University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Marcin Chrzanowski
Student no. 370754

MSO Query Answering on Trees

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr hab. Szymon Toruńczyk
Institute of Informatics

Warsaw, December 2022

Abstract

We define relabel regular questions on trees, which, via the known equivalence between tree automata and MSO formulae on trees, happens to be a generalization of the MSO query answering problem on trees. We show these questions can be answered in time $O_\varphi(m \log(m))$ with respect to question size (constant time in the case of MSO formulae with only first-order free variables) after preprocessing the input tree in time linear with respect to the tree's size. Along the way, we show an algorithm for answering questions of the form "Does the infix of a tree branch between nodes x and y belong to the regular language L" (for a previously fixed regular language L) in constant time after linear preprocessing. Our approach is much simpler in presentation than if using deterministic factorization forests due to Colcombet [7].

Keywords

MSO, query answering, tree automata

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatyka

Subject classification

- Theory of computation
  - Design and analysis of algorithms
    - Data structures design and analysis

Tytuł pracy w języku polskim

Odpowiadanie na zapytania MSO na drzewach

Contents

Chapter 1

Introduction

The model checking problem is a classic problem in theoretical computer science in which we are tasked with answering whether a given structure S satisfies a sentence φ of some logic. Several other interesting computational problems arise when we generalize to formulae with free variables.

As an analogy to databases, formulae with free variables are often referred to as *queries* in the literature. We consider the *query answering* problem: given a query $\varphi(X_1, ..., X_k)$ and a structure S, we want to answer questions of the form "does assigning the set of elements W_1 to X_1, ..., W_k to X_k, satisfy φ?". A naive solution would be to transform the formula φ into a sentence by adding k unary relations $U_1, \ldots, U_k$ and labeling elements of W_i with U_i. Now we can answer an individual question in the same time as model checking.

Already model checking is known to be computationally difficult for various interesting logics, but the situation can improve when restricting the class of structures considered. In this work we will focus on the case of MSO logic over trees. While MSO model checking is PSpace-hard in general, Courcelle [9] showed that deciding if an MSO sentence holds in a structure of bounded treewidth can be done in time linear in the size of the structure. This, using the naive method described above, gives us an algorithm for MSO query answering over trees that answers each question in time $O_\varphi(n)$[1] (n being the size of the tree).

In our setting we suppose that we want to answer multiple such questions about a single tree and formula, varying the valuations we ask about, and thus allow a preprocessing step over the formula and tree. In this case, the above naive algorithm could be improved with Amarilli et al.'s [1] data structure that allows updates of the underlying tree. Let m be the size of a given question, i.e. $m = |W_1 \cup \ldots \cup W_k|$. Amarilli's result allows us to preprocess the tree in linear time, label it with a choice of $W_1, \ldots, W_k$ in time $O_\varphi(m \log n)$, then answer model checking in constant time, thus greatly improving over the naive algorithm for questions that talk about a small number of vertices compared to the size of the whole tree.

1.1. Main result

We will improve on the above method even further, showing an algorithm with which, after linear preprocessing of the tree, we can answer questions in time linearithmic with respect to the question size. In particular, we eliminate any dependence on tree size during question answering.

[1]Throughout the text we use $O_x(f(n))$ notation, where x is some portion of an algorithm's input. It indicates x as a parameter of the algorithm, i.e. that for every integer k, there is some constant c_k such that if $|x| = k$, then the algorithm runs in time $c_k f(n)$ (n being the size of the remaining input).

More explicitly, we will show how to solve Problem 1.1, with a preprocessing algorithm working in time $O_\varphi(n)$ (where n is the size of the input tree), and then answer questions in time $O_\varphi(m \log m)$ (where m is the size of the question):

Problem 1.1 MSO QUERY ANSWERING ON TREES
Fixed: an MSO formula $\varphi(\vec{X})$ over trees with k free second-order variables.
Given: a tree T.
Questions: given a k-tuple of subsets of T's vertices $\vec{W} \in \mathcal{P}(V(T))^k$, is $\vec{W}$ a satisfying assignment to $\vec{X}$? In other words, does $T \models \varphi(\vec{W})$?

Note that though we speak of MSO formulae with second-order variables only, first-order variables are supported by simply restricting to questions in which sets assigned to first-order variables are singletons. Also note that for formulae with first-order free variables only question answering happens in constant time.

Through a series of reductions, we will show that MSO query answering reduces to the following problem about tree automata, which we solve in Chapter 4:

Problem 1.2 RELABEL REGULAR QUESTIONS ON TREES
Fixed: a deterministic bottom-up tree automaton A over ranked alphabet Σ.
Given: a tree T labeled with Σ.
Questions: given m relabelings $v_1 \mapsto a_1, \ldots, v_m \mapsto a_m$, where v_i are vertices of T and a_i are labels from Σ, what state does A arrive at in the root of T', where T' is T with each v_i's label modified to the corresponding a_i.

1.2. Related Work

Query answering is one generalization of model checking to the case of formulae with free variables. Another related one is *query enumeration*. Similar to query answering, we again allow the structure and query to be preprocessed. Then, instead of answering individual questions about valuations, we are interested in outputting all the satisfying valuations one by one.

The complexity class CONSTANT-DELAY$_{lin}$ refers to those enumeration problems in which the preprocessing step takes linear time and the delay between each successive answer is constant. LINEAR-DELAY$_{lin}$ is a generalization of CONSTANT-DELAY$_{lin}$ where the time complexity of the second algorithm is linear in the size of the answer being output. Bagan [2] introduced this second notion and showed that enumerating MSO queries is LINEAR-DELAY$_{lin}$ on trees. This in particular means that enumerating MSO queries for formulae whose free variables are all first-order is in CONSTANT-DELAY$_{lin}$, and Kazana and Segoufin [12] revisit this result, proving it using deterministic factorization forests due to Colcombet [7] rather than Bagan's intricate indexing structure.

Most closely related, Kazana [11] investigated MSO query answering (there called *query testing*) on structures of bounded treewidth in his PhD thesis. He showed that after linear preprocessing, questions can be answered in constant time, however this result was limited to MSO queries whose free variables are first-order variables (i.e., range over single vertices). This work generalizes to also allowing set variables in the query, and uses a different approach. Here again Kazana relied on Colcombet's factorization, an application of semigroup theory. Our appraoch is more algorithmic and straigforward.

1.3. Organization

The rest of our work is organized in the following way:

- In Chapter 2 we introduce definitions and known algorithms we will use to arrive at our main result.
    - In particular, Section 2.2 serves as a short review of some fundamental algorithms for solving problems related to trees and regular languages where a structure is preprocessed and then questions about it are answered.
- In Chapter 3 we solve an important subproblem, which is a generalization of a known algorithm about words to the tree case.
- In Chapter 4 we arrive at our main result by reducing Problem 1.1 to Problem 1.2 and solving it.
    - Of note, in Section 4.1.2 we show how to compute the LCA closure of a set of m vertices of a preprocessed tree in time $O(m \log m)$.
- In Chapter 5 we offer some concluding remarks, including discussion of time complexities with respect to parameters, which we mostly ignore throughout the main text, and a generalization of our result to structures of bounded treewidth.

Chapter 2

Preliminaries

2.1. Definitions

2.1.1. Trees

We work with finite rooted trees whose vertices are labeled with letters from a finite alphabet. More formally, given a finite alphabet Σ, for each $a \in \Sigma$, a is a tree, and if $T_1, \dots, T_k$ are trees, then $a(T_1, \dots, T_k)$ is also a tree. A tree T can then also be seen as an acyclic undirected graph with vertex set $V(T)$ and edge set $E(T)$, with a distinguished root vertex $r \in V(T)$.

We use the standard notions of leaf, child, sibling, ancestor, descendant, etc.

Binary trees are trees where each node has either no children (the node is a leaf), or exactly two children (which, based on their order, can be called the left and the right child).

Tree traversals

Take a binary tree T. The *post-order* of $V(T)$ is an ordering of T's vertices produced by the following recursive procedure:

1. Traverse the root's children's subtrees.
2. Visit the root.

Similarly, the *pre-order* of $V(T)$ is produced by the following recursive procedure:

1. Visit the root.
2. Traverse the root's children's subtrees.

Finally, the following procedure produces the *in-order* of $V(T)$:

1. Traverse the left child's subtree.
2. Visit the root.
3. Traverse the right child's subtree.

2.1.2. Tree automata

Consider the case of binary trees labeled with Σ. A *deterministic, bottom-up tree automaton* (further called just a *tree automaton*) consists of

- A finite set of *states* Q.
- A set of *accepting states* $F \subseteq Q$.
- A bottom-up *transition function* $\delta : Q \times \Sigma \times Q \to Q$.
- An *initialization function* $\iota : \Sigma \to Q$.

A *run* of tree automaton A over tree T is a relabeling of T with elements of Q such that

- Each leaf with label $a \in \Sigma$ is relabeled with $\iota(a)$.
- If an inner node v has label $a \in \Sigma$, its left child got relabeled to $p \in Q$, and its right child got relabeled to $q \in Q$, then v gets relabeled with $\delta(p, a, q)$.

A run is *accepting* if T's root gets relabeled to an accepting state, that is a state $q \in F$. The set of all binary trees accepted by an automaton A is called the *language recognized by* A, notated $L(A)$. We call the class of all languages recognized by tree automata *regular tree languages*, analogously to regular languages recognized by finite state automata.

We note that the expressive power of deterministic bottom-up tree automata is the same as that of nondeterministic (either bottom-up or top-down) tree automata.

2.1.3. Monadic Second Order (MSO) Logic

From a logic point of view, the trees we work with can be seen as structures over a signature with a single binary relation E and unary relations U_a for each $a \in \Sigma$. $E(v, w)$ represents a (directed from parent to child) edge from v to w. $U_a(v)$ signifies that v's label is a.

In the case of binary trees, the relation E is replaced with two binary relations E_l and E_r, which represent an edge from parent to left child, and from parent to right child, respectively.

For convenience, we will also make use of the binary relation $\leq$, with $v \leq w$ signifying that v is an ancestor of w (with every vertex being an ancestor of itself; $<$ can be used to signify a strict ancestor). Note that in the case of MSO on trees, $\leq$ can be defined using just the edge relation E.

We make use of a fundamental theorem tying MSO logic on trees and tree automata:

Theorem 2.1 *For every MSO formula φ over binary trees, there exists a tree automaton A such that for every binary tree T, $T \models \varphi$ if, and only if $T \in L(A)$.*

We note that the converse of this theorem is also true (i.e. that for every tree automaton, there is a corresponding MSO formula), however we will use only the MSO to automata direction in this work. See for example Bojańczyk's text [5] for a proof of both directions.

2.1.4. Computational Model

We use the Random Access Machine (RAM) model with uniform cost measure. In particular, basic arithmetic operations are assigned a constant cost, regardless of bitlength of arguments.

2.2. Question answering problems

Consider a computational problem whose inputs are of the form $(S, q) \in \mathcal{S} \times \mathcal{Q}$ for some set of *structures* $\mathcal{S}$ and some set of *questions* $\mathcal{Q}$, and answers come from a set $\mathcal{A}$. In other words, fix a function $f : \mathcal{S} \times \mathcal{Q} \to \mathcal{A}$. This induces a *question answering problem* which is divided into two phases:

preprocessing An input structure $S \in \mathcal{S}$ is given and a *preprocessing algorithm* outputs a data structure S' (which can be an arbitrary data structure, not necessarily an element of $\mathcal{S}$).

questions Given S' and $q \in \mathcal{Q}$, output $f(S, q)$.

We are interested in the time complexities of both phases. We use the following notation for algorithms that have both a preprocessing and question phase: if it takes $f(n)$ time to complete the preprocessing step for an input of size n, and $g(n, m)$ time to then handle a question of size m, we say the algorithm has time complexity $\langle f(n),\, g(n, m)\rangle$.

We turn to a discussion of several question answering problems with known solutions, both to serve as examples and because we will be using them in our algorithm.

2.2.1. Lowest Common Ancestor

Problem 2.2 Lowest Common Ancestor (LCA)
Given: a tree T.
Questions: given vertices x and y, find the vertex z that's an ancestor of both x and y, and is their lowest (i.e. furthest from the root) common ancestor.

Harel and Tarjan [10] were the first to show an optimal $\langle O(n),\, O(1)\rangle$ algorithm for LCA. Schieber and Vishkin [13] used a similar approach but simplified the indexing structure, keeping the same time complexities. Berkman and Vishkin [4] showed a completely new approach to the problem, which relies on answering range minimum queries (see below) about an array of the tree's vertices arranged in a specific order. Bender and Farach-Colton [3] offer a simpler presentation of the algorithm in [4] and note the equivalence between the LCA and RMQ problems.

2.2.2. Range Minimum Query (RMQ)

Problem 2.3 Range Minimum Queries
Given: an array A of integers.
Questions: given indices i and j, return the index of the smallest element in the subarray $A[i, j]$.

Bender and Farach-Colton [3] show an $\langle O(n),\, O(1)\rangle$ algorithm for the RMQ problem.

Their method is as follows. First they show that a special case of the RMQ problem, ± 1 RMQ, can be solved in $\langle O(n),\, O(1)\rangle$. This restriction of the problem is enough to handle LCA questions. Then, for the general RMQ case, a Cartesian tree[1] of the array is built, and LCA questions on this tree correspond to range minimum queries on the array.

[1]A Cartesian tree of an array is a binary tree with the array's minimum element in its root, the root's children being Cartesian trees of the left and right subarrays around the minimal element. It can be constructed in linear time.

2.2.3. Word infix regular questions

We now turn to a problem about regular languages. Note that the regular language is fixed, i.e. we treat the size of its representation (e.g. the size of an automaton representing it) as a parameter. Below we take a deterministic automaton for convenience, but if we instead start with a nondeterministic automaton we "don't care" about the exponential cost of determinizing it.

Problem 2.4 Word Infix Regular Questions
Fixed: regular language L over alphabet Σ, given by DFA A.
Given: a word $w \in \Sigma^*$.
Questions: given indices i and j with $1 \leq i < j \leq |w|$, does the infix $w[i, j]$ belong to L?

Let n be the length of w. This problem has a very elegant $\langle O(n), O(1)\rangle$ solution, as presented by Bojańczyk [6]. We present the full construction as we will be generalizing its internals for the tree case in Chapter 3.

preprocessing We begin by creating a graph whose vertices are $n + 1$ copies of the set of states of A, Q. More precisely, we will be working with a graph whose vertex set is $\{0, 1, \ldots, |w|\} \times Q$. We will refer to a vertex in the ith copy of Q labeled with state q as $i.q$.

Since A is deterministic, each letter $a \in \Sigma$ can be seen as a function $a : Q \to Q$, and we draw these functions as directed edges between successive copies of Q. For example, suppose the ith letter of w is a. If A in state q, reading a, would move to state q', then there will be an edge from $(i - 1).q$ to $i.q'$. We can think of this construction as first drawing copies of Q around each letter of w, then replacing each letter with the edges induced by A's transition function.

Note that by determinism, each vertex has exactly one outgoing edge (except for vertices in the last copy of Q which have no outgoing edges at all).

Now we will color the vertices of the graph we just constructed with the colors $1, 2, \ldots, |Q|$ in such a way that

1. every copy of Q has one vertex of each of the $|Q|$ colors;
2. when $i.q$ has color c and there is an edge to $(i + 1).q'$, which is colored with color c', then $c \geq c'$.

The second point basically means that we will be trying to draw single-color paths, but when paths need to join, it is the higher-colored path that gets cut off (think of the colors as priorities, with lower numbers representing more important priorities; see Figure 2.1).

The construction is as follows:

1. Color an arbitrary vertex of the first copy of Q with the color 1.
2. Follow the deterministic edges to the end of the word, coloring all vertices along this path with 1.
3. Now color another uncolored vertex of the first copy of Q with the color 2.
4. Try following edges as far as possible, coloring all visited vertices with 2.
5. If your run into an already colored vertex, pick an arbitrary uncolored vertex in this copy of Q to color with 2 and continue from here.

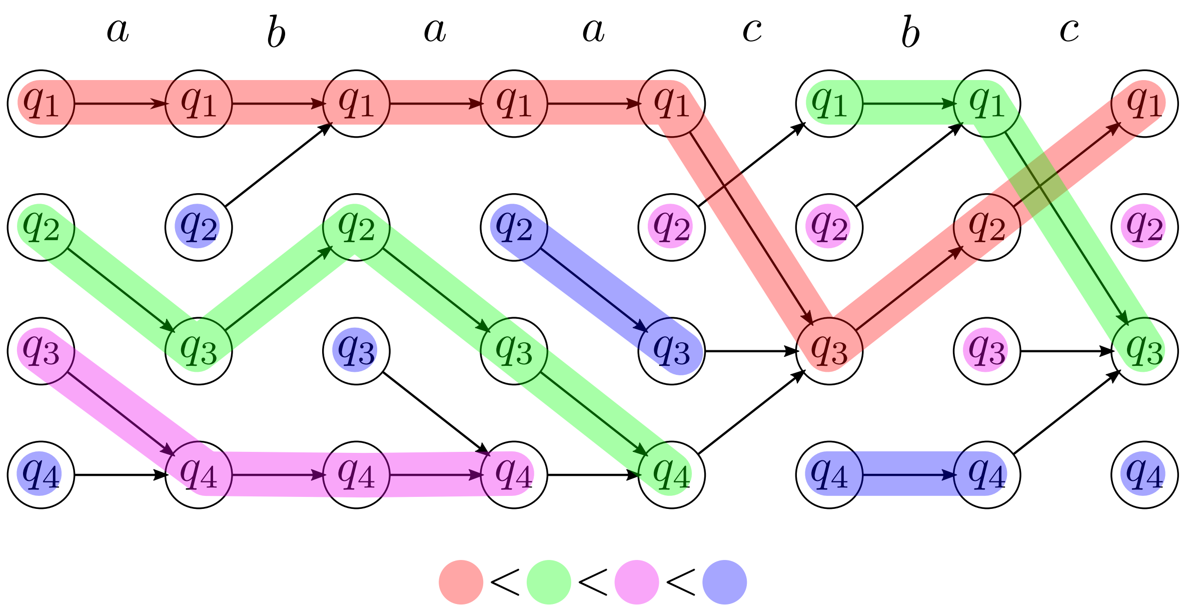


Figure 2.1: The data structure for infix regular questions over word *abaacbc* and a simple automaton with 4 states. As shown by the color blotches below the graph, red is color 1, with the highest priority, green is color 2 with the next priority, pink is color 3, blue is color 4, with the lowest priority. The information from break is not shown in the figure, but it amounts to a pointer from each vertex to the rightmost vertex on its connected single color component.

6. Repeat steps 3.-5. for each successive color up to $|Q|$.

Additionally, for each vertex $i.q$, we store the index of the next copy of Q in which the path of this vertex's color is broken by a lower color. Put this information in table break. For example, if $i.q$ is colored with color c, and when following the deterministic path from $i.q$, all encountered vertices are colored c until $j.q'$ which is colored c', then we set $\mathsf{break}[i.q] = j - 1$. See Figure 2.1 for a visual representation of the data structure described for an example word and automaton.

We can compute break in linear time with a single backwards pass through the colored graph.

answering questions Consider a question "does $w[i, j] \in L$". We claim that using our colored graph and the table break we can, in constant time, conclude in what state the automaton A will end up in on the jth position of w if it had started in its initial state on the ith position.

First, look at vertex $(i - 1).q_0$, which is the vertex of A's initial state in the $(i - 1)$th copy of Q and note its color c. Now we want to answer the following question: if we follow the edges of the graph until the jth copy of Q, what color will we end in? First, look at $k := \mathsf{break}[(i - 1).q_0]$. If $k \geq j$, then we know that in the jth copy of Q, the path we're interested in still has color c. Find the unique vertex $j.q$ in this copy of Q that's colored with c. q is the state we will end up in, so if it is an accepting state answer YES, if not, answer NO.

If $k < j$, then jump to the kth copy of Q and consider the vertex $k.q$, the unique vertex here colored c. From $k.q$, follow its single outgoing edge to $(k + 1).q'$, which will be colored with color $c' < c$. Continue as we did before, by looking at $k' := \mathsf{break}[(k + 1).q']$, comparing it to j, and either halting if $k' \geq j$, or jumping again otherwise. Because with each jump we move to a color strictly smaller than before, the number of jumps is bounded by the number of colors, $|Q|$. Thus the question is answered in time $O(|Q|)$, which is constant with respect to $|w|$.

Chapter 3

Branch Infix Regular Questions

Before solving our main problem, that of MSO query answering on trees, we generalize word infix regular questions (Section 2.2.3) to trees. This will be a vital step in the MSO query algorithm, but is an interesting result on its own.

The question answering problem we will solve is:

Problem 3.1 Branch Infix Regular Questions
Fixed: regular language L over alphabet Σ, given by DFA A.
Given: a Σ-labeled tree T.
Questions: given a vertex x and its descendant y, does the word given by labels on the path from x to y belong to L?

Let n be the size of $V(T)$.

Theorem 3.2 *Problem 3.1 can be solved in time $\langle O_A(n),\, O_A(1)\rangle$.*

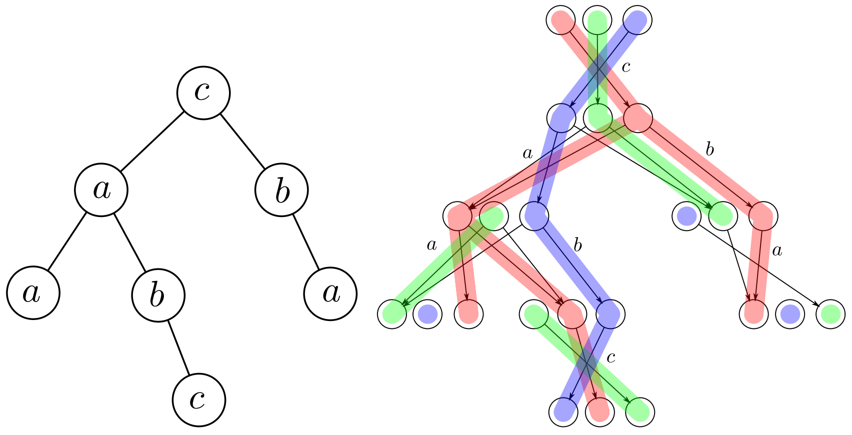
We begin with a similar construction as in the word case, i.e. we create a graph of $n+1$ copies of A's states Q. The vertex set of the graph will be $(V(T) \cup \{\mathsf{root}\}) \times Q\}$ and we can refer to a specific vertex as $x.q$ for $x \in V(T) \cup \{\mathsf{root}\}, q \in Q$.

Again, each letter $a \in \Sigma$ defines a function $a : Q \to Q$, and these functions induce edges in our "fattened" tree: consider a vertex x of T, labeled with letter a and let y be its parent (or the new vertex root when x is the root of T). If A in state q, reading letter a goes to q', then there will be an edge from $y.q$ to $x.q'$. In the word case we had placed copies of Q around each letter and replaced the letters with edges implied by A's transition function, here we perform an analogous transformation on a tree shaped input.

We also color all vertices with colors $1, \ldots, |Q|$ in this graph, analogously to how we did in the word case: begin by coloring an arbitrary vertex in the root with 1, then follow edges downwards, coloring all visited vertices with 1. Then begin the same process with the next color, restarting with a different vertex in a given copy of Q if we run into an already colored vertex.

This process will again lead to a coloring with the desired properties that

1. every copy of Q has one vertex of each of the $|Q|$ colors;
2. when a vertex of color c has an edge to a vertex of color c' in a child copy of Q, then $c \geq c'$.



(a) The original tree T (b) The colored graph

Figure 3.1: On the left, a sample labeled tree. On the right, the colored graph we create for the purposes of answering branch infix regular questions for this tree and a simple three state automaton. Note how the copies of the automaton's states are arranged in the same as shape as T, with an additional new root copy. The letters of the original tree fit in the spaces between copies of Q, inducing the edges between them.

See Figure 3.1 for how the colored graph is constructed for an example tree.

Now let's consider how we could answer the question "is the branch infix from x down to y in L?". Here we can't proceed exactly as in the word case. We don't have a break table since a vertex in an internal node can have arbitrarily many points below it where its color is broken by a lower one. Ideally we would like to be able to find such a color break point that is "in the direction of y from x".

In the next section we formulate this question formally and solve it as a subproblem.

3.1. Highest Marked Descendant on Path

Consider the following question answering problem:

Problem 3.3 Highest Marked Descendant on Path Questions
Given: a tree T with set $M \subseteq V(T)$ of marked vertices.
Questions: given a vertex x, its descendant y, find the node $z \in M$ that is the highest marked node on the path between x and y, if such z exists.

Theorem 3.4 *Problem 3.3 can be solved in time $\langle O(n), O(1)\rangle$.*

We will build an index structure, constructible in linear time, that allows us to handle such questions in constant time. The structure is heavily inspired by Bender and Farach-Colton [3], where a simple algorithm for answering LCA questions is presented.

First, we create the array post of length n, which is the post-order of the nodes.

Next, we label each node of the tree with its pre-order number. We create the array pre with the corresponding pre-order labels of the nodes in post, i.e. if $\mathsf{post}[i] = v$, then $\mathsf{pre}[i]$ is v's pre-order number.

Finally, for each node of the tree, we record its index in post in the array index.

Observation 3.5 *Given a node x and its descendant y, looking at the range* $\mathsf{pre}[\mathsf{index}[y], \mathsf{index}[x]-1]$*:*

1. *All the values in this range correspond to descendants of x.*

2. *The values smaller than* $\mathsf{pre}[\mathsf{index}[y]]$ *correspond to ancestors of y.*

3. *In particular, the minimum of this range corresponds to the highest ancestor of y that is also a descendant of x.*

Proof: The pre-order labels in pre are ordered according to a post-order. In a post-order, the root of a subtree is visited directly after all its descendants. Since y is a descendant of x, all values between its position in pre and x's position there will also be descendants of x, giving item 1.

The values in pre are pre-order numbers. In a pre-order, for a given node z, the only nodes that will have lower pre-order numbers are z's ancestors, and the children of these ancestors that are to the left of the child towards z. By starting our range at index $\mathsf{index}[y]$, we've skipped over all of the second types of vertices with lower pre-order numbers, leaving only y's ancestors as nodes in the range with a lower pre-order number. This gives us item 2.

Items 1 and 2 together give us item 3.

In our problem, we only care about ancestors of y that are marked. So we perform one final modification of our data structure: for all non-marked vertices v, we change $\mathsf{pre}[\mathsf{index}[v]]$ to ∞ (which can be represented by $n+1$, an integer greater than any node's pre-order label). With pre modified like this, we observe that now the minimum value of $\mathsf{pre}[\mathsf{index}[y], \mathsf{index}[x]-1]$ corresponds exactly to the answer of our question – the highest marked node between x and y.

We preprocess pre for RMQ in linear time.

Now when given a question "what is the highest marked descendant of x in the direction of y", we:

1. Look up $i := \mathsf{index}[y]$ and $j := \mathsf{index}[x] - 1$.

2. Perform an RMQ lookup on $\mathsf{pre}[i, j]$, giving us index k of the minimal value in that range.

3. Look up the corresponding vertex as $z := \mathsf{post}[k]$. This is the answer to our question.

We can detect the siutation of there not being a marked node between x and y during step 2 by checking if $\mathsf{pre}[k]$ is smaller than the pre-order label of y. If it isn't, then all the nodes on the path from x to y were unmarked (had higher values in the modified pre since values at indices corresponding to unmarked nodes are set to ∞), and k will correspond to a marked node that's not on the path from x to y (or some unmarked vertex if there were no marked vertices at all in the range we check).

See Figure 3.2 for a visual example of the data structure described.

Constructing each of post, pre, and index takes linear time, RMQ is a $\langle O(n),\ O(1)\rangle$ problem, so we have proved Therom 3.4.

3.2. Answering branch infix regular questions

With the above problem solved, we are ready to finish our algorithm for branch infix regular questions. Recall that we have created a graph of copies of Q shaped similar to T where edges between vertices correspond to transitions of A along corresponding vertices of T. During preprocessing we additionally need to remember in each vertex $x.q$ a unique identifier of the

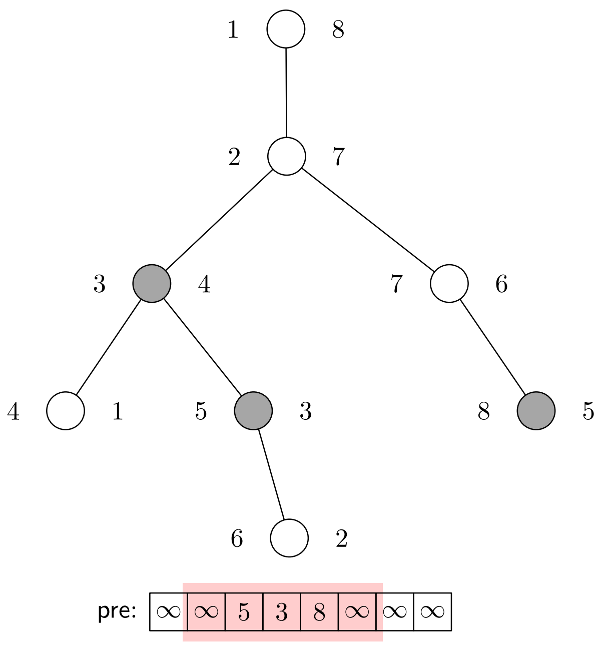


Figure 3.2: A preprocessed marked tree. The marked vertices are colored dark grey. To the left of each vertex is its pre-order number. To the right is its post-order number. On the bottom is the table pre with only the marked vertices' prefix numbers not replaced by ∞. The red rectangle marks the range for which we issue an RMQ question when answering a highest marked descendant question form the vertex with post-order index 7 down to the one with post-order index 2.

connected component of vertices of the same color that it is in (the identifiers can be the root vertices of each such component), call this data component$[x.q]$.

For each color c, we will preprocess the above graph for answering questions of the form "given a vertex $z.q$ colored with c, which is the first copy of Q on the path towards the copy of Q that contains y where the path from $z.q$ changes color?". To achieve this, we create a copy of T with a newly added root (thus this tree's vertices correspond to copies of Q in our fattened tree) in which we mark the vertices corresponding to copies of Q where a non-c-colored vertex is pointed to by an edge from a c-colored vertex in the parent copy of Q. This tree we preprocess for highest marked descendant on path questions, solved in the previous section. Note that Figure 3.2 is exactly the marked tree we create and preprocess for the green color of the colored graph in Figure 3.1.

Now question answering can proceed similar to the word case. Given a question "does the word on vertices from x down to y belong to L?", let x' be x's parent in T (or root if x was T's root), look at the vertex $x'.q_0$ and consider its color c. In the copy of Q corresponding to y consider the vertex $y.q$, the unique vertex here of color c. If it is in the same connected component as $x'.q_0$ (which we check by comparing component$[x'.q_0]$ with component$[y.q]$), we can immediately answer whether or not A will accept the word on this path based on whether this state is accepting. Indeed, if the two vertices are in the same component then there is a single-color path from $x'.q_0$ down to $y.q$ and q is indeed the state A would have ended in after running on the word given by the labels from x down to y.

Otherwise, we need to jump down to a lower copy of Q. We can find the first such copy where the color on our path changes from c to something lower by asking a highest marked descendant on path question on the marked tree we created for color c. The answer to this question exactly corresponds to the point where the path of color c from our initial state merges into a lower color c' on the path towards y. From here we continue as at the beginning, checking component to see if we can answer immediately, or taking another jump down. Handling each jump takes constant time, and the number of jumps is bounded by $|Q|$ since we can only jump to a lower color. Thus we can answer branch infix regular questions in time constant with respect to $|T|$.

This finishes the proof of Theorem 3.2.

Chapter 4

MSO Query Answering on Trees

Now we have all the pieces necessary to present our main result, which we can formulate as Theorem 4.1:

Theorem 4.1 *Problem 1.1 can be solved in time* $\langle O_\varphi(n),\, O_\varphi(m \log m)\rangle$.

First, let's reduce Problem 1.1 to Problem 1.2:

Lemma 4.2 *If Problem 1.2 has an* $\langle O_A(n),\, O_A(m \log m)\rangle$ *solution, then Problem 1.1 has an* $\langle O_\varphi(n),\, O_\varphi(m \log m)\rangle$ *one.*

Proof: We proceed with several standard linear reductions that transform Problem 1.1, which speaks about MSO formulae with free variables, to Problem 1.2, which talks about tree automata over binary trees.

First of all, if T is not binary, we can use a standard encoding to encode it inside of a binary tree T', modifying φ to φ', such that $T \models \varphi(\vec{W})$ if and only if $T' \models \varphi'(\vec{W})$.

Now, to use Theorem 2.1, we turn the formula into an MSO sentence without free variables by adding a unary relation U_X for each variable $X \in \vec{X}$ and instead of considering the formula φ', we now consider the sentence

$$\varphi'' = \exists_{X_1} \dots \exists_{X_k} (\forall_x x \in X_i \iff U_{X_i}(x)) \wedge \varphi'(X_1, \dots, X_k).$$

Selecting a model in which the vertices in set W are exactly those colored with unary relation U_X is equivalent to a valuation of $\vec{X}$ in which the variable X is set to W.

Now by Theorem 2.1, there is a tree automaton A for binary trees over the alphabet $\Sigma \times \{0,1\}^{\vec{X}}$ which accepts a tree if and only if its labeling corresponds to a model satisfying φ''. Here a label $b_1 \dots b_k$ (where $k = |\vec{X}|$ and each $b_i \in \{0,1\}$) should be interpreted as a bit vector signifying which unary relations U_X a vertex is assigned to.

By combining the above reductions, we can solve MSO query answering on trees by solving the relabel regular qeustions problem on trees. Indeed, fix a formula $\varphi(\vec{X})$ and take a tree T. Derive automaton A and tree T' as above. Label each vertex of T' with $0 \dots 0$ (signifying that none of its vertices have yet been assigned to any of the variables in $\vec{X}$). Preprocess A and T' for relabel regular questions. Now a relabeling of T''s vertices corresponds to selecting a specific valuation $\vec{W}$ of $\vec{X}$, thus answering whether or not A accepts the relabeled tree is equivalent to answering whether or not $T \models \varphi(\vec{W})$.

This concludes the proof of the lemma, and Theorem 4.1 will be proved if we prove:

Theorem 4.3 *Problem 1.2 can be solved in time $\langle O_A(n),\ O_A(m \log m)\rangle$. Furthermore, time complexity with respect to the size of the tree automaton is exponential in both the preprocessing and question answering phases.*

The rest of this chapter is dedicated to proving Theorem 4.3.

4.1. Relabel Regular Questions on Trees

Fix a tree automaton A and take a Σ-labeled binary tree T.

We'll preprocess the tree in such a way that when given a relabel question, we'll be able to partition the tree into linearly (with respect to the question's size) many parts, then, in a bottom-up fashion, compute the state in the root of each part in the relabeled tree. The computation for each part will take constant time.

Partitioning into the $O(m)$ parts we're interested in will take time $O(m \log m)$, then after the bottom-up computation we will arrive at the root's state in time $O_A(m)$, thus answering the question in the promised time.

4.1.1. LCA partition

Consider a set of tree vertices $W \subseteq V(T)$. We'll say that W is *LCA closed* if for every pair of vertices $v, w \in W$, it is the case that $\mathit{LCA}(v, w) \in W$ (note that one of v or w might be their LCA).

$W \subseteq V(T)$ is *partition ready* if

1. it is LCA closed;
2. it contains the root of the tree;
3. for every $v \in W$, either it has descendants in W in at most one of its subtrees, or both of its children are elements of W.

We will define the *LCA partition with respect to* W for a partition ready subset W of T's vertices. The partition will assign each vertex of T to a unique part, and each part will be rooted in an element of W (in particular, there will be as many parts as elements in W).

Each part will be of one of three types:

subtree A *subtree of* v is v along with all its descendants; it is rooted in v.

subtree with hole A *subtree of* v *with hole* w is the subtree of v minus the subtree of w (for w a descendant of v); it is rooted in v.

singleton A singleton vertex v, which is the root of such a part.

Now, for a partition ready set $W \subseteq V(T)$, the LCA partition of T with respect to W is a partition into subtrees, subtrees with holes, and singletons constructed according to the following rules:

1. If $v \in W$ is maximal in W with respect to the ancestor relation $<$ (i.e. there are no other elements of W in the subtree of v), then the subtree of v is a part of the partition.

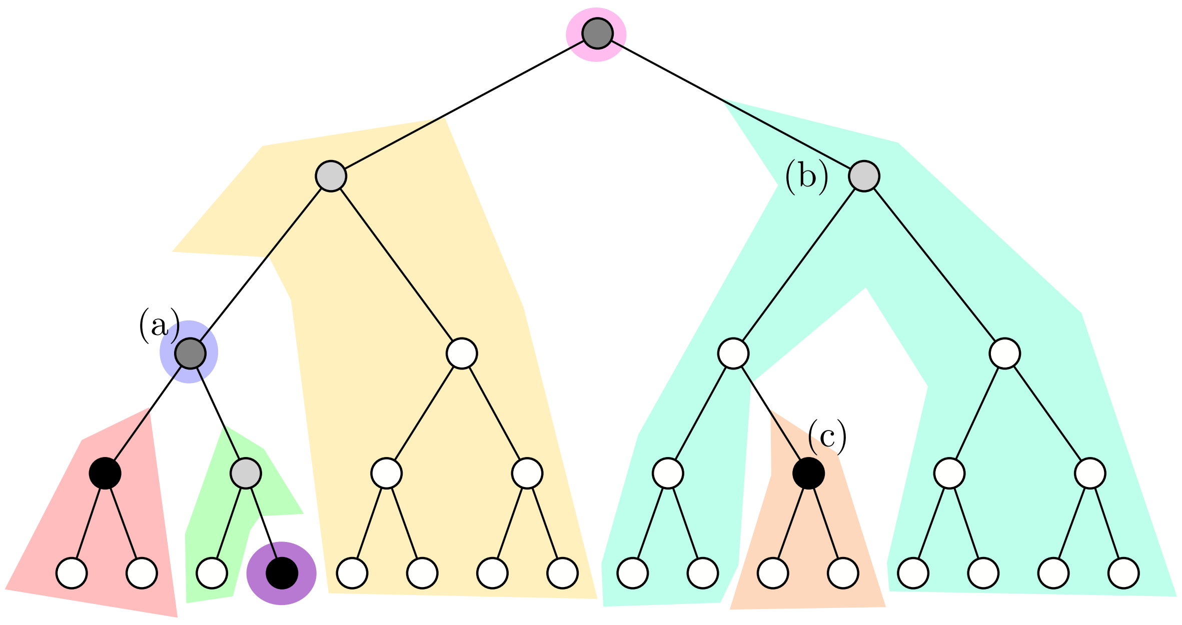


Figure 4.1: An LCA-partition of the tree with respect to the black vertices. The dark grey vertices are added when computing the LCA closure, the light grey vertices are the final vertices needed to create a minimal partition ready set containing the black vertices. Part (a) is an example of a singleton, part (b) of a subtree with a hole, and (c) of a subtree.

2. If $v \in W$ has descendants that are in W only in its left child's subtree, let w be the highest such descendant (there is a unique highest descendant because W is LCA closed). The subtree of v with hole w is a part of the partition.

3. Symmetrically if $v \in W$ only has descendants that are in W in its right child's subtree.

4. If $v \in W$ has descendants that are in W in both its children's subtrees we know that v's children are also elements of W by property 3 of being partition ready. Put v into a singleton part on its own, all the other elements of its subtree were assigned to parts when its children were handled.

If W is not parition ready, we will define the LCA partition with respect to W the partition with respect to the minimal partition ready set that contains W. See Figure 4.1 for an example of a partitioned tree. Skipping ahead, the LCA partition of the set of vertices relabeled in a relabeling question will be the partition we alluded to at the end of the previous section. We promised that this partition will have $O(m)$ parts, so we must show that for a set W, the minimal partition ready set containing it is of size $O(|W|)$.

Take a set of m tree vertices W and consider the following procedure:

1. Let W' be W with the tree's root added.

2. Let W'' be the LCA closure of W'.

3. Let W''' be W'' plus, for every element of W'' that has elements of W'' in both its children's subtrees, the element's both children.

Claim 4.4 *The above procedure produces the minimal parition ready set containing W and each step adds at most $O(m)$ new vertices.*

By defintition of LCA closure, W'' above is the smallest LCA closed set containing W and the tree's root. Step 3 adds at most $2|W''|$ new vertices, we will now prove that these are exactly the only vertices still needed to be added to arrive at a partition ready set. For now assume that only $O(m)$ vertices were added in step 2, this will be proved in the next section.

It's easy to see that W''' is still LCA closed. It is obvious that it is necessary to add at least all the vertices added in step 3, otherwise property 3 of partition readiness won't be satisfied. What we will show is that no other new vertices need to be added. Take $v \in W''$ that has descendants from W'' in both its subtrees, let v_l and v_r be its left and right child, respectively. Without loss of generality, suppose v_l is not an element of W''. The only reason we would need to add even more vertices after adding v_l would be if v_l had descendants in both its children's subtrees in W'''. First let's show that it has descendants in W'' in at most one of its subtrees. Indeed, if it did have descendants from W'' in both its subtrees, W'' wouldn't have been LCA closed – v_l would have been the LCA of the topmost members of W'' of its children's subtrees. Newly added elements of W''' can appear only underneath vertices that were originally in W'', so if v_l didn't have an element of W'' in one of its subtrees, then it also doesn't have elements of W''' in that subtree. Thus W''' is indeed the minimal partition ready set containing W.

4.1.2. Computing the LCA closure

Lemma 4.5 *After linear preprocessing of a tree T, the LCA closure of a set of m tree vertices $W \subseteq V(T)$ can be computed in time $O(m \log m)$, and the size of the closure is linear in the size of W.*

Proof: In the preprocessing step, we will preprocess the tree for LCA questions, and assign each vertex v its in-order number, $\mathsf{in}[v]$.

Now to compute the closure of W, we first sort the vertices in W with respect to their in-order numbers, so we end up with a list $v_1, \ldots, v_m$ of vertices such that $\mathsf{in}[v_1] < \ldots < \mathsf{in}[v_m]$.

Observation 4.6 *If $\mathsf{in}[u] < \mathsf{in}[v] < \mathsf{in}[w]$, then $LCA(u, w)$ is equal to either $LCA(u, v)$ or $LCA(v, w)$.*

This may be proved by a simple case analysis.

Observation 4.7 *Consider two tree vertices, $u, w \in V(T)$ with $\mathsf{in}[u] < \mathsf{in}[w]$, and their least common ancestor v. Then $\mathsf{in}[u] \leq \mathsf{in}[v] \leq \mathsf{in}[w]$.*

Proof: If v is equal to one of u or w, the observation is obvious. For example, if $v = u$, we have that $\mathsf{in}[u] = \mathsf{in}[v] < \mathsf{in}[w]$. If v is a third vertex, then u must be in its left subtree and w must be in its right subtree. Thus an infix walk over T will first visit u, then v, and then w.

Now, once W has been sorted by in-order numbers (which takes time $O(m \log m)$ using a standard sorting algorithm), we can complete our computation in time $O(m)$. Consider the following set of vertices:

$$U := \{v \mid v = LCA(v_i, v_{i+1}) \text{ for } 1 \leq i \leq m-1\}$$

U is of size $O(m)$ and can be computed in time $O(m)$ by issuing LCA questions to successive pairs of the sorted W. We claim that $W' := W \cup U$ is the LCA closure of W.

Let's define $u_i := LCA(v_i, v_{i+1})$. By Observation 4.7, each u_i falls between v_i and v_{i+1} in the infix order, i.e. W' sorted by infix numbers is the sequence $v_1, u_1, v_2, u_2, \ldots, u_{m-1}, v_m$

(note that in the way this sequence is written, some vertices might be repeated, e.g. it might be the case that $LCA(v_2, v_3) = v_2$, in which case u_2 will be equal to v_2). We claim that for every pair of vertices in W', their LCA is also in W'. We'll proceed by induction, and it will be easier if we now rename the sorted W' to $w_1 := v_1, w_2 := u_2, \ldots, w_{2m-2} := u_{m-1}, w_{2m-1} := v_m$. As quick recap, the following are properties of the sequence of w_i's:

1. For $i < j$, $\mathsf{in}[w_i] \leq \mathsf{in}[w_j]$.
2. For odd i, w_i comes from the original set W.
3. For even i, w_i comes from the set of added vertices U. In particular, w_i is equal to $LCA(w_{i-1}, w_{i+1})$.

We prove that W' is LCA closed by showing that for all pairs w_i, w_j, their LCA belongs to W', by induction on the difference between i and j.

The base case is to show that the LCA of w_i and w_{i+1} is in W' for every pair of successive vertices. Suppose i is odd, then w_{i+1} comes from U and was constructed as the LCA of w_i and w_{i+2}. Then w_{i+1} must be on the path from w_i to the root and is the LCA of the pair. Similarly for when i is even.

Now assume that for all w_i, w_j, if $j - i < k$ then $LCA(w_i, w_j)$ is in W'. We want to show the same is true when $j - i \leq k$. Take w_i and w_j with $j - i = k$. Consider the triple w_i, w_{i+1}, w_j. By Observation 4.6, $LCA(w_i, w_j)$ is equal to $LCA(w_i, w_{i+1})$ or $LCA(w_{i+1}, w_j)$. By the induction assumption, we already know both of these are in W', thus showing W' is LCA closed and finishing the proof of Lemma 4.5

4.2. Computing root states of parts

Let's now describe how we will use LCA paritions to solve the relabel questions problem. Our approach is the following: given the question "is T accepted by A after relabeling according to $v_1 \mapsto a_1, \ldots, v_m \mapsto a_m$?", let $W = \{v_1, \ldots, v_m\}$. Consider the partition of T with respect to W. We will show how to compute A's state in the root of each part of the partition, after the tree had been relabeled. Per our definition of the partition, we have three types of parts to consider:

1. A subtree rooted at vertex v.
2. A singleton vertex v.
3. A subtree rooted at vertex v with hole w.

The first case is trivial to compute: no descendants of v had been relabeled, so the states of the children of v are the same as in A's run on the original tree T. We look them up in the precomputed run, then apply A's transition to those states and v's new label.

We will compute the states of part roots in a bottom-up fashion, so the second case is also simple. Both of v's children are roots of parts (since the LCA partition is based on a partition ready set), so once we've computed the states in the children of v, we again simply apply A's transition function to those states and v's new label.

The third is the interesting case, that of a subtree with a hole. Consider the path from the hole w up to the root v. What we claim is that the question "if A's state in w is q, is the state in v going to be q'?" can be answered with a branch infix regular question.

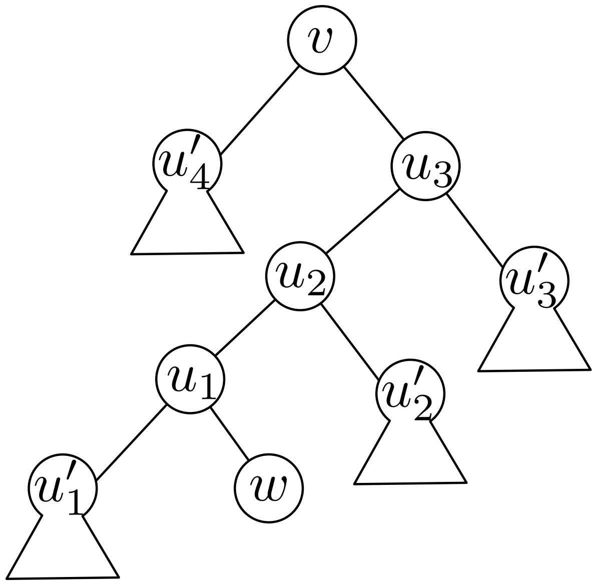


Figure 4.2: The subtree of v with hole w. All its nodes can be partitioned into the path $u_1, \dots, u_4 = v$ and the subtrees of $u_1', \dots, u_4'$.

4.2.1. Computing the root state of a subtree with a hole

Consider the subtree of v with hole w. It can be divided into the following components:

- A path from the parent of w up to v, notate it as $u_1, u_2, \dots, u_p = v$.
- For each i from 1 to p, the subtree of u_i's child that doesn't lie on the path from w to v. Let u_i' be the child of u_i that doesn't lie on the w to v path.

The node w is not a part of the subtree with hole, but we will need to consider it during the following computation, so for ease of notation we will define $u_0 := w$, extending the path of u_i's.

Figure 4.2 depicts a subtree with hole labeled according to this nomenclature.

With our bottom-up computation, we've already computed the new state in w. There have been no relabelings in the subtrees of the u_i''s, so we can get their states from the precomputed run of A on the original tree. Now consider how, using all this information, we could easily compute v's new state.

To compute the state in u_1, all we need is u_1's label and the states of u_0 and u_1' – u_1's children. This is all information we already have. So going up the path from w to y, we can compute each u_i's new state from its label and the states of u_{i-1} and u_i'. The computation as just described would take time linear in p, the length of the path, but the computation is simple enough that it could be performed by a finite state word automaton, allowing us to use our result from Chapter 3. To show this formally, consider the alphabet $\Sigma' := Q \times \Sigma \times Q \times \{\mathsf{left}, \mathsf{right}\}$. A word over this alphabet can be used to represent a subtree with a hole. A subtree with hole as described above would be represented by word $\alpha_0\alpha_1 \dots \alpha_p$ with each α_i being a quadruple $\langle p_i, a_i, q_i, d_i \rangle \in \Sigma'$ (with $p_i, q_i \in Q$, $a_i \in \Sigma$, d_i either left or right) where:

- a_i is the original label of u_i.
- p_i and q_i are the states of u_i's children in the precomputed run of A over T.

- d_i encodes the information of whether u_i is the left or right child of its parent.

Note that this is slightly more information than we previously mentioned: in our description of the computation above we only cared about the state of the child not on the w to v path. The information about the other child is redundant when considering just one subtree with hole and can be ignored when doing computation on that particular subtree with hole. But we will keep information about both children so that we can handle questions about any subtree with hole.

Now consider the language $L_{q,q'} \subseteq \Sigma'^*$ of words $\alpha_0 \dots \alpha_p$ such that the subtree with hole encoded by such a word, if the hole's state were set to q, would lead to A arriving at state q' after running up it.

Lemma 4.8 *$L_{q,q'}$ is a regular language.*

Let's explicitly construct the automaton recognizing $L_{q,q'}$. Its state space will be $\{\mathsf{initial}\} \cup Q \times \{\mathsf{left}, \mathsf{right}\}$, states other than initial encoding, for the vertex represented by the previously read letter, what state we arrived in at it and whether it was the current vertex's left or right child. The automaton works as follows:

1. In the initial state initial, read $\alpha_0 = \langle p_0, a_0, q_0, d_0 \rangle$, set the state to $\langle q, d_0 \rangle$.

2. Now when reading letter $\alpha_i = \langle p_i, a_i, q_i, d_i \rangle$ in state $\langle r, d \rangle$, if d if left, use A's transition function δ to compute the new tree state as $\delta(r, a_i, q_i)$ (the previously computed tree state is of our left child's, and our right child's comes from the current letter). If d is right, the new tree state will instead be $\delta(p_i, a_i, r)$. Store this and d_i as the new state.

3. Accept if the final tree state is q'.

This automaton recognizes exactly $L_{q,q'}$, by performing essentially the same computation we described before. This completes the proof of the lemma.

Our algorithm for branch infix regular questions dealt with top-down questions (i.e. the word we ask about runs from a higher vertex down to its descendant). Regular languages are reversible, so we can preprocess T for branch infix regular questions for each language $L^R_{q,q'}$ (where $\cdot^R$ denotes the reversed language). A question about whether the word on the path from v down to w belongs to $L^R_{q,q'}$ is the same as asking if the word from w up to v belongs to $L_{q,q'}$. Note that the number of languages we need to preprocess for is constant in the size of the tree – it only depends on the size of the automaton.

Given the subtree of v with hole w, once we've computed w's new state q, to compute v's new state, take v' – v's child in the direction of w, find $q' \in Q$ such that the path from v' to w belongs to $L^R_{q,q'}$ (using branch infix regular questions; since A is deterministic, there will be exactly one such q'). Now from q', the state of v's other child, and v's new label, use A's transition function to compute v's new state.

Since T's root is the root of one of the parts of our LCA partition, in the end we will know whether or not A accepts T after relabeling. Handling each part takes $O_A(1)$ time, and there are $O(m)$ parts to handle. We did require $O(m \log m)$ time to sort the vertices mentioned in the question when computing their LCA closure, thus question answering takes time $O_A(m \log m)$. This concludes the proof of Theorem 4.3.

4.3. The full algorithm

As a recap we summarize all the steps of the full MSO query answering algorithm.

preprocessing Take an MSO formula $\varphi(X_1, \ldots, X_k)$ and a Σ labeled tree T.

1. Transform φ and T into a deterministic bottom-up automaton A and binary tree T'.
2. Preprocess T' for computing LCA closures:
   (a) Preprocess T' for LCA questions;
   (b) Store each vertex's in-order number.
3. For each pair of A's states, q and q', preprocess T' for branch infix regular questions about the language $L^R_{q,q'}$.

answering questions We receive the question "for a tuple of sets of T's vertices, $\vec{W}$, does $T \models \varphi(\vec{W})$?"

1. From the selection of $\vec{W}$, generate a relabeling of T''s vertices that labels vertices mentioned in $\vec{W}$ as being assigned to the appropriate variables in $\vec{X}$. Let W be this set of relabeled vertices.
2. Compute W', the smallest partition ready set containing W.
3. Compute the LCA parition of T' with respect to W'.
4. In a bottom-up fashion, compute the state of A running over the relabeled tree in the roots of each part of the LCA partition.
5. Return YES if the state of T's root is accepting, NO otherwise.

Chapter 5

Conclusion

5.1. Complexities with respect to formulae and automata

In our algorithms in Section 2.2.3 and Chapters 3 and 4 we treated the MSO formula or automaton (over words or trees) that needed to be handled as a fixed parameter to our algorithms, and didn't analyze how they impact time complexities. Let's quickly perform this analysis now.

The MSO to automaton formula reduction implied by Theorem 2.1 is unfortunately non-elementary, making MSO query answering non-elementary with respect to $|\varphi|$. The situation improves if we start with a relabel regular questions problem directly, rather than needing to reduce from an MSO formula. The branch infix regular questions problem itself, which we use as a subalgorithm when answering relabel questions, behaves very well when given a deterministic automaton on input. To answer branch infix regular questions about a tree T and deterministic automaton A we need to create $|Q|$ copies of T for highest marked descendant questions, one for each color (and there are $|Q|$ colors). When answering a question, we will make at most $|Q|$ jumps, each jump itself taking constant time. Thus, using our notation for algorithms with a preprocessing phase, branch infix regular questions have an $\langle O(|A||T|), O(|A|)\rangle$ algorithm. Relabel regular queries themselves will unfortunately work in exponential time: recall that the languages we preprocess for branch infix questions for are constructed by *reversing* a deterministic automaton's language. This is a procedure that requires determinization of a nondeterministic automaton, and thus can explode exponentially.

5.2. Structures of bounded treewidth

With our algorithm we get, "for free", an $\langle O_\varphi(n), O_\varphi(m \log m)\rangle$ algorithm for MSO query answering on structures of bounded treewidth. Indeed, bounded treewidth structures are MSO interpretable in trees in linear time [8].

5.3. LCA closure question answering

In Section 4.1.2 we showed that the following question answering problem

Problem 5.1 LCA CLOSURE QUESTION ANSWERING
Given: a tree T.
Questions: given a set of tree vertices S, compute the LCA closure of S.

has an $\langle O(|T|),\, |S| \log |S| \rangle$ solution. We do not know whether this is an optimal solution. If the question answering step's complexity could be reduced to $O(|S|)$, MSO query answering on structures of bounded treewidth would then be known to be solvable in $\langle O_\varphi(n),\, O_\varphi(m)\rangle$. Or is there a lower bound on computing LCA closures in this setting, for example, would it be possible to reduce comparison sorting to computing LCA closures?

Bibliography

[1] Antoine Amarilli et al. “Enumeration on Trees with Tractable Combined Complexity and Efficient Updates”. In: *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems - PODS ’19*. ACM Press, 2019. DOI: 10.1145/3294052.3319702. URL: https://doi.org/10.1145%2F3294052.3319702.

[2] Guillaume Bagan. “MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay”. In: Springer Berlin Heidelberg, 2006, pp. 167–181. DOI: 10.1007/11874683_11.

[3] Michael A. Bender and Martín Farach-Colton. “The LCA Problem Revisited”. In: Springer Berlin Heidelberg, 2000, pp. 88–94. DOI: 10.1007/10719839_9.

[4] Omer Berkman and Uzi Vishkin. “Recursive Star-Tree Parallel Data Structure”. In: 22.2 (Apr. 1993), pp. 221–242. DOI: 10.1137/0222017.

[5] Mikołaj Bojańczyk. *An Automata Toolbox*. 2018. URL: https://www.mimuw.edu.pl/~bojan/papers/toolbox-reduced-feb6.pdf.

[6] Mikołaj Bojańczyk. *Factorization Forests*. 2009. URL: https://www.mimuw.edu.pl/~bojan/upload/confdltBojanczyk09.pdf.

[7] Thomas Colcombet. “A Combinatorial Theorem for Trees”. In: Springer Berlin Heidelberg, pp. 901–912. DOI: 10.1007/978-3-540-73420-8_77.

[8] B. Courcelle and M. Mosbah. “Monadic second-order evaluations on tree-decomposable graphs”. In: Springer Berlin Heidelberg, 1992, pp. 13–24. DOI: 10.1007/3-540-55121-2_2. URL: https://doi.org/10.1007%2F3-540-55121-2_2.

[9] Bruno Courcelle. “Graph Rewriting: An Algebraic and Logic Approach”. In: Elsevier, 1990, pp. 193–242. DOI: 10.1016/b978-0-444-88074-1.50010-x. URL: https://doi.org/10.1016%2Fb978-0-444-88074-1.50010-x.

[10] Dov Harel and Robert Endre Tarjan. “Fast Algorithms for Finding Nearest Common Ancestors”. In: 13.2 (May 1984), pp. 338–355. DOI: 10.1137/0213024.

[11] Wojciech Kazana. “Query Evaluation with Constant Delay”. PhD thesis. ENS Cachan, 2013.

[12] Wojciech Kazana and Luc Segoufin. “Enumeration of monadic second-order queries on trees”. In: 14.4 (Nov. 2013), pp. 1–12. DOI: 10.1145/2528928.

[13] Baruch Schieber and Uzi Vishkin. “On Finding Lowest Common Ancestors: Simplification and Parallelization”. In: 17.6 (Dec. 1988), pp. 1253–1262. DOI: 10.1137/0217079.