z) through the z-th element in the identifier tuples and puts every tree from the list into the bucket corresponding to the identifier on the z-th position in the tuple.

We maintain a list of the buckets which are currently non-empty, so we can access them in linear time for emptying – going through all the buckets sequentially would take too much time.

After the list has been depleted, we traverse all the buckets, emptying them

and putting all trees back into the list L – preserving the order as we would do in a regular bucket sort.

If, at this point, a tree runs out of elements in the tuple (say in the third iteration there is a tree with degree 3 and yet some other tree with degree 7), we can safely assign to it (and every other root with the same tuple) a new identifier and remove it from the procedure.

Because of the way the bucket sort is operating, we know that trees with the same tuples (both in size and in identifiers) will be one after another in the designated bucket, so labeling all isomorphic trees the same simply reduces to checking if the last tree in the bucket had the same tuple as the current one.

With the rest of the unfinished subtrees in L, we repeat the refilling/emptying procedure until every subtree of this depth has been assigned an identifier. We can now return and assign numbers to trees of higher depths, as their subtrees of lower depths were already assigned an identifier.

Please note that the only assigned identifiers which are of interest to us are identifiers which are assigned not just to subtrees, but to the roots of the input trees, so we separately store the information about how many different identifiers are assigned to the roots of the input trees.

After we run through all the phases and all the possible depths, we sum up the number of non-isomorphic trees from this algorithm with the number of all isomorphic trees from the second run, which we have launched on the subdivided trees. Comparing this number to the number of all non-isomorphic trees of size k gives us the confirmation or rejection of this candidate as a potential induced-universal tree.

4.1.3 Complexity

In the first step, the most time-consuming factor is checking every k-tuple. The check will be done in $\mathcal{O}(k^2)$ as we need to create a copy of the induced subgraph or work with the adjacency lists of the candidate C. Multiplied together, we get $\mathcal{O}((n/k)^k \cdot k^2) = \mathcal{O}(n^k/k^{k-2})$.

Centralization is done using a linear time algorithm, and is applied only on trees, so it takes $\mathcal{O}(k)$ time.

Computation of the isomorphism classes surprisingly takes only linear time in terms of the total size of trees on input. The first part (creating a separate pointer to every subtree, sorting subtrees by depth) is linear. Every subtree enters the bucket sort procedure exactly once and stays inside the empty/refill process for z rounds, where z corresponds to the degree of the root. Therefore, it will be checked once for every edge oriented away from the root of the subtree and so, every edge will be checked at most once.

Since there are $k-1$ edges in a tree of size k (or k edges in a tree of size $k+1$), we get that the total worst case complexity is $\mathcal{O}(k \cdot T)$, where T is the number of trees that were generated from the candidate graph C.

For larger values of n, there may arise an issue with the memory complexity of the bucket sort algorithm, as we need in theory as many buckets as $\sum_{i=0}^{n} T(i)$, where $T(i)$ is the number of non-isomorphic oriented subtrees of size i. We get this factor because unlike the roots of the trees, the subtrees are not centralized and can therefore contain more non-isomorphic trees than general unoriented trees.

However, this issue has not been a problem for the smaller values of n that we checked. The more acute problem is the sheer number of non-isomorphic graphs to check, which raises exponentially and for $n = 10$ it amounts to 11716571.

4.2 Implementation

We have implemented our algorithm in two separate executable programs called `treegen.c` and `treecomp.c`. The first generates all the induced trees of given size from a candidate universal graph, and the other does the sorting and comparing of the list of trees. Both of those programs are written in the C programming language, version C99. They are launched together using a Perl script.

In order to generate all non-isomorphic graphs of size n in order to use them as universality candidates, we employ Brett McKay's computer program toolset **nauty**, which is available online at `http://cs.anu.edu.au/~bdm/nauty/` .

The tree sizes can either be calculated from **nauty** or found in the On-Line Encyclopedia of Integer Sequences at `http://oeis.org/`.

4.3 Results

The following table denotes the number of non-isomorphic connected induced universal graphs for trees of size exactly k.

Tree size k	4	5	6	7	8
No. of classes	2	3	6	11	23
Cand. size: 5	2	0	0	0	0
6	42	0	0	0	0
7	593	18	0	0	0
8		1277	0	0	0
9			66	0	0
10				0	0

From the table we can see that we need at least $\lceil \log n \rceil + 1$ bit labels to store all trees of size 7. It is hard to interpolate the results as the amount of candidates to check dominates the running time of the algorithm.

Conclusion

Conjecture 1. *There exists an adjacency labeling scheme for all trees of size at most n that employs labels of length at most* $\log n + O(1)$.

The chief conjecture of the adjacency labeling schemes for trees, stated in a general form in [2] is still open and seems to be rather difficult, given the tight bounds around it ($\lceil \log n \rceil + 1$ from below, $\log n + \mathcal{O}(\log^* n)$ from above).

While Traversal and Jumping methods may still be improved in the future, especially for the bounded degree graphs, where the proof is rather technical, it seems that this technique itself will not be enough to create labels for all trees. Traversal and Jumping has the most difficulty when subjected to trees with variable degrees, where suggested interval sizes may fluctuate wildly and it is hard to encode every number efficiently inside the code.

However, we remain optimistic and hope that the conjecture will be answered in the affirmative, perhaps by creating an improved version of the methods that Traversal and Jumping employs.

Referenced literature

[1] Bonichon, Nicholas and Gavoille, Cyril and Labourel, Arnaud. *Short Labels by Traversal and Jumping.* Electronic Notes in Discrete Mathematics, pages: 153-160, 2007.

[2] Kannan, Sampath and Naor, Moni and Rudich, Steven. *Implicit Representations of Graphs.* Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, p. 334-343, 1988.

[3] Alstrup, Stephen and Rauhe, Theis. *Small Induced-Universal Graphs and Compact Implicit Graph Representations.* 43rd Annual IEEE Symposium on Foundations of Computer Science, pages 53-62. IEEE Computer Society Press, 2002.

[4] Alstrup, Stephen and Bille, Philip and Rauhe, Theis. *Labeling Schemes for Small Distances in Trees.* SIAM Journal on Discrete Mathematics, 19(2): 448-462, 2005.

[5] Thorup, Mikkel. *Undirected single source shortest paths with positive integer weights in linear time.* Journal of the ACM (JACM), 46(3): 362-394, 1999.

[6] Fraigniaud, Pierre and Kormand, Amos. *Compact Ancestry Labeling Schemes for XML Trees.* 21st ACM-SIAM Symposium on Discrete Algorithms, 2010.

[7] Schnyder, W. *Embedding planar graphs on the grid.* 1st ACM-SIAM Symposium on Discrete Algorithms, pages 138-148, 1990.

[8] Gonçalves, D. *Covering planar graphs with forests, one having a bounded maximum degree.* Electronic Notes in Discrete Mathematics, 31: 161-165, 2008.

[9] Gavoille, Cyril and Labourel, Arnaud. *Shorter Implicit Representations for Planar Graphs and Bounded Treewidth Graphs.* ESA 2007, LNCS 4698, pages 582-593, 2007.

[10] Babai, L. and Chung, F. R. K. and Erdős, P. and Graham, R. L. and Spencer, J. *On graphs which contain all sparse graphs.* Ann. Discrete Math, 12: 21-26, 1982.

[11] Chung, F. R. K. *Universal Graphs and Induced-Universal Graphs.* Journal of Graph Theory, 14(4): 443-454, 1990.