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 concen-

trate 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.

Notations

$\lceil p \rceil$: p rounded up to the nearest power of two.

$\lceil q \rceil$: q rounding up to the nearest integer.

$[p(x)]$: 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.

Preliminaries

A *Graph* $G(U, V)$ is defined as a set V of elements V_j : $j = 1, 2, 3, ..., n$ which can be represented as points and a set U of pairs (v_j, v_k) elements of V which can be represented as arcs joining points of V . T he elements of V are called Vertices and the elements of U are called Arcs, denoted by u_1, u_2, u_n. The

graph with directed arcs is called a *Directed Graph*. If U and V are finite sets, then the graph is called a *Finite Graph*. The directed arc $u_i = (v_j, v_k)$ is said to be *Incident from* or *Going from* v_j and *Incident to*or *Going to* v_k. v_j is called the Incident Vertex and v_k is the *Terminal Vertex*. The sequence of arcs of a graph such that every intermediate arcu_k has one vertex common with the arc u_{k-1} and u_{k+1} is called a *Chain*. A chain becomes a *Cycle* if the sequence of arcs, no arc is used twice and first arc has a vertex common with the last arc and this vertex is not common with any intermediate arc. A *Path* is a chain in which all the arcs are directed in the same sense such that the terminal vertex of the preceding arc is the initial vertex of the succeeding arc. A path is a chain but every chain is not a path. A *Circuit* is a cycle in which all the arcs are directed in the same sense. A graph is said to be *Connected* if for every pair of vertices there is a chain connecting the two.A *Tree* is a connected graph with atleast two vertices.

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. 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.

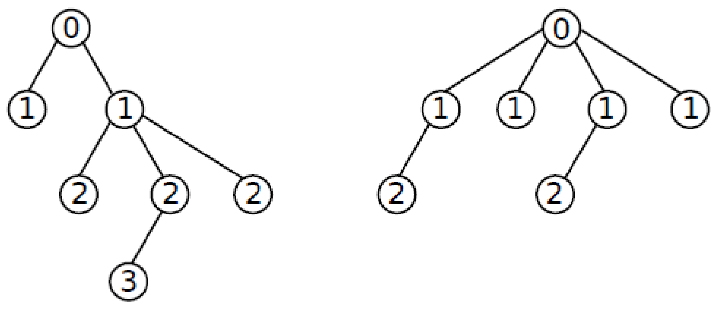

Figure 1: Two rooted trees with depth inside the vertices. Note that the two trees are isomorphic if viewed as unrooted trees, but the choice of the root changes the parent relation.

A *Chordless Path* in a graph G is a path for which no two vertices are connected by an edge that is not in the path. Chordless paths are also known as induced paths. A *Caterpillar Graph* is a tree in which the removal of all pendant vertices results in a chordless path.

A vertex with degree one is called a *Pendent Vertex*. For a non-negative integer x, $bin(x)$ will denote its binary representation. When working with binary strings, we will denote their concatenation as xy. The *Iterated Logarithm* $\log^* n$, a function growing asymptotically slower than any chain of logarithms with a

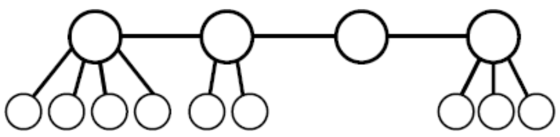

Figure 2: A caterpillar with 4 spine vertices and 9 leg vertices. We can create an isomorphic caterpillar with 5 spine vertices and 8 legs, or an isomorphic one with 6 spine vertices and 7 legs.

fixed length, is defined as

$$log^{*}\ n = s$$

if and only if we need to iterate the logarithm operation *s* times, starting from *n*, until we get a negative number. Iterated Logarithm or $log * n$ is the number of times the logarithm function must be iteratively applied before the result is less than or equal to 1.

$$\log^{*} n = \begin{cases} 0 & n \leq 1 \\ 1 + log^{*}(logn) & n > 1 \end{cases}$$

The iterated logarithm is an inverse function to the tower function

$$\mathrm{T(k)} = 2^{2^{2^{\cdot ... k\ times}}}$$

An expression that returns the Boolean value (or representation of a Boolean value) that is to be evaluated by the function is called a *Boolean Expression*. A *Boolean Predicate* returns the truth value of a Boolean expression. A *Labeling* is a function $\lambda : V \to \{0, 1\}^{l}$.

1. 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.

Definition 1.0.1. : An adjacency-labeling function is a function $\lambda : V \to \{0, 1\}^l$ for which the following conditions hold:

- The labeling λ is injective
- 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 G, usually with a fixed limit on the number of vertices n.

Definition 1.0.2. The triple $(\mathrm{G}, , \delta)$ is a **Adjacency Labeling Scheme** if G is a graph class and for every graph $G \in \mathcal{G}$ is λ an adjacency labeling function with decoding function δ.

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 $\mathcal{G}$ has been encoded.

1.1 Universal Graphs

Definition 1.1.1. Given a set of graphs $\mathcal{G}$, a graph U is called **Universal** if it contains every graph from $\mathcal{G}$ as a subgraph.

Definition 1.1.2. A graph U is **Induced-Universal** for a set $\mathcal{G}'$ if every graph in $\mathcal{G}'$ is an induced subgraph of U.

For every graph class with size at most n, there is a simple universal graph of size n - the complete graph K_n.

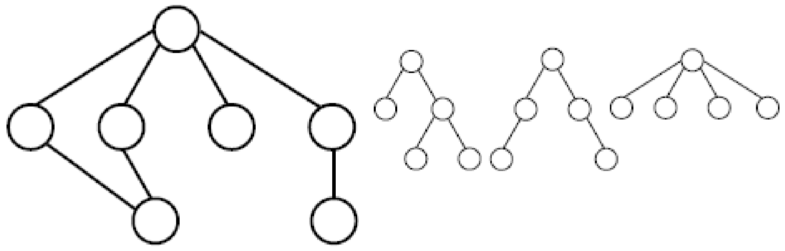
Figure 1.3: An induced-universal graph for trees of size 5 and the list of non-isomorphic trees of that size.

Lemma 1.1.1. Given an adjacency labeling scheme for a graph class 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 G, we can construct an adjacency labeling scheme with labels of lengths b for such class.

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 $G \in \mathsf{G}$, 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 $G \in \mathsf{G}$ (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 $\mathrm{O}(log\ n)$ 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

O(*log n*) must differ at least in one bit in their labels, and so there can be at-most $2^{O(log\ n)}$ such graphs. For example, bipartite graphs of size *n* cannot have any universal graph of size O(*n*) .

1.2 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**.

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.

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 [3] uses the following set of recursive prefix codes, which encode all non-negative integers *x*:

$$code_0(x) = 1 \circ 0^{x\ times}$$

$$code_i(x) = bin(x) \circ code_{i1}(|bin(x)| - 1) \qquad when i > 0.$$

In this thesis, we use only the first two codes in this recursion, $code_0$ and $code_1$, which we'll call c_l (the long code) and c (the short code) respectively.

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

2. …….

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 *logn* 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.

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 *logn*+O(*log logn*) bits[2]. 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 *logn* 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 [2], 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.

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

2.3 Traversal and Jumping

In 2007, Bonichon, Gavoille and Labourel published a technique named "Traversal and Jumping" which gives $\log n + O(1)$ label lengths for several classes of trees, most notably caterpillars and binary trees. These are some of the few non-trivial classes of trees for which $\log n + O(1)$ 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 $\log n+O(1)$ 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 [4]. 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 $log\ n + 2\ log\ d + O(1)$. We can translate it to a $log\ n + 3 \log d + O(1)$ 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.

Result 2. It is not possible to get an $log\ n + O(1)$ 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 $log\ n + \vartheta(log\ logn)$. [1]

2.5 Planar graphs

Planar Graphs are graphs which can be drawn without crossings on the plane.

Every planar graph can be decomposed into a union of three forests (disjoint sets of trees). Using this decomposition, we apply the straightforward labeling method and get the upper bound of $6\ log\ n + O(1)$. 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 $4\ log\ n + O(1)$.

We can employ another theorem, 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 $3\ log\ n + O(1)$.

In [5]Gavoille et al. suggested a different labeling, which produces labels of size $2 \log n + O(\log \log n)$. 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.

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* + O(1) adjacency labeling schemes for caterpillars and bounded degree trees.

This method was developed by Bonichon, Gavoille and Labourel 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.

3.2 Caterpillars

A **Caterpillar** is a simple tree consisting of a path (the spine) and vertices connected each to one element of the spine, called legs.

Theorem 3.2.1. There exists an adjacency labeling scheme for the set of all cater- pillars of size at most n, which uses $log\ n + 6$ bits for the label length. Also, we can construct such labeling in linear time and decode the labels in time 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 $_1$ 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 the number of leg vertices for a given s_i by d_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).

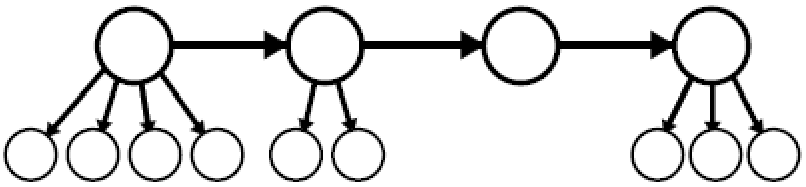

Figure 3.4: Caterpillar showing the suggested orientation.

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.

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.

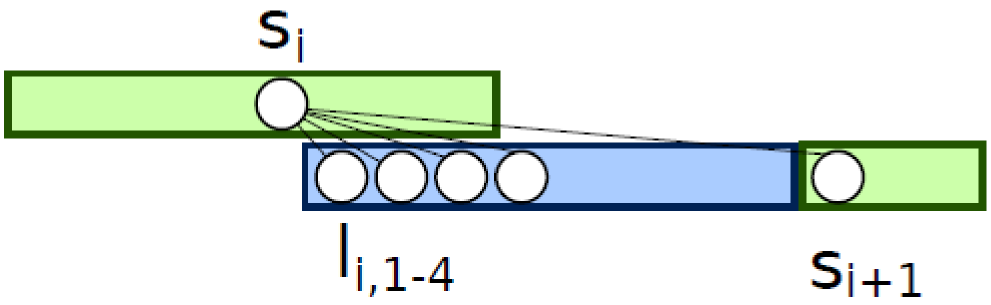

Figure 3.5: The location interval and the leg interval fors_i and a location interval for s_{i+1}. Note that the leg interval of s_i starts immediately after the set label for s_i and that it is of the same length as the location interval for s_i.

[1] .