Algorithms for regular languages that use algebra

Mikołaj Bojańczyk*
University of Warsaw

ABSTRACT

This paper argues that an algebraic approach to regular languages, such as using monoids, can yield efficient algorithms on strings and trees.

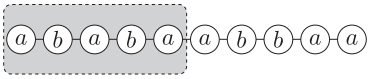
1. INTRODUCTION

A practically minded reader might have doubts about using monoids and other algebras for regular languages, instead of automata. The goal of this paper is to show that algebra is actually quite straightforward and can be practically useful in algorithms for words and trees.

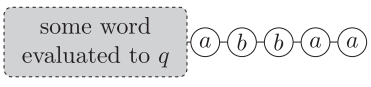
2. MONOIDS

A monoid is a method of recognizing regular languages, which is an alternative to automata and regular expressions. We illustrate the differences between recognizing a language with a deterministic finite automaton and a monoid.

In a deterministic automaton, a state $q \in Q$ represents a prefix of an input word:

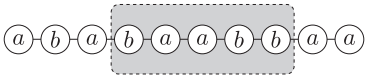


The state captures all the relevant information that the automaton needs to know about the prefix. This can be stated as the following compositionality principle: swapping a prefix with another prefix that gives the same state does not affect the run of an automaton on the remaining part of the input. Therefore, a prefix can be abstracted away as just its state:



*Supported by FET-Open grant agreement FOX, number FP7-ICT-233599. I would like to thank Jakub Łącki and an anonymous referee for comments that improved this paper.

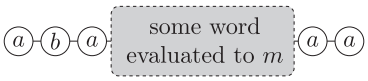
In the monoid approach, instead of an automaton, we have a function α, which maps every word w to an element of a set M, called the monoid. The idea is that the function α can be applied not just to prefixes of the input, but also to infixes:



The values assigned by the function α to an infix should capture all the relevant information about the infix. This can be stated as the following compositionality principle: swapping an infix with another infix of the same value does not affect the value of the whole input. In other words,

$$\alpha(v) = \alpha(v') \quad \text{implies} \quad \alpha(w_1 v w_2) = \alpha(w_1 v' w_2)$$

holds for all words $v, v', w_1, w_2 \in A^*$. A function that is compositional in the above sense is called a *monoid morphism*. In a monoid morphism α, an infix of the input can be abstracted away by its value under the monoid morphism:



We say that a monoid morphism $\alpha : A^* \to M$ recognizes a language $L \subseteq A^*$ if there is some set $F \subseteq M$ of accepting elements such that a word belongs to L if and only if it is mapped to F. A monoid morphism can recognize several languages, depending on the choice of F.

Examples of monoid morphisms and languages that they recognize include:

- The function $\{a, b\}^* \to \{0, 1\}$ which maps a word to 1 if and only if it contains some a. If we choose the accepting set F as $\{0\}$, then the morphism recognizes the language b^*.
- The function which maps a word to its length. The monoid used is not finite. This morphism

recognizes, e.g. the set of words of prime length. We are not interested in this kind of monoid morphism, because we only care about finite monoids.

- For $n \in \mathbb{N}$, the function

$$\alpha_n : \{a,b\}^* \to \{a,b\}^{\leq n}$$

which maps words of length $< n$ to themselves, and longer words to their prefix of length n. This morphism recognizes the language "the n-th letter is a". The set $\{a,b\}^{\leq n}$ is exponential in n; it is not difficult to see that this language cannot be recognized using a smaller set. A deterministic automaton for this language needs only $n+2$ states, since it corresponds to the function

$$\beta_n : \{a,b\}^* \to \{a,b\} \cup \{0,\ldots,n-1\}$$

which maps words of length $< n$ to their length, and longer words to their n-th letter. The function β_n is not a monoid morphism, because

$$\alpha(a^{n-1}) = \alpha(b^{n-1}) \quad \text{but} \quad \alpha(aa^{n-1}) = \alpha(ab^{n-1})$$

A corollary of compositionality is that for every words w_1, w_2, the value $\alpha(w_1 w_2)$ depends only on the values of $\alpha(w_1)$ and $\alpha(w_2)$. The operation

$$(\alpha(w_1), \alpha(w_2)) \in M^2 \qquad \mapsto \qquad \alpha(w_1 w_2) \in M$$

is called the *monoid operation* in M. If the monoid morphism is surjective, the operation is defined on all pairs in M^2. It is not difficult to see that because of compositionality, the monoid operation must be associative, which means that the result of the monoid operation for a sequence $m_1, \ldots, m_k$ does not depend on its bracketing. Finally, the image of the empty word is a neutral element for the monoid operation.

A monoid morphism is uniquely represented by: the images of single letters and the empty word, and the multiplication table for the monoid operation.

From automata to monoids and back. Monoid morphisms are just another way of talking about regular languages.

THEOREM 1. *A language $L \subseteq A^*$ is regular if and only if it is recognized by a morphism into a finite monoid.*

PROOF. Suppose that L is recognized by a monoid morphism $\alpha : A^* \to M$. Then one constructs an automaton, with states M, which maps every infix of the input to its value under α.

A bit more effort is required to go from an automaton to a monoid. The construction even works for nondeterministic automata. Consider a nondeterministic automaton with states Q. Consider a function which maps a word $w \in A^*$ to the set

$$\{(p,q) \in Q^2 : \text{ some run over } w \text{ goes from } p \text{ to } q\}.$$

It is not difficult to see that this mapping is compositional in the monoid sense. □

As seen in the above proof, the conversion from a monoid to a deterministic automaton, is linear. In the other direction, sometimes the exponential explosion from the theorem's proof cannot be avoided, as witnessed by the example "the n-th letter is a". Languages, for which the monoid has approximately the same size as a nondeterministic automaton include "the word contains at least n letters a".

The exponential blowup from automata to monoids is a problem for some algorithms; but it will not be a problem for the algorithms presented in this paper.

Expository Note. Typically, monoids are introduced as follows. One defines a monoid as an abstract algebraic structure, which is a carrier M equipped with an associative binary operation and a neutral element. Then one observes that the set of all words is a monoid. Finally, one defines a monoid morphism $\alpha : A^* \to M$ to be any function which preserves the structure of a monoid, namely the monoid operation and the monoid identity.

In this paper, a monoid is presented in the opposite order: we have seen that any surjective mapping from words to a set M which is compositional in the appropriate sense uniquely determines a structure of a monoid in its image M. The reason for this choice is that sometimes it is easier to think of a compositional mapping than an abstract algebraic structure.

3. ALGORITHMS FOR WORDS

As explained in the previous section, a monoid morphism assigns values to all infixes of an input word, and not just the prefixes. This means that a monoid morphism is a more flexible structure, and it is better suited to some algorithms. We illustrate this on several examples.

3.1 Incremental updates

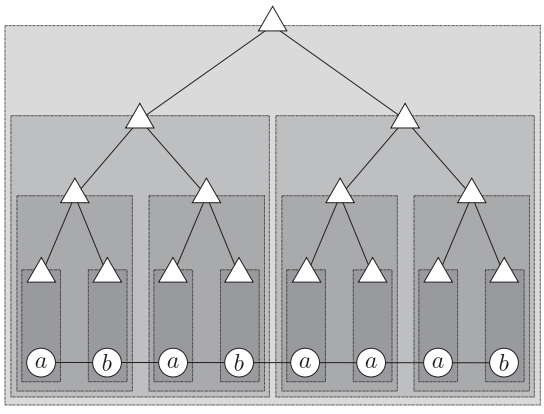
We begin with a very straightforward algorithm, which introduces a data structure that is heavily used in the rest of the paper. The data structure is a hierarchical decomposition of the input, decorated by values of a monoid morphism.

Here is the problem we want to solve. Let $L \subseteq A^*$ be a regular language of words. We begin with some initial word in $a_1 \cdots a_n \in A^*$. A user produces a sequence of edits of the form "change the label of letter $i \in \{1, \ldots, n\}$ to a". We want an algorithm which can tell, after a sequence of edits, if the current state of the word belongs to L. This problem can be solved with linear preprocessing and logarithmic cost per edit.

THEOREM 2. *Let $L \subseteq A^*$ be a regular language. There is an algorithm which:*

1. *inputs an initial word and builds a data structure in linear time;*
2. *receives a sequence of edits and updates the data structure in logarithmic time for each edit;*
3. *can tell in constant time, using the data structure, if the word after the edits belongs to L.*

PROOF. The algorithm uses a straightforward tree decomposition approach. Let $a_1 \cdots a_n \in A^*$ be the initial word. Divide the set of all positions in the word into two infixes, of approximately the same lengths, and then do this division recursively for the infixes. As a result, we get a tree decomposition of the initial word, as depicted in the following picture, where the circles denote positions in the word, and the triangles denote nodes in the tree (and nodes correspond to infixes):



.

In general, if the length of the initial word is not a power of two, then not all leaves have the same depth, but the maximal depth is still logarithmic.The tree has at most $2n$ nodes and can be computed in linear time (of course, we store the infixes by keeping their first and last positions).

Suppose that α is a monoid morphism which recognizes the language L. Such a monoid morphism exists by Theorem 1. For a node x of the tree, let us denote by w_x the infix of the word that corresponds to x. Because a node x with children y and z satisfies

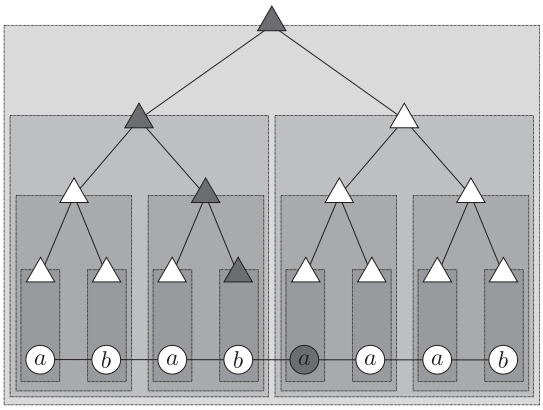
$$\alpha(w_x) = \alpha(w_y) \cdot \alpha(w_z), \tag{1}$$

we can do leaf-to-root pass through the tree t and compute in time linear in $|w|$ the values

$$\{\alpha(w_x)\}_{x \in nodes(t)}.$$

The tree t together with the values (1) is our data structure. We have just argued that it can be computed in linear time.

Consider an edit operation, which changes the label of letter $i \in \{1, \ldots, n\}$. The infixes in the data structure that contain i form a root-to-leaf path in the tree t, which is illustrated by a darker shade of gray in the following picture:



This path has logarithmic length. We only need to change the labels for these infixes, in a leaf-to-root pass. We can do this in logarithmic time using (1).

Finally, if we want to know if the current word belongs to the language L, we just need to look at the root of the tree, and see if the monoid element stored there belongs to the accepting set. □

When describing the running time of the algorithm in Theorem 2, we assumed that the language L was fixed. Suppose now that the language L is also given on the input, and represented by a nondeterministic automaton with states Q. What is the running time of the algorithm, in terms of both the input word $a_1 \cdots a_n$ and the state space Q? We claim that the data structure is built in time

$$poly(Q) \cdot O(n),$$

and each update operation is processed in time

$$poly(Q) \cdot \log n.$$

This is despite the fact that the algorithm uses monoids, and that monoids can be exponentially

larger than automata. The reason is that the algorithm does not need to compute the whole monoid. It only needs to store elements of the monoid in the tree nodes; and the space to store a monoid element is logarithmic in the size of the monoid, and therefore polynomial in Q. The algorithm also needs to consult the multiplication table of the monoid, but this multiplication table can be generated on-the-fly, in time polynomial in Q.

3.2 Evaluation of binary queries

In this section, we adapt the algorithm from Theorem 2 to evaluate binary queries on words.

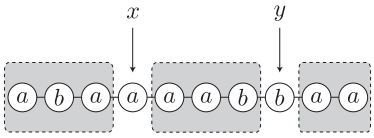
Binary queries. A binary query on words over alphabet A is a function φ, which maps every word $w \in A^*$ to a set of pairs of nodes in w. Typical queries include:

- Select pairs $x \leq y$ such that there is at least one a in the interval $\{x, \ldots, y\}$.
- Select pairs $x \leq y$ such that there an even number of a's in the interval $\{x, \ldots, y\}$.

In this paper, we are interested in regular binary queries, which are the query equivalent of regular languages. Just like for regular languages, there are multiple ways of defining regular queries. We use a definition based on monoids.

Binary query recognized by a monoid morphism. Let x, y be two distinct positions in a word w. There are two possible scenarios:

1. x is before y. In this case, the positions of the word can be partitioned into: x, y, and three infixes as in the picture below:



2. The second scenario is when x is after y. A decomposition similar to the one in the first scenario exists.

Suppose that $\alpha : A^* \to M$ is a monoid morphism. We define the α-type of a pair (x, y) of distinct positions in a word $w \in A^*$ to be the following information: which scenario holds, what are the labels of x and y, what are the values under α of the three infixes in the scenario. The α-type belongs to

$$\{<, >\} \times A^2 \times M^3$$

A monoid morphism is said to recognize a binary query if one can choose a subset F of accepting α-types, such that for every word and every pair of positions, the pair of positions satisfies the query if and only if its α-type belongs to F.

We say that a binary query is regular if it is recognized by some monoid morphism. The examples at the beginning of Section 3.2 are regular. Also, the class of regular binary queries coincides with the class of binary queries that can be defined by formulas of monadic second-order logic with two free first-order variables.

THEOREM 3. *Let φ be a regular binary query. There is an algorithm which:*

1. *inputs a word $w \in A^*$ and builds a data structure in linear time;*
2. *using the data structure, answers in logarithmic time questions of the form: is the pair (i, j) selected by φ?*

PROOF. Suppose that the binary query is recognized by a monoid morphism $\alpha : A^* \to M$. We use the same data structure as in Theorem 2.

The key observation is the following one: every infix v of the input word can be decomposed as a concatenation

$$v = v_1 \cdots v_k$$

of at most logarithmically many infixes $v_1, \ldots, v_k$ that correspond to nodes in the data structure. This decomposition can be computed in logarithmic time, given the first and last position in the infix v. By this observation, we can use the data structure to compute the α-type of any pair of positions in logarithmic time. □

4. SIMON FACTORIZATION

The algorithms we have seen so far used a tree-like decomposition of the input word, of logarithmic depth. What if we could have a constant depth decomposition, with the depth only depending on the monoid morphism, and not on the input word?

In this section, we provide such a constant depth decomposition. The underlying result is called the *Simon Factorization Forest Theorem*. This section is where monoids and their theory start to do some real work.

Simon factorization trees. Let $\alpha : A^* \to M$ be a monoid morphism. Let w be a word with n positions. A Simon α-factorization tree is similar to the tree from Theorem 2 in the following ways:

- Every node of x of the tree corresponds to an infix w_x of the word w. The leaves correspond to single letters; the root corresponds to the whole word w.
- If a node x has children $x_1, \ldots, x_n$ then
$$w_x = w_{x_1} \cdots w_{x_n}.$$
- Every node x stores the value $\alpha(w_x) \in M$.

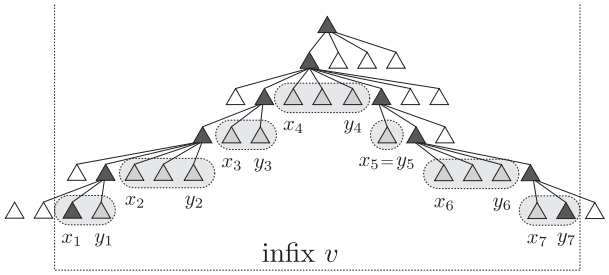
The difference is that in a Simon factorization tree, a node can have more than two children. Having an unbounded number of children is necessary if we want to have trees of constant depth for words of unbounded length. In a Simon factorization tree, there is a restriction for nodes x with three or more children:

- If x has at least three children $x_1, \ldots, x_n$, then
$$\alpha(w_{x_1}), \ldots, \alpha(w_{x_n})$$
are the same element, call it $m \in M$. Furthermore, m is idempotent, which means that $m \cdot m = m$. It follows that
$$\alpha(w_x) = \alpha(w_{x_1}) \cdots \alpha(w_{x_n}) = m \cdots m = m.$$

Consider a Simon α-factorization tree as in the definition above. Let x and y be siblings such that y is to the right of x. Let
$$x = z_1, \ldots, z_k = y$$
be all of the siblings between x and y. Define
$$w_{x \ldots y} = w_{z_1} \cdots w_{z_k}.$$
Observe that the monoid element $\alpha(w_{x \ldots y})$ can be computed in constant time; as a function of x and y. Indeed, when $k = 1, 2$, then $\alpha(w_{x \ldots y})$ is $\alpha(w_x)$ and $\alpha(w_x) \cdot \alpha(w_y)$, respectively. The more interesting case is when $k \geq 3$. In this case, the parent of x and y has at least three children. Let m be the (idempotent) monoid element stored in the parent. By the definition of Simon factorization trees,
$$\alpha(w_{x \ldots y}) = \alpha(w_{z_1}) \cdots \alpha(w_{z_k}) = m.$$

LEMMA 1. *For every infix v one can compute a sequence of sibling pairs*
$$(x_1, y_1), \ldots, (x_m, y_m)$$
such that
$$v = w_{x_1 \ldots y_1} \cdot w_{x_2 \ldots y_2} \cdot \ldots \cdot w_{x_m \ldots y_m}$$
The number of sibling pairs and the time to compute them is proportional to the depth of the tree.

PROOF BY PICTURE.



□

COROLLARY 1. *The value under α of an infix can be computed in time linear in the depth of the tree*[1].

PROOF. We use the decomposition of the infix from Lemma 1. For each sibling interval, its value under α can be computed in constant time: either it corresponds to at most two nodes, or its value is the same as the value in the parent of the interval. □

Constant depth. The key result about Simon factorization trees is that they can be built so that their height depends only on the monoid, and not the length input word:

THEOREM 4. *Let $\alpha : A^* \to M$ be a monoid morphism. Every word $w \in A^*$ has a Simon α-factorization tree of depth at most $3|M|$, which can be computed in time $poly(M) \cdot |w|$.*

The proof of this theorem, although not difficult, relies on results about monoids that cannot be easily recovered by treating monoids as a decoration on automata.

From Corollary 1 and Theorem 4 it follows that the algorithm from Theorem 3 can be improved from logarithmic time to constant time query evaluation:

THEOREM 5. *Let φ be a regular binary query. There is an algorithm which:*

1. *inputs a word $w \in A^*$ and builds a data structure in linear time;*
2. *using the data structure, answers in constant time questions of the form: is the pair (i, j) selected by φ?*

PROOF. The same proof as for Theorem 3, but use Corollary 1 to compute the values of infixes. □

[1]By adding accelerating pointers, this can be improved to time logarithmic in the depth of the tree, see [4]

Incremental updates. We have just shown how to use Theorem 4 to improve the algorithm for binary query evaluation. What about the algorithm for incremental updates? Could Simon factorization trees be used to get an algorithm that processes edits in constant time? Unfortunately [6] shows that some languages require at least

$$\frac{\log n}{\log \log n}$$

operations per edit.

4.1 References

A survey on the Simon factorization theorem, and some other applications for algorithms, can be found in [3]. The original version of Theorem 4 is from [9]. The $3|M|$ bound on the depth of the factorization tree, which is optimal, comes from [7]. A version of the theorem where the factorization tree is constructed by a deterministic left-to-right automaton is shown in [5]. The case when the monoid is obtained from a nondeterministic automaton is studied in [4].

5. REGULAR TREE LANGUAGES

We now move from regular languages of words to regular languages of trees. We used node-labelled, unranked (which means that there is no bound on the number of children), sibling-ordered trees, which are a common model for XML documents.

Nondeterministic tree automata. A *nondeterministic tree automaton* is given by the following ingredients:

1. An *input alphabet* A;
2. A set of states Q, with a distinguished accepting subset $F \subseteq Q$ of *root accepting* states.
3. A finite set δ of *transitions*. Each transition is a triple (q, a, L) where $q \in Q$, $a \in A$, and $L \subseteq Q^*$ is a regular language.

Of course, when representing such an automaton, one needs to choose some representation for the regular word languages in the transitions, e.g. by using regular expressions or maybe nondeterministic word automata. This choice of representation influences the complexity of algorithms that deal with the automata.

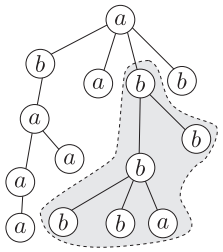
Consider an input tree t. An *run* of such an automaton is a labeling $\rho : nodes(t) \to Q$ such that for every node x with children $x_1, \ldots, x_n$ written from left to right, there exists a transition $(q, a, L) \in \delta$ such that the run maps x to q, the label of x is a, and the language L contains the word $\rho(x_1) \cdots \rho(x_n)$. In the special case when x is a leaf, the language L should contain the empty word. This special case explains why initial states are not needed in the definition of the automaton.

A tree is *accepted* if there is a run where the root is mapped to a root accepting state. The language *recognized* by an automaton is the set of accepted trees. A tree language is called *regular* if it is recognized by some automaton.

Just like for regular word languages, the class of regular tree languages is very robust and can be described in many other equivalent ways, including deterministic/alternating automata, Myhill-Nerode equivalence and monadic second-order logic[2].

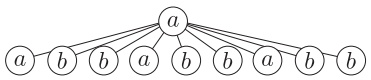
6. FOREST ALGEBRA

Just as in the case of words, an automaton can be seen as a device which assigns a value to parts of its input. In the case of nondeterministic tree automaton, the parts which get a value are subtrees, such as in the following picture:



Since we are using a nondeterministic automaton model, one should think of a subtree as being mapped to a set of states.

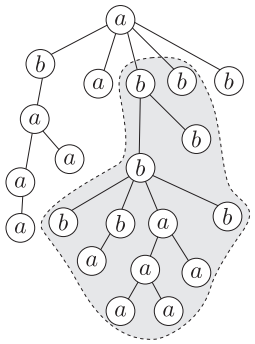
Forests. Subtrees are not general enough for the purposes of some of the algorithms we use. They suffer from the same problems as prefixes (as opposed to infixes) for words, together with some new problems. For instance, if you are only allowed to use subtrees, then how are you going to hierarchically decompose a tree with one root and many children, such as the one below?



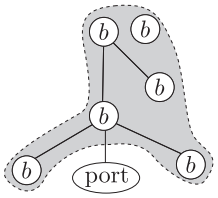
That is why our basic object of interest will not be a subtree, but an (ordered) forest, which is an ordered

[2]One of the choices of definition that *does not* lead to regular tree languages is the very natural model of deterministic root-to-leaves automata. Deterministic leaves-to-root automata, on the other hand, are equivalent to nondeterministic ones, and therefore to regular tree languages.

sequence of trees. In other words, a forest is a tree with multiple (ordered) roots. Inside a bigger forest we can find a smaller forest, by distinguishing a set of nodes called *forest zone*. A forest zone is a set of nodes in a forest which is closed under descendants, and such that the roots of the zone (which means the least nodes with respect to the ancestor relation) form a sequence of consecutive siblings. Here is a picture of a forest zone inside a tree

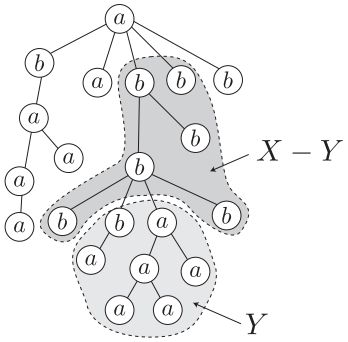


Contexts. Forests alone are also not general enough for the purposes of our algorithms. Recall that in Theorem 2, we used a data structure where each infix was split into two infixes, and so on recursively. This will not work for forests — for instance a tree cannot be split into two forests in any way. That is why we use a second kind of object, which is called a *context*. Formally, a context is a forest with one distinguished leaf, which is called a *port*, as in the following picture:

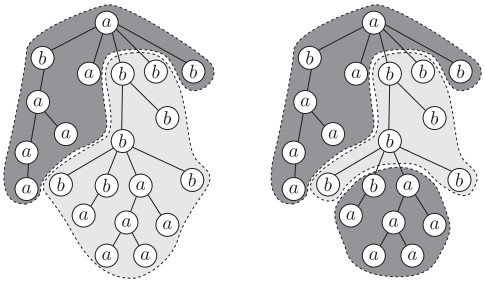


The idea is that the port can be filled by any forest. After the port in the picture above is filled by a forest with n roots, then parent of the port will have $n + 2$ children (because there are already two siblings of the port).

A context can be seen as a part of a forest, by distinguishing a set of nodes called *context zone*. A context zone inside a forest is defined as a difference of two forest zones $X - Y$. Here is a picture:



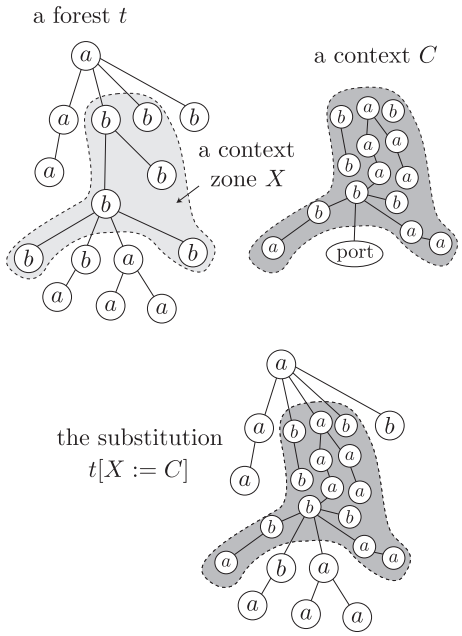
Partitioning zones. We claim that forest and context zones are general enough to do decompositions. As a first example, consider the complement of a (forest or context) zone. Although the complement is not necessarily a zone itself, it can be partitioned into at most two zones. The following pictures illustrates a forest (respectively, context) zone in light gray, and its complement in darker gray.



Generally speaking, since zones are sets of nodes, we can do the boolean operations on them: union, intersection and complementation. Although the result of a boolean combination of zones is not necessarily a zone itself, it can be partitioned into a small number of zones, as the following lemma shows.

LEMMA 2. *A boolean combination of n zones can be partitioned into $O(n)$ zones.*

Substitution. Suppose that we have a forest t and a context C. If we distinguish a context zone X inside t, then we can replace the contents of that context zone by C, with the result being called $t[X := C]$. This process is illustrated below:



In general, there are four kinds of substitution: the enclosing object can be a context or a forest, and the substituted object can be a context or forest.

Forest algebra morphism. We now introduce forest algebra. Forest algebra is for forests and contexts, as monoids are for infixes. Like we did for monoids, we focus on the morphisms. A forest algebra morphism over the alphabet A consists of two sets (H, V) and functions:

1. a function α_H from forests over A to H;
2. a function α_V from contexts over A to V;

which are compositional in the following sense. Let t be a forest, X a context zone inside t, and let C and C' be two contexts. Then

$$\alpha_V(C) = \alpha_V(C') \quad \text{implies}$$
$$\alpha_H(t[X := C]) = \alpha_H(t[X := C']).$$

Likewise for the other three kinds of substitutions. In other words, all we need to know about a zone (forest or context) is its value under α_H or α_V.

A forest algebra morphism can be used to recognize a set of forests. This is done by distinguishing an accepting subset $F \subseteq H$; the recognized language is then the set of forests that are mapped to F by the morphism. If you want to recognize a tree language, you should make sure that only trees are mapped to elements of F.

The following theorem shows that forest algebras, as a recognizing device for tree languages, are equivalent to automata.

THEOREM 6. *A tree language is recognized by a nondeterministic tree automaton if and only if it is recognized by a forest algebra morphism.*

As in the word case, there is a (singly) exponential blowup when going from a nondeterministic automaton to a forest algebra morphism.

6.1 References

For a survey on regular tree languages with a database angle, see [8]. A discussion of forest algebra and other algebras for trees can be found in [2].

7. ALGORITHMS FOR TREES

In this section, we show how forest algebra can be used to generalize from words to trees the algorithms that we have seen in Section 3.

7.1 Hierarchical decompositions of forests

The data structure that we used in Section 3 was a hierarchical decomposition, where the word was split into halves, quarters, and so on. We now show that a similar decomposition is possible for forests.

LEMMA 3. *Every (forest or context) zone X can be partitioned into at most four (forest or context) zones such that each of the parts has at most $2/3 \cdot |X|$ nodes.*

PROOF. Let n be the size of X.

We say that a node $x \in X$ is small if at most one third of the nodes of X are descendants of x. Suppose that not all nodes in X are small (the degenerate case when all nodes are small is treated the same way). There must be a node $x \in X$ which is not small, but which has only small children, call them $x_1, \ldots, x_k$. For each $i \in \{1, \ldots, k\}$, define D_i to be the nodes in X that are (not necessarily strict) descendants of one of the nodes $x_1, \ldots, x_i$. The set D_1 has at most $n/3$ nodes and the set of D_k has at least $n/3 - 1$ nodes, because otherwise the parent x would be small. Take i to be the first index such that D_i has at least $n/3$ nodes. Then

$$\frac{n}{3} \quad \leq \quad |D_i| \quad < \frac{2n}{3}$$

because the difference between consecutive sizes of D_i is at most $n/3$.

We now show the decomposition.

- X is a forest zone. We partition X into the forest zone D_i and the context zone $X - D_i$. Both parts have between a third and two thirds the nodes.
- X is a context zone, which means that X is the difference $Y - Z$ of two forest zones. There are two cases to consider.

– D_i is a context zone, which means that $Z \subseteq D_i$. In this case we decompose X into two context zones D_i and $X - D_i$.

– D_i is a forest zone. In this case, the set $X - D_i$, which itself is not a zone, can be partitioned into at most three zones, all of which have at most one third of the nodes.

□

COROLLARY 2. *Let t be a forest. There exists exists a tree s, where each node x is labelled by a (forest or context) zone in t such that*

- *The root of s is labelled by the set of all nodes in t.*
- *Leaves of s are labelled by singletons.*
- *Let x be a non-leaf node of s, and let $x_1, \ldots, x_k$ be its children. Then $k \leq 4$, and the zones in the labels of $x_1, \ldots, x_k$ form a partition of the zone in the label of x.*
- *The depth of s is logarithmic in the size of X.*

PROOF. Apply Lemma 3 to the zone containing all nodes, and then recursively apply the same lemma to the resulting zones. Each time, the size of the zone decreases to at most $2/3$, so the process creates a tree of depth at most $\log_{3/2} n$. □

7.2 Incremental updates

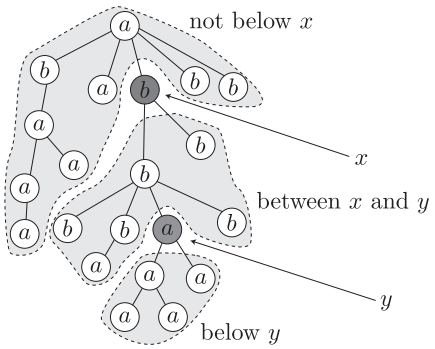
Using the hierarchical decomposition from Corollary 2, we can generalize to trees the algorithms from Section 3. The algorithm for incremental updates for tree languages is exactly the same as in the case of words. We just use the data structure given in Corollary 2.

7.3 Binary queries on forests

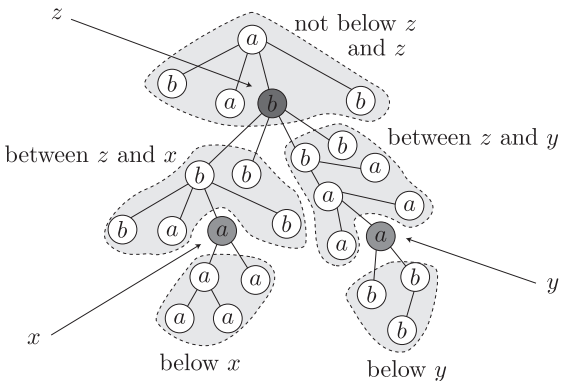
As in Section 3, we can use the hierarchical decomposition to evaluate binary queries. Most of the discussion in this section is devoted to defining regular binary queries for trees; once these are defined, a straightforward application of the hierarchical decomposition can be used to evaluate them.

In this section, we consider regular binary queries on trees (actually, on forests). Let x, y be two nodes in a forest t. There are four possible scenarios for their relative positions in the forest.

1. The first scenario is when x is an ancestor of y. In this case, the nodes of the forest can be partitioned into: x, y, two context zones and one forest zone, as in the picture below:



2. The second scenario is when x is a descendant of y. A decomposition similar to the one in the first scenario can be found.
3. The third scenario is when x and y are incomparable with respect to the descendant relation, but x comes first in document order. Let z be the closest common ancestor of x and y. In this case the complement of $\{x, y\}$ can be partitioned into three context zones and two forest zones, as in the following picture:



4. There is a degenerate variant of the third scenario, when x and y are descendants of different roots, and z does not exist. In this degenerate case when z does not exist, the uppermost context zone is empty.
5. The fifth scenario is when x and y are incomparable with respect to the descendant relation, but y comes first in document order. A decomposition similar to the one in the third scenario can be found.
6. The sixth scenario is the degenerate version of the fifth.

Suppose that α is a forest algebra morphism. Then type α-type of a pair of nodes (x, y) in a forest t consists of the following information:

- the labels of x and y;
- which scenario holds;
- for the appropriate scenario, the values assigned by α to the zones

A query is said to be recognized by α if there is a set F of α-types such that for every forest and pair of nodes (x, y), the pair (x, y) is selected by the query if and only if its α-type belongs to F. A query is called *regular* if it is recognized by some α. A similar definition makes sense queries of higher arities, such as three or four, but the number of scenarios grows with the arity of the query.

The above definition of regular queries coincides with more standard definitions of regular binary queries for trees or forests, such as queries defined by a formula of monadic second-order logic with two free variables.

THEOREM 7. *Let φ be a regular binary query on forests. There is an algorithm which:*

1. *on input forest t, builds a data structure in linear time;*
2. *using the data structure, answers in logarithmic time questions of the form: is a pair of nodes (x, y) selected by φ?*

PROOF. The same proof as for Theorem 3, except using the decomposition from Corollary 2. □

7.4 References

The first algorithm for incremental updates of regular tree languages is from [1]. The algorithm here is a slight improvement over [1], because for a tree with n nodes it runs in time $O(\log n)$ and not $O(\log^2 n)$.

8. REFERENCES

[1] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[2] Mikołaj Bojańczyk. Algebra for trees. In *Handbook of Automata Theory*. European Mathematical Society Publishing House. To appear.

[3] Mikołaj Bojańczyk. Factorization forests. In *Developments in Language Theory*, pages 1–17, 2009.

[4] Mikołaj Bojańczyk. and Paweł Parys. Efficient evaluation of nondeterministic automata using factorization forests. In *ICALP (1)*, pages 515–526, 2010.

[5] Thomas Colcombet. Factorization forests for infinite words and applications to countable scattered linear orderings. *Theor. Comput. Sci.*, 411(4-5):751–764, 2010.

[6] Gudmund Skovbjerg Frandsen, Johan P. Hansen, and Peter Bro Miltersen. Lower bounds for dynamic algebraic problems. *Inf. Comput.*, 171(2):333–349, 2001.

[7] Manfred Kufleitner. The height of factorization forests. In *MFCS*, pages 443–454, 2008.

[8] Frank Neven. Automata theory for xml researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[9] Imre Simon. Factorization forests of finite height. *Theor. Comput. Sci.*, 72(1):65–94, 1990.