Information and Computation 211 (2012) 1–28

Contents lists available at SciVerse ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Regular languages with variables on graphs

Simone Santini [1]

Universidad Autónoma de Madrid, Spain

ARTICLE INFO

Article history:
Received 14 April 2010
Revised 29 December 2010
Available online 1 December 2011

Keywords:
Regular expressions
Graphs
Paths in graphs
Matching

ABSTRACT

This paper presents a pattern language based on regular expressions that allows the introduction of *variables* that can be instantiated to portions of the path that matches the expression. The paper will define a simple syntax for the language and its formal semantics. It will also study a modification of finite state automata that, through the introduction of *actions* on transitions, allows the variables to be instantiated while matching the expression. Finally, the paper will show that the problem of answering queries with variables is inherently harder than simple matching, essentially because, even for fairly simple expressions, the size of the results can be exponential in the size of the graph. The class of expressions and a class of graphs for which query answering is polynomial will be identified, and a processing algorithm for these expressions based on the intersection graph will be provided and analyzed.

1. Introduction

The more and more pressing necessity to extend data bases beyond the reach of the standard relational model has created a mounting interest in alternative data structures, especially in trees and graphs, which are thought to better reflect the modeling needs of a data glut dominated by hierarchies and by the web. A consequence of this interest has been a burgeoning activity in the design of query languages capable of handling these structures. There is a broad family of query languages for trees and graphs [1–4] many of which are based on modification or restrictions of regular expressions [5–8].

Some query languages for trees are based on the *world wide web consortium* standard *xpath* [9], in general extending it to give it the expressivity of first order logic [10] or, by including regular expressions into them, of first order logic with certain transitive closure operations [11]. Languages based on regular expressions are especially interesting, being the ones for which expressivity and optimization problems have been thoroughly studied [12,13]. Regular expressions are often studied as language recognizers and, in order to be adapted for use as a query formalism, they have to be equipped with a suitable semantics, one that specifies which results are returned and how. Query languages based on xpath are based on what we might call a *pair of nodes* semantics, viz. one in which expressions return pairs of nodes such that the path between them matches the expression. This semantics might be an insufficient modeling tool when it comes to graphs since, while in a tree two nodes determine uniquely a path, in a graph this is no longer true. For graphs, a great deal can be gained by allowing the introduction of *variables* in the language so that the path(s) that matches an arbitrary sub-expression can be returned as a result. I shall present the syntax and semantics of such a language, based on regular expressions, and show that answering queries using variables is inherently harder than just matching the expression, but easier than other forms of variable usage such as backreferencing [14]. In particular, while matching regular expressions on directed acyclic graphs

E-mail address: simone.santini@uam.es.

[1] The author was supported in part by the Ministerio de Educación y Ciencia under the *Ramón y Cajal* programme and by *Consejería de Educación, Comunidad Autónoma de Madrid*, under the grant CCG08-UAM/TIC/4303, *Búsqueda basada en contexto como alternativa semántica al modelo ontológico*.

0890-5401/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2011.10.010

(DAGs) is PTIME [15],[2] answering queries with variables can generate results of size exponential in the size of the graph and therefore intractable in time and space. The paper presents an algorithm for processing regular expressions with variables, and shows that for a certain class of expressions this algorithm has a polynomial running time.

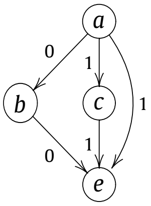
I shall put some restrictions on the kind of expressions we shall work with. In general, I shall only consider expressions for which, were it not for the presence of variables, an efficient deterministic or non-deterministic automaton exists. The reason for this restriction are methodological. It will be clear in the following that the presence of variables will not change the basic problem of *matching* a regular expression to a path, a tree, or a graph.[3] Variables will only add the complexity of having to collect results. Therefore, any problem that is hard (e.g. NP-complete) for regular expressions will continue to be hard once we add variables. We are therefore interested in the *added* complexity that variables introduce, that is, in those problems that would be tractable were it not for the presence of variables. Therefore, we shall restrict ourselves to tractable regular expressions, and determine what additional hypotheses we must make for them to remain tractable in the presence of variables.

As a modeling note, in this paper I shall consider trees and graphs with labels on the edges rather that on the vertices. In the case of trees, the placement of the labels is indifferent, since one can always pass from one model to the other by placing a vertex's label on the (unique) edge that enters the vertex. In the case of graphs, labels on edges provide a more general model, which subsumes the one with labels on the vertices.

2. Why variables

Many of the pattern languages proposed for trees return, as the result of a query, the *pairs of nodes* between which lies a path that satisfies the query; this is consistent with their semantics since, in their standard form, the satisfaction of a path formula ϕ is of the form $(T, n, m) \models \phi$, where n and m are nodes of T. This semantics, and the query results that it entails, are adequate for these languages since in a tree a pair of nodes is joined by at most one path, so whenever a query returns a pair (n, m), where m is a descendant of n, a path is uniquely identified.

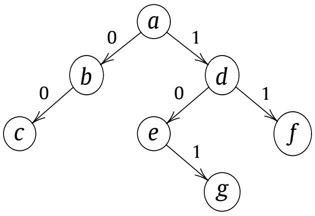
This kind of semantics, however, is often inadequate for graphs. Consider the regular expression $\phi = 11^*$ applied to the graph



(1)

There are three paths from a to e, but in the *pair of nodes* semantics the query would only return the pair (a, e), thus not providing the information that, of the three paths, only two satisfy the query. If on this graph we issued the query 00^*, the pair (a, e) would still constitute a result, but this time the only path that satisfies the query would be $[a, b, e]$, that is, the path that did not satisfy the previous one. The possibility of returning paths is in this case essential to discriminate between results.

Returning paths, or fragments of paths, can be useful in the case of trees as well. We might, for instance, execute the query 10^*1 on a tree, but be interested only in the portion of the path that matches 0^*. In the following tree



(2)

the path $[a, d, e, g]$ matches the expression but, in the scenario presented, we are only interested in the portion $[d, e]$, which matches the 0^* portion of it. The path $[a, d, f]$ also matches the expression, but the portion relative to the sub-expression 0^* is empty, so we wouldn't generate any result for it. On the other hand, the path $[a, b, c]$ does match the expression 0^*, but we would not consider it as a valid result, since it is not part of a path of the form 10^*1.

I shall present a language that allows the introduction of variables into regular expressions, thereby specifying what portion of the matching paths one wants to be returned as a result. I shall define the semantics of variable insertion and show that, with the addition of variables, processing the language is inherently harder than recognizing simple regular

[2] In fact, matching any graph on regular expressions is PTIME; in this paper, however, I shall consider only DAGs, so the weaker property will suffice.

[3] In this sense, regular expressions with variables are different from other kinds of expressions, such as regular expressions with backreferencing, in which variables are used to extend the class of languages recognized by the expressions. I shall consider this issue more in depth in Section 10.

expressions since, even in relatively easy cases the size of the result set is exponential in the size of the input (expression plus data graph). I shall identify certain restrictions on the form of the expression and of the graph that make the problem tractable.

3. The language

The syntax of the language is a straightforward extension of that of regular expressions. If ϕ is an expression, then $[x : \phi]$ is also an expression that assigns to the variable x the path that matches ϕ. With this addition, the grammar of the language is

$$\phi ::= a \mid \phi + \phi \mid \phi\phi \mid \phi^* \mid (\phi) \mid [v : \phi]$$

where a is a symbol of the input alphabet Σ, and v is an element of the set Θ of variable names. I assume that Σ and Θ are disjoint and that a variable name can appear only once in an expression. Relaxing the latter requirement would not bring any substantial change to the results of this paper, but it would complicate the formal definition of the semantics. That is, this restriction has been imposed more for the sake of clarity than as a real restriction of the class of expressions that we are considering. I shall show in Section 9 that if we take as definitional the *operational* semantics given by the automaton that recognizes the expressions, this restriction can be dropped without consequences.

I shall first introduce the general concepts and the model-theoretic semantics of the variable-free language, then introduce the concept of *environment* and use it to extend the semantics to patterns with variables. For the sake of simplicity, the model that I use is a simple graph: all the results in the following can be easily extended to multi-graphs at no other cost than a bigger investment in notation. In this paper, a graph is a 4-tuple $G = (V, E, \Sigma, \lambda)$, where $E \subseteq V \times V$ is the set of edges, Σ is the edge label alphabet, and $\lambda : E \to \Sigma$ is the labeling function. To this graph we can associate the relational structure $\underline{G} = (E, P_{a|a\in\Sigma})$, with $P_a = \{e \mid e \in E \wedge \lambda(e) = a\}$.

A *path* in a graph G is a list $\pi = [\pi_1, \ldots, \pi_n]$ such that

i) $\forall i.(1 \leqslant i \leqslant n \ \Rightarrow\ \pi_i \in V)$;
ii) $\forall i.(1 \leqslant i < n \ \Rightarrow\ (\pi_i, \pi_{i+1}) \in E)$.

Given j, k with $0 \leqslant j \leqslant k \leqslant n$, define the sub-paths $\pi^{|k} = [\pi_1, \ldots, \pi_k]$, $\pi^{j|} = [\pi_j, \ldots, \pi_n]$, and $\pi^{j|k} = [\pi_j, \ldots, \pi_k]$.

Also, define the path *stitching* operation

$$\pi = \pi' \mathbin{⧖} \pi'' \quad \equiv \quad \exists k.\bigl(\pi' = \pi^{|k} \ \wedge\ \pi'' = \pi^{k|}\bigr) \tag{3}$$

Note that the last vertex of π' is the same as the first vertex of π''; if this is not the case, the operation is not defined. This overlap between π' and π'' is necessary because paths are defined on vertices, while labels are placed on edges, so that a regular expression matches sequences of edges: two consecutive edges share a vertex, which is the vertex that overlaps in the stitch. With these operations in place, we can define the general sub-path relation

$$\pi' \subseteq \pi \quad \equiv \quad \exists h, k.\bigl(\pi' = \pi^{h|k}\bigr) \tag{4}$$

and the proper sub-path relation

$$\pi' \subset \pi \quad \equiv \quad \pi' \subseteq \pi \quad \wedge \quad \pi' \neq \pi \tag{5}$$

The semantics of the variable-free portion of the language is fairly standard. Matching conditions are expressions of the form $(G, \pi) \models \phi$, where G is a graph, π a path in G, and ϕ an expression. The conditions are as follows:

$$\begin{aligned}
&(G,\pi) \models a \quad \equiv \quad \pi = [u, v] \quad \wedge \quad (u, v) \in P_a\\
&(G,\pi) \models \phi + \phi' \quad \equiv \quad (G,\pi) \models \phi \quad \vee \quad (G,\pi) \models \phi'\\
&(G,\pi) \models \phi\phi' \quad \equiv \quad \exists \pi'\pi''.\bigl(\pi = \pi' \mathbin{⧖} \pi'' \ \wedge\ \bigl(G,\pi'\bigr) \models \phi \ \wedge\ \bigl(G,\pi''\bigr) \models \phi'\bigr)\\
&(G,\pi) \models \phi^* \quad \equiv \quad \pi = \epsilon \vee \exists k.\bigl(\pi = \pi_1 \mathbin{⧖} \cdots \mathbin{⧖} \pi_k \ \wedge\ \forall i.\bigl(1 \leqslant i \leqslant k \ \Rightarrow\ (G,\pi_i) \models \phi\bigr)\bigr)
\end{aligned} \tag{6}$$

The language recognized by an expression ϕ is defined in the customary way as

$$L(\phi) = \bigl\{(G,\pi) \bigm| (G,\pi) \models \phi\bigr\} \tag{7}$$

or, when dealing with paths, simply as

$$L(\phi) = \bigl\{\pi \bigm| (\pi,\pi) \models \phi\bigr\} \tag{8}$$

3.1. Environments

In order to define the full semantics of the pattern language, we shall have to deal with variable assignments. Variables take their values in *environments* [16]. In this paper, we only need to deal with a fairly restricted class of environments, since all the variables are of the same type. If α is the data type of the vertices of the graph, then each variable is of type $[\![\alpha]\!]$: a list of lists of vertices, that is, a list of paths (I shall clarify shortly why variables are lists of paths instead of simple paths). So, if Θ is the set of variable names, an environment is a function $u \in \Theta \to [\![\alpha]\!]$, and $u(x) \in [\![\alpha]\!]$ is the value of the variable x in the environment u. The domain $D(u)$ of an environment u is the set of variables defined in it. If $u \in \Theta \to [\![\alpha]\!]$ is an environment, $x \in \Theta$ a variable name, and $q \in [\![\alpha]\!]$ a value, then the environment $(u|x \mapsto q)$ is defined as:

$$(u|x \mapsto q)(z) = \begin{cases} u(z) & \text{if } z \neq x \\ q & \text{otherwise} \end{cases} \tag{9}$$

Note that, according to this definition, even if x were defined in u, the assignment $x \mapsto q$ would prevail. If $u, v \in \Theta \to [\![\alpha]\!]$, the environment $(u|v) \in \Theta \to [\![\alpha]\!]$ is defined as

$$(u|v)(z) = \begin{cases} v(z) & \text{if } z \in D(v) \\ u(z) & \text{otherwise} \end{cases} \tag{10}$$

Note that if $D(u) \cap D(v) \neq \emptyset$, and the two environments differ in the intersection, then $(u|v) \neq (v|u)$.

The empty environment $\perp$ has no variables defined ($D(\perp) = \emptyset$), and, for all environments u, $(u|\perp) = (\perp|u) = u$. Given a set of variables $X = \{x_1, \ldots, x_n\}$, the null environment for X, ${}^{X}\square$ is the environment that assigns the empty list to all variables in X:

$${}^{X}\square = \bigl(x_1 \mapsto [\,] \bigm| \cdots \bigm| x_n \mapsto [\,]\bigr) \tag{11}$$

If ϕ is an expression, and $X(\phi)$ is the set of variables defined in ϕ, I shall use ${}^{\phi}\square$ as a shortcut for ${}^{X(\phi)}\square$. Finally, in the present case, in which the values of variables are lists, one can introduce the *environmental append*. Given two environments u, v, define the environment $u@v$ by $(u@v)(x) = u(x)@v(x)$, where @ is the usual list append operation.

Given an expression ϕ and a graph G, the semantics of ϕ is a function $[\![\phi]\!]$ that associates to G a set of pairs of the form (π, u), where π is a path and u an environment. Intuitively, the paths π are the paths of G that match the expression, and the environment u attached to π associates portions of π to the variables defined in the expression. Before introducing the formal definition of the semantics, I shall illustrate it with two examples, the second of which will clarify why the value of a variable is a list of paths.

Example 1. Consider applying the expression $\phi = 1^*[x : 1^*]$ to the graph

$$A \xrightarrow{1} B \xrightarrow{1} C \xrightarrow{1} D \tag{12}$$

and consider, for the sake of simplicity, only the path that starts in A and ends in D. Any suffix of this path can be assigned to the variable x, leaving the initial 1^* to match the first portion. For example, if we interpret the initial 1^* as matching the path $[A, B, C]$, then the second 1^* will match the sub-path $[C, D]$, and this will be the value assigned to x; if we interpret the initial 1^* as matching the path $[A, B]$, then the value assigned to x will be $[B, C, D]$. All these alternatives are valid, so that the result set for this expression will be

$$\begin{aligned} &\bigl\{\bigl([A, B, C, D], \bigl(x \mapsto [\![\,]\!]\bigr)\bigr), \bigl([A, B, C, D], \bigl(x \mapsto [\![D]\!]\bigr)\bigr), \bigl([A, B, C, D], \bigl(x \mapsto [\![C, D]\!]\bigr)\bigr), \\ &\quad \bigl([A, B, C, D], \bigl(x \mapsto [\![B, C, D]\!]\bigr)\bigr), \bigl([A, B, C, D], \bigl(x \mapsto [\![A, B, C, D]\!]\bigr)\bigr)\bigr\} \end{aligned} \tag{13}$$

Note that if there are no variables, the expression 1^*1^* can be reduced to 1^*, while no such reduction is possible with the presence of the variable x.

Example 2. Consider now the expression $\phi = (0^*[x : 1^*])^*$ and the graph

$$A \xrightarrow{0} B \xrightarrow{1} C \xrightarrow{1} D \xrightarrow{0} E \xrightarrow{1} F \xrightarrow{0} G \xrightarrow{0} H \tag{14}$$

The path A-H matches the expression as follows:

$$A \underbrace{\xrightarrow{0}}_{0^*} B \underbrace{\xrightarrow{1} C \xrightarrow{1}}_{x \mapsto 1^*} D \underbrace{\xrightarrow{0}}_{0^*} E \underbrace{\xrightarrow{1}}_{x \mapsto 1^*} F \underbrace{\xrightarrow{0} G \xrightarrow{0}}_{0^*} H \tag{15}$$

So that the portion that is assigned to the variable x is not an uninterrupted path, but is composed of two separate “chunks”. So, the portion of the result relative to this path (there are sub-paths of this one that also match the expression, which we do not consider in this example) is

$$\bigl\{\bigl([A, B, C, D, E, F, G, H], \bigl(x \mapsto \bigl[[B, C, D], [E, F]\bigr]\bigr)\bigr)\bigr\} \tag{16}$$

Formally, the semantics of a regular expression with variables ϕ is a function

$$\llbracket\phi\rrbracket : \Gamma(\alpha) \to \bigl\{[\alpha] \times \bigl(\Theta \to \llbracket\alpha\rrbracket\bigr)\bigr\} \tag{17}$$

where $\Gamma(\alpha)$ is the data type of graphs with vertices of type α. The function is defined by induction on the structure of the expression ϕ as follows:

$$\begin{aligned}
&\llbracket a\rrbracket(G) = \bigl\{(\pi, \bot) \bigm| (G, \pi) \models a\bigr\}\\
&\bigl\llbracket\phi + \phi'\bigr\rrbracket(G) = \bigl\{\bigl(\pi, \bigl({}^{\phi'}\square|u\bigr)\bigr) \bigm| (\pi, u) \in \llbracket\phi\rrbracket\bigr\}(G) \cup \bigl\{\bigl(\pi, \bigl({}^{\phi}\square|u\bigr)\bigr) \bigm| (\pi, u) \in \bigl\llbracket\phi'\bigr\rrbracket(G)\bigr\}\\
&\bigl\llbracket\phi\phi'\bigr\rrbracket(G) = \bigl\{\bigl(\pi \mathbin{⧖} \pi', \bigl(v@v'\bigr)\bigr) \bigm| (\pi, v) \in \llbracket\phi\rrbracket(G) \ \wedge\ \bigl(\pi', v'\bigr) \in \bigl\llbracket\phi'\bigr\rrbracket(G)\bigr\}\\
&\bigl\llbracket\phi^*\bigr\rrbracket(G) = \bigl\{(\pi_1 \mathbin{⧖} \cdots \mathbin{⧖} \pi_n, v_1@\cdots@v_n) \bigm| 1 \leqslant i \leqslant n \ \Rightarrow\ (\pi_i, v_i) \in \llbracket\phi\rrbracket(G) \ \wedge\ (G, \pi_1 \mathbin{⧖} \cdots \mathbin{⧖} \pi_n) \models \phi^*\bigr\}\\
&\bigl\llbracket[x : \phi]\bigr\rrbracket(G) = \bigl\{\bigl(\pi, \bigl(u|x \mapsto [\pi]\bigr)\bigr) \bigm| (\pi, u) \in \llbracket\phi\rrbracket(G)\bigr\}
\end{aligned} \tag{18}$$

I call this the *strict* semantics. In this variable assignment semantics, if the expression ϕ matches the empty path, the environment u assigns to the variable x the empty list. For example, if we have $\psi = a^*[x : b^*]$ and apply this expression to the graph

$$A \xrightarrow{a} B \xrightarrow{a} C \tag{19}$$

the result will be

$$\bigl([A, B, C], \bigl(x \mapsto \llbracket[]\rrbracket\bigr)\bigr) \tag{20}$$

In some cases we might not want this, but we might want the results to contain only the non-empty paths of x. In this case we shall modify the semantics as

$$\bigl\llbracket[x : \phi]\bigr\rrbracket(G) = \bigl\{\bigl(\pi, \bigl(u|x \mapsto [\pi]\bigr)\bigr) \bigm| (\pi, u) \in \llbracket\phi\rrbracket(G) \ \wedge\ \pi \neq []\bigr\} \tag{21}$$

I call this the *lax* semantics. In this paper I shall deal mostly with the strong semantics, although all the results of this paper do hold in both cases.

Both the strict and the lax semantics are intended to be extensions of the matching conditions to the case with variables. The following theorem shows that this is indeed the case (for the theorem, it is indifferent which one of the two semantics we are using):

Theorem 3.1.

$$\exists v.\bigl((\pi, v) \in \llbracket\phi\rrbracket(G)\bigr) \quad \Rightarrow \quad (G, \pi) \models \phi \tag{22}$$

Proof. The proof is by induction on the number of operations necessary to build ϕ.

If $n = 0$ then $\phi = a$, and the theorem is trivially true from the definition.

Suppose now that the theorem is true for n operations. We must distinguish three cases:

i) $\phi = \phi' + \phi''$:

$$\begin{aligned}
(\pi, v) \in{}& \bigl\llbracket\phi' + \phi''\bigr\rrbracket(G)\\
&\equiv \exists\hat{v}.(\pi, \hat{v}) \in \bigl\llbracket\phi'\bigr\rrbracket(G) \ \vee\ \exists\hat{v}.(\pi, \hat{v}) \in \bigl\llbracket\phi'\bigr\rrbracket(G) \quad \text{(by definition)}\\
&\equiv (G, \pi) \models \phi' \ \vee\ (G, \pi) \models \phi'' \quad \text{(by the inductive hypothesis)}\\
&\equiv (G, \pi) \models \phi' + \phi'' \quad \text{(by definition)}
\end{aligned} \tag{23}$$

ii) $\phi = \phi'\phi''$:

$$\begin{aligned}
(\pi, v) \in{}& \bigl\llbracket\phi'\phi''\bigr\rrbracket(G)\\
&\equiv \exists\pi', \pi'', v, v'.\bigl(\pi = \pi' \mathbin{⧖} \pi'' \ \wedge\ \bigl(\pi', v'\bigr) \in \bigl\llbracket\phi'\bigr\rrbracket(G) \ \wedge\ \bigl(\pi'', v''\bigr) \in \bigl\llbracket\phi''\bigr\rrbracket(G)\bigr)\\
&\equiv \exists\pi', \pi''.\bigl(\bigl(G, \pi'\bigr) \models \phi' \ \wedge\ \bigl(G, \pi''\bigr) \models \phi''\bigr)\\
&\equiv (G, \pi) \models \phi'\phi''
\end{aligned} \tag{24}$$

iii) $\phi = (\phi')^*$: in this case the definition of $\llbracket\phi\rrbracket(G)$ requires explicitly that any π such that $(\pi, v) \in \llbracket\phi\rrbracket(G)$ satisfy $(G, \pi) \models (\phi')^*$. □

4. The bad news

Matching regular expressions in graphs, even in the “pairs of nodes” semantics can be hard. In particular, while finding paths is polynomial [17], finding *simple* paths is NP-hard [15]. However, in some cases finding simple paths may not be quite as hard. One such case is that of DAGs: since in a DAG every path is simple, the problem has polynomial complexity. If we have variables, things are inherently harder. If the graph contains cycles, the query processing algorithm might not terminate, as the result set might be infinite. Consider the graph:

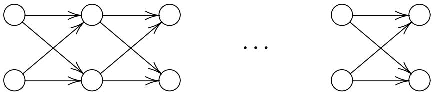
$$\begin{tikzcd} A \arrow[r, "0"] & B \arrow[r, "0"] \arrow[loop below, "1"] & C \end{tikzcd} \tag{25}$$

and the expression $0[x:1^*]0$. The only pair of vertices that matches the expression is (a,c), but the paths that, according to the semantics of the previous section, are assigned to the variable x are $[], [b], [b,b], [b,b,b], \ldots$. In this paper, I shall avoid the problem by dealing exclusively with directed acyclic graphs, for which the queries always return a finite result. Unfortunately, in this case too the problem is hard. Informally, the issue is that, in order to assign values to variables, the query processor will have to explore too many paths, and won’t be able to take any shortcuts, because the paths that we fail to analyze may have to be returned as a result. This entails that we may have queries whose result set grows exponentially with the size of the input. It is clearly impossible to compute these queries efficiently, since writing the results alone requires a time exponential in the size of the input. Let us consider a (possibly infinite) family of pattern expressions P, and a family of graphs $\mathcal{G}$ of varying sizes, parametrized by a size parameter $|G|$, $G \in \mathcal{G}$ (in general, I shall use the number of vertices in a graph as the family parameter).

Definition 4.1. The problem of matching the expressions of P on the graphs of the family $\mathcal{G}$ is *naughty* if there is an expression $\pi \in P$ and a subset of graphs $\mathcal{G}' = \{G_1, \ldots, G_n, \ldots\} \subseteq \mathcal{G}$ with $|G_n| \geqslant n$ such that the result produced by π on the graphs of $\mathcal{G}'$ has size $\Theta(2^{|G_n|})$.

Theorem 4.1. *Matching regular expressions with variables on DAGs is naughty.*

Proof. The theorem can be proven by showing that there is a regular expression ϕ and family of graphs such that the size of the results produced is exponential in the size of the input graph. Let $\phi = [x:1^*]$ and let G_n be the graph with n vertices with structure



where all the edges are labeled 1. There are $2^{\frac{n}{2}}$ maximal paths in the graph, and 2^{n-1} paths in total. All these paths match the expression and therefore must be returned as values for the variable x, so that the query will generate 2^{n-1} environments. □

This kind of complexity is unavoidable, since it is really in the graph, rather than in the expression: there are simply too many paths that can be returned. Even if the graphs are not as rich in paths, however, things are not necessarily better.

Definition 4.2. Let $\mathcal{G} = \{G_n \mid n \in \mathbb{N}\}$ be a family of graphs indexed by the parameter n. The family is *cold* if there exists a number $k \in \mathbb{N}$ such that the number of paths in a graph grows at most as $O(n^k)$.

Note that all families of trees are cold since in a tree two nodes determine at most one path, so that there are $O(n^2)$ possible paths.

Theorem 4.2. *Matching regular expressions with variables on cold graphs is naughty.*

Proof. I shall show again that there is a query that produces a result whose size is exponential in the number of vertices of the graph (and, consequently, given that the graph is cold, in the number of edges as well). Let $\phi = (1^*[x:1^*])^*$ and consider the family where G_n is the path

$$\pi^{(n)} = \pi_1 \xrightarrow{1} \pi_2 \xrightarrow{1} \cdot \qquad \cdots \xrightarrow{1} \pi_n \tag{26}$$

The family is clearly cold. The whole path satisfies the expression, but we are free to choose which parts will match the instances of 1^* without a variable and which will be assigned to the variable x. Since the whole expression is under the star, we can repeat this double assignment as many times as we wish. Consider an $n-1$ bit binary number, and set a bit to 1 if the corresponding edge is assigned to x, to 0 if it is not. For example:

$$\begin{array}{c}
\pi_1 \xrightarrow{1} \pi_2 \xrightarrow{1} \pi_3 \xrightarrow{1} \pi_4 \xrightarrow{1} \pi_5 \xrightarrow{1} \pi_6 \xrightarrow{1} \pi_7 \xrightarrow{1} \pi_8 \\
\underbrace{\qquad}_{1^*}\ \underbrace{\qquad\qquad\qquad}_{[x:1^*]}\ \underbrace{\qquad\qquad\qquad}_{1^*}\ \underbrace{\qquad\qquad\qquad}_{[x:1^*]} \\
\begin{array}{ccccccc} 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{array}
\end{array} \tag{27}$$

This assignment of sub-paths to the variable is therefore associated to the binary number $011011_2 = 51$. Every binary number has a different combination associated to it (as a matter of fact, there are more combinations than binary numbers), so there are at least 2^{n-1} possible results that must be produced. □

In the expression used in this theorem, the problem is caused by the presence of the variable x under the star: it is this variable that can be assigned in different combinations with the previous 1^* expression, and the possibility of replicating the combination makes the number of possible results unmanageably large.

Definition 4.3. An expression ϕ is *peachy* if, whenever it contains a sub-expression of the form ψ^*, ψ does not contain variables.

Definition 4.4. An expression ϕ is *hard* if it contains a sub-expression of one of the following two forms:

i) $(\zeta^*[x:\psi]\xi)^*$;
ii) $(\psi[x:\xi\zeta^*])^*$

with $L(\zeta) \cap L(\psi\xi) \neq \emptyset$.

The expression $(1^*[x:1^*])$ of the example above is hard, of type ii), with $\psi = 1^*$, $\xi = \epsilon$, $\zeta = 1$, and $L(\zeta) \cap L(\psi\zeta) = \{1\} \neq \emptyset$.

An expression that is not hard will be called *friendly*. All peachy expressions are friendly. In the following, I shall prove complexity results for friendly expressions so, from the point of view of the final results, the definition of peachy expressions is superfluous. They do have, however, a technical justification: we shall first prove that an efficient algorithm exists for peachy expressions and then use this result as a hypothesis in order to prove the more general result.

The following lemma, that I shall use in the following to simplify some proofs, restricts the class of graphs on which we have to go looking for exponential behavior:

Lemma 4.1. *Let $\mathcal{G}$ be a family of cold graphs and ϕ an expression; then ϕ is naughty on $\mathcal{G}$ if and only if there is a family $\mathcal{P}$ of paths such that ϕ is naughty on $\mathcal{P}$.*

Proof (Sketch). If ϕ is naughty on a family of paths, the theorem is obvious.

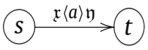
Suppose ϕ is not naughty on $\mathcal{G}$. Since each graph G_n of the family has a polynomial number of paths, if ϕ could be run on every path of the graph in polynomial space, it could be run in polynomial space (polynomial time) on the graph simply by running it on a path at the time. Therefore, there is in each graph of the family at least one path on which the expression is naughty. Taking all these paths we obtain the family $\mathcal{P}$. □

This theorem entails that, when showing that the algorithm runs in polynomial time, we can limit the proof to paths. So, from now on, we shall derive tractability results only for regular expressions with variables applied to paths. All these results will be extended, by virtue of this lemma, to cold graphs.

5. Finite state automata

Looking for paths that match an expression with variables is done through finite state automata, suitably modified. The use of finite state automata to match regular expressions is one of the classic topics in computing science, and there is a significant literature that analyzes the use of these automata on trees [18–21] and graphs [22–24]. I shall modify the basic automaton slightly in order to handle variables by *accumulating* vertices of the graph into their lists as the automaton proceeds to explore the graph. As usual, a finite state automaton is a 5-tuple $M = (Q, \Sigma, s_0, \delta, F)$, where Q is the set of states, Σ the input alphabet, s_0 the initial state, F the set of final states, and $\delta \subseteq Q \times \Sigma \times Q$ the (non-deterministic) state transition relation. If δ can be seen as a function $\delta : Q \times \Sigma \to Q$, the automaton is deterministic. Runs and accepting runs are defined in the usual way.

The states of the automaton are traversed during the exploration of the graph and, whenever a final state is reached, the automaton has traversed a path that matches the expression ϕ and, in so doing, it has also traversed the sub-paths that must be assigned to its variables. In order to build lists of vertices into the variables, we shall label the transitions of the automaton with *actions*. If $a \in \Sigma$ is a symbol whose presence causes a transition to be executed, then a general transition with actions will be indicated as



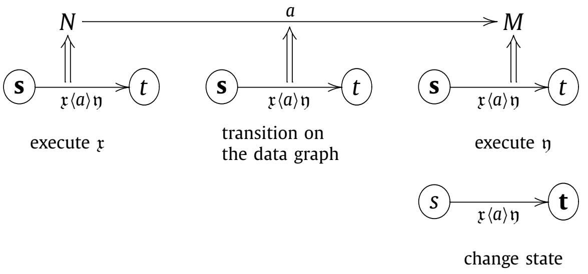
(28)

Either one or both the actions $\mathfrak{x}$ and $\mathfrak{y}$ can be missing, in which case the arc would be labeled $\langle a\rangle\mathfrak{y}$, $\mathfrak{x}\langle a\rangle$, or simply a, which corresponds to the action-less transition of the usual finite state automata. The symbols $\mathfrak{x}$ and $\mathfrak{y}$ may also represent collections of actions rather than a single action, in which case all the actions of the collection will be executed. The actions $\mathfrak{x}$ are executed before the automata visits the vertex on the data graph that matches the symbol a, while the actions $\mathfrak{y}$ are executed afterwards. Making reference to (28), we define two types of actions:

$\mathfrak{B}x$: begins a new list for the variable x, and adds to the list all vertices of the data graph that are crossed from that moment on; if $\mathfrak{B}x$ appears before the transition symbol a, then the data vertex on which the automaton was positioned before the transition is inserted into the list, otherwise it isn't;

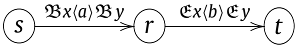
$\mathfrak{C}x$: closes the open list for the variable x; vertices of the data graphs crossed after this action is executed will not be added to the list of the variable; if the action is placed after the transition symbol a, the data vertex on which the automaton will be placed after executing the transition is inserted into the list, otherwise it isn't.

Conceptually, we can consider that the automaton moves from one vertex of the data graph to the next while "crossing" the symbol a that marks the transition. All the actions placed before the symbol a will consider only the data node on which the automaton is placed before the transaction, while all the actions placed after a will consider the automaton already placed at the following node. Schematically, we can describe the operation as in the following diagram, in which the boldface state symbol indicates the current state:


N
a
M
$\mathbf{s}$
$\mathfrak{x}\langle a\rangle\mathfrak{y}$
t
execute $\mathfrak{x}$
$\mathbf{s}$
$\mathfrak{x}\langle a\rangle\mathfrak{y}$
t
transition on the data graph
$\mathbf{s}$
$\mathfrak{x}\langle a\rangle\mathfrak{y}$
t
execute $\mathfrak{y}$
s
$\mathfrak{x}\langle a\rangle\mathfrak{y}$
$\mathbf{t}$
change state

(29)

Example 3. Suppose that we are in the state s of an automaton, and that so far we have built the environment u, not containing the variable x nor the variable y. Suppose the automaton has two transitions


s
$\mathfrak{B}x\langle a\rangle\mathfrak{B}y$
r
$\mathfrak{C}x\langle b\rangle\mathfrak{C}y$
t

(30)

and that it is positioned on the vertex N of a graph with an edge

$$N \xrightarrow{a} M \xrightarrow{b} Q \tag{31}$$

then the automaton will move from state s to state t, advance from the vertex N to the vertex Q, and the new environment will be

$$\big(u|x \mapsto [\![N, M]\!]|y \mapsto [\![M, Q]\!]\big) \tag{32}$$

Whenever there is an expression such as $[x : \phi]$, a transition is created with a $\mathfrak{B}x$ action entering the automaton that recognizes ϕ, and one with an action $\mathfrak{C}x$ that leaves its final states. Formally, an automaton with action is a 6-tuple $M = (Q, s_0, \delta, F, A, \alpha)$ where A is the set of possible actions, and $\alpha : Q \times \Sigma \times Q \to \{A\} \times \{A\}$ is the function that associates two sets of actions (those to be executed before the transition and those to be executed afterwards) to each transition of M. If ϕ is an expression, let M_ϕ be the automaton that recognizes it. If expressions are combined without the intervention of variables, the respective automata are combined in the customary way; if the combination of expression involves the definition of variables, the automata are combined as follows:

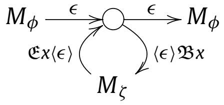
$\phi[x : \zeta]\psi$:

$$M_\phi \xrightarrow{\langle\epsilon\rangle\mathfrak{B}x} M_\zeta \xrightarrow{\mathfrak{C}x\langle\epsilon\rangle} M_\psi \tag{33}$$

$\phi([x:\zeta]+\zeta')\psi$:

$$\begin{tikzcd}
 & M_{\zeta'} \arrow[dr, "\epsilon"] & \\
M_\phi \arrow[ur, "\epsilon"] \arrow[dr, "\langle\epsilon\rangle\mathfrak{B}x"'] & & M_\psi \\
 & M_\zeta \arrow[ur, "\mathfrak{E}x\langle\epsilon\rangle"'] &
\end{tikzcd} \tag{34}$$

$\phi[x:\zeta]^*\psi$:

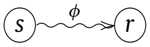


M_ϕ
ϵ
ϵ
M_ϕ
$\mathfrak{E}x\langle\epsilon\rangle$
$\langle\epsilon\rangle\mathfrak{B}x$
M_ζ

(35)

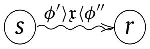
If one of the expressions is empty, the relative automaton will be the empty string recognizer $F = (\{q\}, q, \lambda x.\bot, \{q\})$.

As a shortcut, if an automaton recognizes the expression ϕ while going from a state s to a state t, I shall use the notation



(36)

If $\phi = \phi'\phi''$ and between ϕ' and ϕ'' the action $\mathfrak{x}$ is executed, I shall write



(37)

Note that these changes do not affect the structure of the automaton, and that on any given input path, the automaton with and without variable commands will go through exactly the same sequence of states, so that if we build an automaton for an expression ϕ we know that, apart from building variable values, the automaton will recognize it. The procedure that transforms a non-deterministic automaton into a deterministic one may exponentially increase the number of its states. This fact will have an impact on our complexity results, as I shall comment in the following section.

I call *action transitions* those transitions that contain one or more actions $\mathfrak{B}x$ or $\mathfrak{E}x$, where x is a variable. From now on, I shall also assume that all automata with n states have them numbered from 0 to $n-1$. Automata processing peachy expressions have a characteristic form, which will be crucial in proving their efficiency.

Definition 5.1. Let M be a finite state automaton. M is *action ordered* if its states are numbered in such a way that whenever there is an action transition from state l to state m, we have $l < m$.

Note that if ϕ does not contain variables, M_ϕ is trivially action ordered.

Theorem 5.1. *Let ϕ be a peachy expression, then M_ϕ can be action ordered.*

Proof. The proof is by induction on the number of variables in an expression. Let v be that number. If $v = 0$, the theorem is trivially true. Suppose now that the theorem is true for v variables. If the expression is peachy, there are only two ways in which the new variable may appear:

i) $\xi[x:\zeta]\psi$, which gives rise to the automaton (33). By the inductive hypothesis, M_ξ, M_ζ and M_ψ can be action ordered; numbering sequentially the states in M_ξ, M_ζ and M_ψ (in this order), the whole automaton will be action ordered.
ii) $\xi([x:\zeta]+\zeta')\psi$, which gives rise to the automaton (34). As before, all the individual automata can be action ordered. Numbering the states in the order M_ξ, M_ζ, $M_{\zeta'}$, M_ψ will action order the whole automaton. □

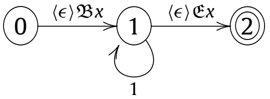
6. Elimination of ϵ-arcs

The general procedure that creates a recognition automaton for a given regular expression yields, in general, a sub-optimal result, that is, an automaton with a number of states greater than the minimum necessary for an automaton to recognize the same expression. The fundamental technique used in order to optimize automata without actions is the elimination of ϵ-arcs, and the consequent identification of states separated only by ϵ-arcs. The same general idea can be used here, with the caveat that not all ϵ-arcs with actions can be eliminated, as the next example will show.

Example 4. Consider the expression $\phi = [x:1^*]$, applied to the data graph

$$A \xrightarrow{1} B \xrightarrow{1} C \xrightarrow{1} D \tag{38}$$

starting at vertex A and considering, for the sake of simplicity, only the maximal match $[A, B, C, D]$. The finite state automaton with ϵ-transitions that recognizes the expression is

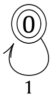

0
$\langle\epsilon\rangle\mathfrak{B}x$
1
$\langle\epsilon\rangle\mathfrak{E}x$
2
1

(39)

which, for the maximal path, creates the environment

$$\big([A, B, C, D], \big(x \mapsto [\![A, B, C, D]\!]\big)\big) \tag{40}$$

Suppose now we eliminate the ϵ-arcs. The resulting automaton without actions would be



(41)

Where do we place the actions relative to the variable x? The only place is the lone arc, and the only syntactically correct way in which they can be placed is to give it a label $\mathfrak{B}x\langle 1\rangle\mathfrak{E}x$. But it is easy to verify that on the maximal path of the data graph this automaton produces

$$\big([A, B, C, D], \big(x \mapsto \big[[A], [B], [C], [D]\big]\big)\big) \tag{42}$$

which doesn't respect the semantics that we have given to the expression. In fact, this is the result of the application to the same graph of the expression $\psi = ([x:1])^*$.

Before we start looking at techniques specific for automata with actions, we state the following property, whose proof is quite obvious:

> All ϵ-arcs without actions can be eliminated using the same rules as in automata without actions, and the resulting automata will be equivalent to the ones with ϵ-arcs even in the sense of the variable semantics.

Our reduction strategy will be of trying to remove actions from as many ϵ-arcs as possible, and then use the standard techniques in order to eliminate them.

Before introducing the two classes of actions that can be used, I shall introduce some notation that will avoid the inconvenience of having to draw too many graphs.

Let $\xi, \tau, \ldots$ be labels on arcs of the form $\mathfrak{x}\langle a\rangle\mathfrak{y}$. An arc between states s and r with label ξ will be indicated as $s(\xi)r$ or, explicitly, as $s(\mathfrak{x}\langle a\rangle\mathfrak{y})r$. A state s with n incoming arcs with actions ξ_i and m outgoing arcs with actions τ_i will be indicated as $(\xi_1, \ldots, \xi_n)s(\tau_1, \ldots, \tau_m)$.

Given two automata M and M', I write $M \equiv M'$ if M and M' accept the same expressions and give them the same variable semantics (in the sense of the semantics of Section 3). Given an automaton M and an operation ω, I indicate with $M[\omega]$ the automaton that results from the application of ω to M. We are interested in those operations ω such that, for any automaton M, it is the case that $M[\omega] \equiv M$.

We shall consider two classes of operations. The operations of the first class are called *in-arc displacement*, and their effect is to "move" an action $\mathfrak{x}$ on the "other side" of the transition condition of the arc. Given an arc α between states s and r, with label $\lambda(\alpha) = \mathfrak{x}\langle a\rangle\mathfrak{y}$, the *forward in-arc displacement* is represented as

$$\xrightarrow{\alpha} \equiv s\big(\mathfrak{x}\langle a\rangle\mathfrak{y}\big)r \mapsto s\big(\langle a\rangle\mathfrak{x}\mathfrak{y}\big)r \tag{43}$$

while the *backward in-arc displacement* is defined dually:

$$\xleftarrow{\alpha} \equiv s\big(\mathfrak{x}\langle a\rangle\mathfrak{y}\big)r \mapsto s\big(\mathfrak{x}\mathfrak{y}\langle a\rangle\big)r \tag{44}$$

Note that the two operations are not the inverse of each other. For example, if $\lambda(\alpha) = \mathfrak{x}\langle a\rangle\mathfrak{y}$, in $M[\xrightarrow{\alpha}][\xleftarrow{\alpha}]$ we have $\lambda(\alpha) = \mathfrak{x}\mathfrak{y}\langle a\rangle$. In general

$$M[\xrightarrow{\alpha}][\xleftarrow{\alpha}] \neq M[\xleftarrow{\alpha}][\xrightarrow{\alpha}] \neq M \tag{45}$$

However, it is immediate to prove the following property:

Theorem 6.1. *It is the case that*

$$M[\xrightarrow{\alpha}][\xleftarrow{\alpha}] \equiv M[\xleftarrow{\alpha}][\xrightarrow{\alpha}] \equiv M \tag{46}$$

if and only if $\exists\mathfrak{x}, \mathfrak{y}.(\lambda(\alpha) = \mathfrak{x}\langle\epsilon\rangle\mathfrak{y})$.

In-arc displacements are not terribly exciting, but they are useful to bring action across ϵ-arcs, to place the automaton in a condition in which we can apply the second class of operations: *cross-state displacements*. Cross-state displacements move actions from a transition that enters a state to a transition that leaves it (the *forward cross-state displacement*) or vice versa (the *backward cross-state displacement*) without crossing any transition condition.

The i, j forward cross-state displacement for state s, indicated as $\overset{i\rightarrow j}{s}$, takes the action that follows the transition symbol from the ith arc entering the state and places it before the transition condition on the jth arc leaving the state. That is, $\overset{i\rightarrow j}{s}$ transforms

$$\big(\langle a_1\rangle\mathfrak{x}_1,\ldots,\langle a_i\rangle\mathfrak{x}_i,\ldots,\langle a_n\rangle\mathfrak{x}_n\big)s\big(\mathfrak{y}_1\langle b_1\rangle,\ldots,\mathfrak{y}_j\langle b_j\rangle,\ldots,\langle\mathfrak{y}_m\rangle b_m\big) \tag{47}$$

into

$$\big(\langle a_1\rangle\mathfrak{x}_1,\ldots,\langle a_i\rangle,\ldots,\langle a_n\rangle\mathfrak{x}_n\big)s\big(\mathfrak{y}_1\langle b_1\rangle,\ldots,\mathfrak{x}_i\mathfrak{y}_j\langle b_j\rangle,\ldots,\langle\mathfrak{y}_m\rangle b_m\big) \tag{48}$$

The backward cross-state displacement, $\overset{i\leftarrow j}{s}$ is defined similarly. If $j=0$ in the forward operator (resp. $i=0$ in the backward operator), the action is removed from the incoming (resp. outgoing) arc and eliminated. The composition of operators is defined in the obvious way.

Definition 6.1. Given the state transition

$$\big(\langle a_1\rangle\mathfrak{x}_1,\ldots,\langle a_n\rangle\mathfrak{x}_n\big)s\big(\mathfrak{y}_1\langle b_1\rangle,\ldots,\mathfrak{y}_m\langle b_m\rangle\big) \tag{49}$$

a k-complete forward cross-state displacement is an operation of the form

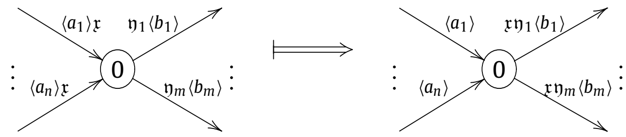
$$\overset{k\rightarrow *}{s} = \Big(\bigcirc_{j=1}^{m}\overset{k\rightarrow j}{s}\Big)\circ\Big(\bigcirc_{i\neq k}\overset{i\rightarrow 0}{s}\Big) \tag{50}$$

That is, a k-complete operation moves the action $\mathfrak{x}_k$ to all the output arcs and eliminates all of the input actions. A k-complete backward operation is defined analogously.

Definition 6.2. A forward cross-state displacement is *complete*, indicated with $\vec{s}$, if there is a k such that the operation is k-complete and $\mathfrak{x}_1=\mathfrak{x}_2=\cdots=\mathfrak{x}_n$.

It is immediate to see that if a k-complete operator is complete, then all k-complete operators are, independently of k, and they are all equal.

Graphically, a complete operation has the form


$\langle a_1\rangle\mathfrak{x}$
$\mathfrak{y}_1\langle b_1\rangle$
0
$\langle a_n\rangle\mathfrak{x}$
$\mathfrak{y}_m\langle b_m\rangle$
$\Longrightarrow$
$\langle a_1\rangle$
$\mathfrak{x}\mathfrak{y}_1\langle b_1\rangle$
0
$\langle a_n\rangle$
$\mathfrak{x}\mathfrak{y}_m\langle b_m\rangle$

(51)

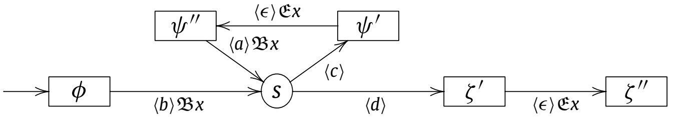
The backward operation is defined analogously with the caveat that, this time, the equality is required to hold for the actions on the outgoing arcs.

The following theorem characterizes the semantic-preserving cross-state operations:

Theorem 6.2. *Let ω be a cross-state displacement operation, then $M[\omega]\equiv M$ if and only if ω is complete.*

Proof (Sketch). The complete proof of the theorem is a long and rather tedious enumeration of possible cases. Here I shall give the gist of it by considering only a forward cross-state displacement with $\mathfrak{x}=\mathfrak{B}x$. For the sake of simplicity, I assume that $n=m=2$. This can be done without loss of generality as, if n and m are greater, the theorem holds if and only if it holds for each combination of two incoming and two outgoing arcs.

Assume first that we alter the automaton M to $M[\vec{s}]$, where $\vec{s}$ is complete. Since there is an action $\mathfrak{B}x$ upon entering the state s, there must be a corresponding $\mathfrak{E}x$ somewhere on all the paths from s to any final state and from s to itself. The most complete configuration of the automaton M is then

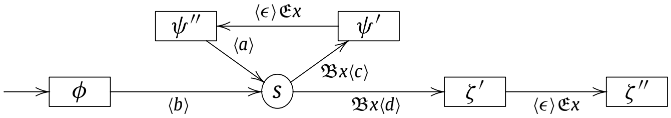

ψ''
$\langle\epsilon\rangle\mathfrak{E}x$
ψ'
$\langle a\rangle\mathfrak{B}x$
$\langle c\rangle$
ϕ
$\langle b\rangle\mathfrak{B}x$
s
$\langle d\rangle$
ζ'
$\langle\epsilon\rangle\mathfrak{E}x$
ζ''

(52)

and the expression that it recognizes is

$$\phi\, b \left(\left[x : c\psi' \right] \psi''\, a \right)^{*} \left[x : d\, \zeta' \right] \zeta'' \tag{53}$$

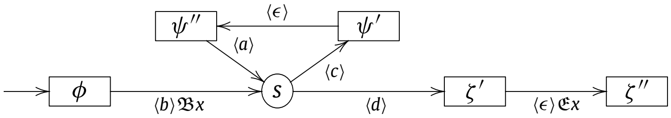
The automaton $M[\overset{1\rightarrow *}{s}] = M[\overset{2\rightarrow *}{s}] = M[\vec{s}]$ is


ψ''
$\langle\epsilon\rangle\,\mathfrak{E}x$
ψ'
$\langle a\rangle$
$\mathfrak{B}x\langle c\rangle$
ϕ
s
ζ'
ζ''
$\langle b\rangle$
$\mathfrak{B}x\langle d\rangle$
$\langle\epsilon\rangle\,\mathfrak{E}x$

(54)

which, as it is easy to verify, recognizes the same expression, and gives it the same variable semantics.

Consider now the case in which there is only one $\mathfrak{B}x$ entering the state s. The only case in which this can happen in a syntactically valid automaton is that in which the action is placed on the arc labeled b, and there is a corresponding $\mathfrak{E}x$ between ζ' and ζ''.

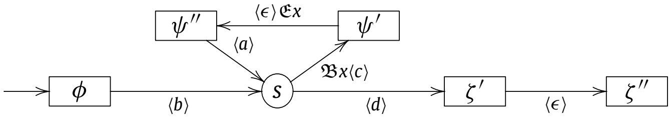


(55)

This automaton recognizes the expression

$$\phi\, b \left[x : \left(c\, \psi' \psi''\, a \right)^{*} d\, \zeta' \right] \zeta'' \tag{56}$$

There are three possible moves that will displace the action $\mathfrak{B}x$: $M[\overset{2\rightarrow 1}{s}]$, $M[\overset{2\rightarrow 2}{s}]$, and $M[\overset{2\rightarrow *}{s}]$. In the first case, in order to obtain a valid automaton, some subsequent operation would have to place an $\mathfrak{E}x$ between ϕ' and ϕ'' (note that ψ' and/or ψ'' may be empty):



(57)

This automaton recognizes

$$\phi\, b \left(\left[x : c\, \psi' \right] \psi''\, a \right)^{*} d\, \zeta'\, \zeta'' \tag{58}$$

which, as can easily be verified, has a different variable semantics than (56). Similarly, $M[\overset{2\rightarrow 2}{s}]$ recognizes

$$\phi\, b \left(c\, \psi'\, \psi'' \right)^{*} \left[x : d\, \zeta' \right] \zeta'' \tag{59}$$

and $M[\overset{2\rightarrow *}{s}]$ recognizes

$$\phi\, b \left(\left[x : c\, \psi' \right] \psi''\, a \right)^{*} \left[x : d\, \zeta' \right] \zeta'' \tag{60}$$

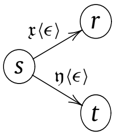
both of which have a different variable semantics than (56). □

In the proof of this theorem, it is worth noting that the inputs accepted by the automata are always the same, and correspond to the inputs accepted by the regular expression without variables

$$\phi\, b \left(c\, \psi'\, \psi''\, a \right)^{*} d\, \zeta'\, \zeta'' \tag{61}$$

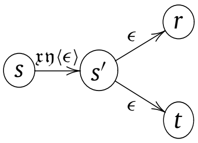
What does change is the variable semantics of the automaton, that is, the way in which values are assigned to the variable x.

Finally, we can modify the automaton so that there is at most one ϵ-arc leaving each state. If there are, for example, two:


r
$\mathfrak{x}\langle\epsilon\rangle$
s
$\mathfrak{y}\langle\epsilon\rangle$
t

(62)

we can transform it into



(63)

and then we can remove the ϵ-transitions without actions using the standard techniques. Note that the automaton that we obtain is not significantly better than the one with multiple ϵ-transitions. I introduced this operation essentially because it will be easier to prove the complexity results on automata that have this form.

7. The intersection graph

One well-known way of executing finite state automata on graphs is by reducing the problem to a traversal of the *intersection graph* [15].

Definition 7.1. Let $G = (V, E, \Sigma, \lambda)$ be an edge-labeled directed acyclic graph, and $M = (Q, s_0, \delta, F, A, \alpha)$ an automaton with actions. The *intersection graph* $G \times M = (V', E', \mu)$ is a graph with $V' = V \times Q$, $E' \subseteq V' \times V'$ defined as follows: $((v,q),(v',q')) \in E'$ if either

$$\bigl(v, v'\bigr) \in E \quad \text{and} \quad \lambda\bigl(v, v'\bigr) = a \quad \text{and} \quad q' \in \delta(q, a) \tag{64}$$

or

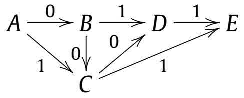
$$v = v' \quad \text{and} \quad q' \in \delta(q, \epsilon) \tag{65}$$

and $\mu : E' \to A$ with $\mu((v,q),(v',q')) = w$ if $\lambda(v,v') = a$, and $\alpha(q,a,q') = w$.

A vertex (v, s_0) is called a *source* of the graph, while a vertex (v, q) with $q \in F$ is called a *sink*.

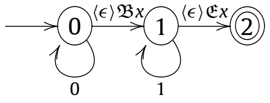
Note that, because of the way the graph is constructed, it is more akin to a Cartesian product of the data graph and the automaton than to an intersection. Nevertheless, the name *intersection graph* is pretty much a standard, and I shall retain it. If ϕ is an expression, and the automaton used to recognize it is clear from the context, I shall informally use the notation $G \times \phi$.

Example 5. Consider the expression $0^*[x : 1^*]$, applied to the graph



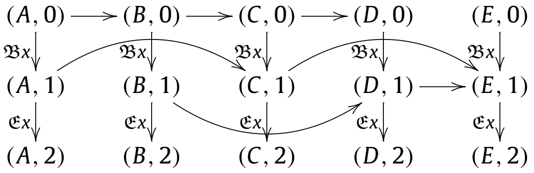
(66)

The expression is processed by the automaton



(67)

and the intersection graph is



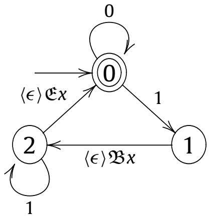
(68)

The states $(A, 0)$ through $(E, 0)$ are sources, while the states $(A, 2)$ through $(E, 2)$ are sinks.

Example 6. Consider the expression $(0^*1[x : 1^*])^*$ applied to the path

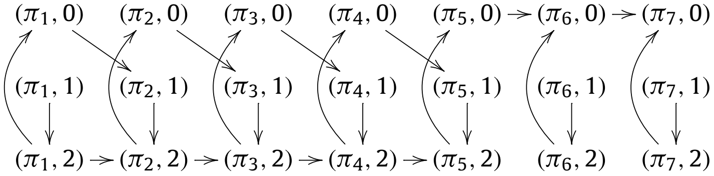
$$\pi_1 \xrightarrow{1} \pi_2 \xrightarrow{1} \pi_3 \xrightarrow{1} \pi_4 \xrightarrow{1} \pi_5 \xrightarrow{0} \pi_6 \xrightarrow{0} \pi_7 \tag{69}$$

The automaton with actions that recognizes the language is



(70)

giving the intersection graph



(71)

(for the sake of simplicity and clarity of the diagram, actions are not represented here).

We are now dealing with two graphs: the data graph, on which we are matching the regular expression, and the intersection graph, that we built by combining the data graph with the automaton. To avoid confusion, we shall use the prefix "d-" when referring to the data graph, and the prefix "i-" when referring to the intersection graph. So we will talk about d-vertices, i-vertices, d-edges, i-edges, etc.

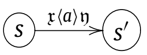
7.1. Matching in the intersection graph

It is a known result that matching a regular expression on a graph is equivalent to finding a path from a source to a sink in the intersection graph [15]. The function *visit*, that we shall consider shortly, traverses all the paths in the intersection graph (the *i-paths*) and, for each i-edge traversed will call the function *trans*, which we shall also consider in a moment. In order to present the algorithm, I need a bit of notation. Let $\left[\begin{smallmatrix} s \\ A \end{smallmatrix}\right] = (A, s)$ be a vertex of the intersection graph, where A is a d-vertex, and s a state of the automaton. Define the canonical projections as:

$$\mathtt{fst}\left(\begin{bmatrix} s \\ A \end{bmatrix}\right) = A$$

$$\mathtt{snd}\left(\begin{bmatrix} s \\ A \end{bmatrix}\right) = s$$

Given an i-edge $e = (\left[\begin{smallmatrix} s \\ A \end{smallmatrix}\right], \left[\begin{smallmatrix} s' \\ A' \end{smallmatrix}\right])$, let left$(e)$ be the set of actions placed to the left of the transition symbol in the transition $(\mathrm{snd}(\left[\begin{smallmatrix} s \\ A \end{smallmatrix}\right]), \mathrm{snd}(\left[\begin{smallmatrix} s' \\ A' \end{smallmatrix}\right]))$, and let right$(e)$ be the actions placed on the right. That is, if the automaton has a transition



(72)

then

$$\mathtt{left}(e) = \mathfrak{x}$$

$$\mathtt{right}(e) = \mathfrak{y}$$

Each i-vertex $[v]$ has associated with it a state state$[\![v]\!] \in \{P, B\}$ and an adjacency list ady$[\![v]\!]$ containing all the i-vertices reachable from $[v]$ in one step. The state of a node is P (Pending) if the node can be visited, and B (Busy) if the algorithm can't visit it at the moment.

In addition to these quantities and functions related to the intersection graph, the algorithm will manage the following variables:

π: a path that matches the expression; the algorithm will build a collection of these paths as it traverses the graph;
O: the set of "open" variables, that is, the set of variables for which, during the exploration of the current path, the algorithm has encountered an action $\mathfrak{B}x$ not yet matched by a corresponding $\mathfrak{E}x$;
u: the environment with the variable bindings for the path that is being explored; this environment is built progressively while traversing a path, until a final state of the automaton is reached; at that point the environment is complete, and it is inserted in
R: the set of environments and the related paths that correspond to paths already fully traversed; this will be the final result of the algorithm.

The function *trans* is called each time an i-edge is traversed. It receives as parameters the four quantities defined above. Moreover, if *trans* is called while traversing the edge $e = (\left[\begin{smallmatrix} s \\ A \end{smallmatrix}\right], \left[\begin{smallmatrix} s' \\ A' \end{smallmatrix}\right])$, it receives the following parameters as well:

w: the d-vertex corresponding to the i-vertex in which the edge originates, that is: $w = \mathrm{fst}(\left[\begin{smallmatrix} s \\ A \end{smallmatrix}\right]) = A$;
s: state of the automaton where we end up after traversing the edge, that is: $\mathrm{snd}(\left[\begin{smallmatrix} s' \\ A' \end{smallmatrix}\right])$;

```
trans($\mathfrak{x}$, $\mathfrak{y}$, s, w, $\pi$, u, R, O)
if $\exists x.(\mathfrak{x} = \mathfrak{B}x)$ then
  start(u, x);
  O ← O$\cup\{x\}$;
else if $\exists x.(\mathfrak{x} = \mathfrak{E}x)$ then
  O ← O$-\{x\}$;
fi
$\pi$ ← $\pi$@[w];
foreach x in O do
  last(u(x)) ← last(u(x))@[w];
od
if $\exists y.(\mathfrak{y} = \mathfrak{B}y)$ then
  start(u, y);
  O ← O$\cup\{y\}$;
else if $\exists y.(\mathfrak{y} = \mathfrak{E}y)$ then
  O ← O$-\{y\}$;
fi
if s in F then
  R ← R$\cup$\{ ($\pi$, u) \};
fi
return [$\pi, u, R, O$];
```

Fig. 1. Transition function of the automaton with actions.

$\mathfrak{x}$: actions to be executed before traversing the i-edge: $\mathfrak{x} = \text{left}(e)$;
$\mathfrak{y}$: actions to be executed after traversing the i-edge: $\mathfrak{x} = \text{right}(e)$.

The function (shown in Fig. 1) returns a 4-tuple with the values of π, u, R, and O resulting after doing the traversal.

Here, @ is the list append operation, *last* is a function that returns the last element as a list, and *start* is the function that starts a new variable or, if the variable already exists, a new list for it:

```
start(u, x)
if x in D(u) then
  u(x) ← u(x)@[];
else
  u ← (u | x ↦ [])
fi
```

The function *visit* is the one that explores the intersection graph, is called with the initial i-vertex, empty path and empty environments and sets, and is a simple modification of depth-first search:

```
visit(v, $\pi$, $u$, $R$, $O$)
1.   state[v] ← B;
2.   foreach w in ady[v] do
3.     if state[w] = P then
4.       ($\pi'$, $u'$, $R'$, $O'$) ← trans(left(v,w), right(v,w), snd(w), fst(v), $\pi$, $u$, $R$, $O$);
5.       $R$ ← $R\cup$visit(w, $\pi'$, $u'$, $R'$, $O'$);
6.     fi
7.   od
8.   state[v] ← P;
9.   return $R$;
```

The main difference with depth-first search is line 8: once we have visited all the paths that start from a vertex v, we set v back to the non-visited state so that, if we come back to v through a different route, all the paths that originate with it will be visited again. It is quite easy to see that with this change the algorithm will traverse all the paths in the graph, and I shall omit the proof:

Lemma 7.1. *If $\tilde{\Pi}$ is a path in a graph originating from a vertex x, and the function* visit *is first called with source x, then there is a recursive invocation of visit such that, when the function trans of line* 4 *returns, it is with $\pi' = \tilde{\Pi}$.*

What is new in this case, and what we do need to prove, is the correctness of the algorithm with respect to the environments and the variables that they define. That is, we have to show that, while the function *visit* traverses a given i-path, the function *trans* generates the correct variable bindings.

The basic results are summed up by two lemmas.

Lemma 7.2. *Let π be a d-path constructed by the function* visit *during the traversal of an intersection graph $G \times M_\phi$; then* visit *calls*

$$\mathrm{trans}(\mathfrak{x}, \mathfrak{y}, s, w, \pi, u, R, O)$$

with $s \in F$ (remember that F is the set of final states of the automaton M_ϕ) if and only if $(G, \pi) \models \phi$.

The lemma is an immediate consequence of the correctness of the traversal algorithm for intersection graphs (see [15]).

Lemma 7.3. *Let π be a d-path constructed by the function* visit *during the traversal of an intersection graph $G \times M_\phi$, and let there be a call*

$$\mathrm{trans}(\mathfrak{x}, \mathfrak{y}, s, w, \pi, u, R, O)$$

with $s \in F$. Then the environment u that is added to R in line 18 *is the* (*only*) *environment such that $(\pi, u) \in [\![\phi]\!](G)$.*

Proof. The path π that is inserted in R (line 18) has been built by concatenating all the vertices that the automaton has crossed (line 7) so, by Lemma 7.2, we have $(G, \pi) \models \phi$ and, by Theorem 3.1, there is an environment u such that $(\pi, u) \in [\![\phi]\!](G)$.

To see that the environment u built during the execution is just the one that satisfies the semantics of the expression, I shall proceed by induction on the number of variables in the expression.

If there are no variables, then $u = \bot$, and the theorem is trivially true.

Assume now that the result is true for any formula with at most k variables and consider a formula with $k+1$ variables. We have to distinguish three cases, depending on how the $(k+1)$th variable appears.

Consider first the case in which the formula is of the type $\xi[x:\zeta]\psi$, where ξ, ζ, and ψ contain at most k variables. This expression is realized by the automaton

$$M_\phi \equiv M_\xi \xrightarrow{\langle\epsilon\rangle\mathfrak{B}x} M_\zeta \xrightarrow{\langle\epsilon\rangle\mathfrak{E}x} M_\psi \tag{73}$$

Let the path that matches the expression be

$$\pi = [v_1, \ldots, v_p, \ldots, v_h, \ldots, v_n] \tag{74}$$

where v_p is the graph vertex on which the automaton executes the action $\mathfrak{B}x$, and v_h the vertex on which it executes the action $\mathfrak{E}x$. In v_p the automaton leaves M_ξ, so it must be in one of its final states, which implies $(G, \pi^{|p}) \models \xi$. By the same token, in v_p M_ϕ is on the initial state of M_ζ and in v_h in one of its final states, so $(G, \pi^{p|h}) \models \zeta$. Similarly, $(G, \pi^{h|}) \models \psi$. We have thus proved the first condition for paths according to the semantics, since $\pi = \pi^{|p} \bowtie \pi^{p|h} \bowtie \pi^{h|}$.

During the execution of M_ξ, $x \notin V$. By the inductive hypothesis, at the point in which we analyze v_p, the automaton has assembled an environment u^ξ such that $(\pi^{|p}, u^\xi) \in [\![\phi]\!](G)$. If we run M_ζ on $u^{p|h}$, by the inductive hypothesis, the automaton will create an environment u^ζ such that $(\pi^{p|h}, u^\zeta) \in [\![\zeta]\!](G)$. The variable x is placed in O when M_ϕ is analyzing u^p (step 3), and removed when M_ϕ is analyzing u^h (step 5) so, it is assigned the value $\pi^{p|h}$. Therefore, when entering M_ψ, the accumulated path is $\pi^{|h}$ and the environment is

$$\tilde{u} = \left(u^\xi | u^\zeta | x \mapsto \pi^{p|h}\right) \tag{75}$$

which is precisely the environment such that $(\pi^{|h}, \tilde{u}) \in [\![\phi[x:\zeta]]\!](G)$.

Similarly, at the end of M_ψ, the automaton will have created the environment

$$\hat{u} = \left(u^\xi | u^\zeta | x \mapsto \pi^{k|h} | u^\psi\right) \tag{76}$$

which is the one such that $(\pi, \hat{u}) \in [\![\phi[x:\zeta]\psi]\!](G)$.

Consider now the expression $\phi([x:\zeta] + \zeta')\psi$, which is realized by the automaton (34). In this case, if the execution goes through ζ', the pair (π, u) is correct by the induction hypothesis (there are at most k variables in that path), while if it goes through ζ, we can apply the same reasoning as in the previous case.

Finally, for the expression $\phi([x:\zeta])^*\psi$, every execution consists of a number r of repetitions of the machine M_ζ, that is, it is equivalent to an execution for the expression

$$\phi\,\overbrace{[x:\zeta]\cdots[x:\zeta]}^{r}\,\psi$$

which can also be analyzed using the same technique, with the difference that, in this case, the variable x is created anew only once by the function *start*: in all other cases a new list will be added to it so that in the end the value of x will be composed of r disjoint paths. □

Theorem 7.1. *Executing the algorithm* visit *in an intersection graph $G \times M_\phi$ starting from the source of the graph, the algorithm returns $[\![\phi]\!](G)$.*

Proof. By Lemma 7.1, the algorithm tests all the graphs in the intersection graph, and therefore builds all paths π compatible with the state transitions of the automaton. By Lemma 7.2, for each path π in the graph such that $(G, \pi) \models \phi$, the function *trans* will be called with a final state, so in line 18 it will add a pair (π, u) to the result set. By Lemma 7.3, this is a pair $(\pi, u) \in [\![\phi]\!](G)$. This proves that $R \subseteq [\![\phi]\!](G)$.

On the other hand, if $(\pi, u) \in [\![\phi]\!](G)$, then π is a d-path which is the projection on the first element of an i-path that ends in an i-vertex $\begin{bmatrix} s \\ N \end{bmatrix}$ with $s \in F$. By Lemma 7.2, this means that the function *trans* will be called with the path π and the state s, so that, by Lemma 7.3, at line 18 (π, u) will be placed into R. Therefore $[\![\phi]\!](G) \subseteq R$. □

Materializing the intersection graph is a significant burden and, in practice, a useless one, since it can be avoided through a simple change to the traversal algorithm. I am giving the algorithm in this form for ease of exposition, and because the results that follow are easier to prove if one considers the algorithm that works on materialized graphs.

7.2. Some properties of the intersection graph

From now on, I shall consider intersection graphs of the form $\Pi \times M_\phi$, where Π is a path, and M_ϕ is an automaton reduced as per the method of Section 6. A vertex of the graph is of the form (π_i, s_k) which, for the sake of conciseness, I shall often indicate simply as $\begin{bmatrix} k \\ i \end{bmatrix}$. Given a vertex $\begin{bmatrix} k \\ i \end{bmatrix}$ of $\Pi \times \phi$, there can be two types of outgoing edges. If the data graph has, between the vertices π_i and π_{i+1}, an edge labeled a, and if the state s_k has an a-transition to a state s_h, then the intersection graph will have an edge $\left[{}_i{}^{k}\;\right] \rightarrow \begin{bmatrix} h \\ i+1 \end{bmatrix}$. I call these: *cross edges*. If there is an ϵ-transition from state s_k to state s_l, then the intersection graph will have an edge $\begin{bmatrix} k \\ i \end{bmatrix} \rightarrow \begin{bmatrix} l \\ i \end{bmatrix}$. I call these: *vertical edges*. All these edges will contain one or more actions. Note that, since we assume that the automaton has been reduced as per Section 6, each vertex of the intersection graph will have at most two outgoing edges: a vertical one and a cross one.

Lemma 7.4. *Let $\Gamma = \Pi \times \phi$, where Π is a path. Then each vertex $\begin{bmatrix} l \\ i \end{bmatrix}$ of Γ has at most two outgoing edges: a cross edge $\begin{bmatrix} l \\ i \end{bmatrix} \rightarrow \begin{bmatrix} u \\ i+1 \end{bmatrix}$ and a vertical edge $\begin{bmatrix} l \\ i \end{bmatrix} \rightarrow \begin{bmatrix} v \\ i \end{bmatrix}$.*

The traversal of the graph will begin at the *source* vertex $\begin{bmatrix} 0 \\ 1 \end{bmatrix} = (\pi_1, s_0)$, and each path that leads to a state $\begin{bmatrix} f \\ k \end{bmatrix}$ with $s_f \in F$ will mean that the path $[\pi_1, \ldots, \pi_k]$ has been recognized. The set of *final* vertices is $\mathcal{F} = \{\begin{bmatrix} q \\ k \end{bmatrix} \mid s_q \in F\}$.

Sometimes, rather than working directly with the graph, we shall work with a tree generated by *unfolding* it; unfolding is defined for general graphs with sources and destination. The definition I give here is the same as given in [1]:

Definition 7.2. Let $G = (V, E, S, F)$ be a graph, where V is the set of its vertices, $E \subseteq V \times V$ the set of its edges, $S \subseteq V$ the set of its sources, and $F \subseteq V$ the set of its destinations. The *unfolding* of G, $\mathcal{U}(G)$ is the graph

$$\mathcal{U}(G) = \left(V', E', S', F'\right) \tag{77}$$

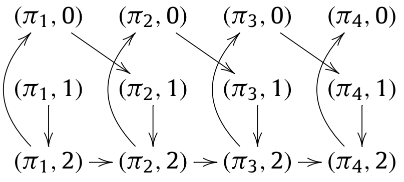
such that:

$$\begin{aligned}
&V' = \{p \mid p = [v_0, \ldots, v_n] \text{ is a path of } G, \text{ and } v_0 \in S\} \\
&E' = \{(p, q) \mid \exists v \in V.(q = p@[v])\} \\
&S' = \{p \mid \exists v.(p = [v] \wedge v \in S)\} \\
&F' = \{p \mid p = [v_0, \ldots, v_n] \wedge v_n \in F\}
\end{aligned} \tag{78}$$

It is easy to show that $\mathcal{U}(G)$ is a forest of trees whose roots are the paths of the form $[v]$, with $v \in S$, and that the tree is finite if and only if G has no cycles.

In our case, since there is only one source, and the intersection graphs have no ϵ-cycles, $\mathcal{U}(\Pi \times \phi)$ is always a single, finite tree.

Example 7. Consider the first portion of the graph of the previous example



(79)

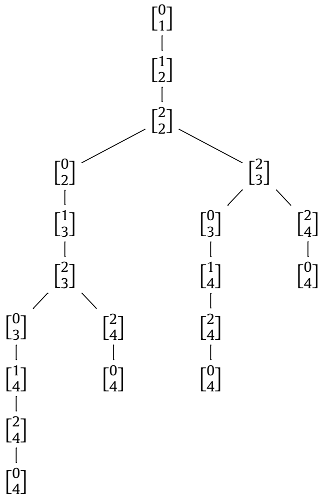

$\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}1\\2\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}2\\2\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}0\\2\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}2\\3\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}1\\3\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}0\\3\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}2\\4\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}2\\3\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}1\\4\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}0\\4\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}0\\3\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}2\\4\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}2\\4\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}1\\4\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}0\\4\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}0\\4\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}2\\4\end{smallmatrix}\right]$
$\left[\begin{smallmatrix}0\\4\end{smallmatrix}\right]$

Fig. 2. Unfolding of the graph in Example 7.

The source is in this case the vertex $\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right] = (\pi_1, 0)$, and the unfolding of the graph is as shown in Fig. 2.

The problem of processing regular expressions with variables can be reduced to finding all the paths in the intersection graph (viz. all the i-paths, according to our nomenclature).

The problem is tractable (non-naughty) if and only if the intersection graph has a number of i-paths that varies polynomially with the length of the d-path. Given that each vertex of the unfolding corresponds to an i-path from the source, the following property follows immediately:

Lemma 7.5. *The expression ϕ is naughty if and only if there is a family of paths $\pi^{(n)}$ with $|\pi^{(n)}| = n$ such that, denoting by $|G|$ the number of vertices in a graph G, we have*

$$\left|\mathcal{U}\left(\pi^{(n)} \times \phi\right)\right| = \Theta\left(2^n\right) \tag{80}$$

8. Complexity results

As was mentioned in the introduction, the presence of variables will not make the matching problem any easier. Theorem 3.1 is a statement of this fact: any answer to a matching problem with variables entails an answer to the corresponding matching problem without variables. This in turn implies that any intractable class of regular expressions without variables will continue to be so when we add variables. Since we are interested in the added complexity due to variables, I shall only consider expressions, that, were it not for the presence of variables, would be tractable.

8.1. Peachy expressions are not naughty

We shall first consider *peachy* expressions, that is, expressions in which, whenever there is a sub-expression of the form ϕ^*, ϕ does not contain any variable. To these expressions we can apply the equivalence

$$\xi\left([x:\zeta] + \zeta'\right)\psi = \xi[x:\zeta]\psi + \xi\zeta'\psi \tag{81}$$

to bring all the "+" operators at the top level of the expression, obtaining the form

$$\phi = \phi_1 + \phi_2 + \cdots + \phi_n \tag{82}$$

where the operator "+" doesn't appear in any of the ϕs. I shall call *conjunctive* a regular expression in which no + appears. Not that this transformation could lead to an exponential explosion of the size of the expressions, since the number of conjunctive expression could be exponential in the size of the original expression ϕ. I do not consider this kind of complexity in this section, in which I shall work with conjunctive expressions only. The results of the next section, being independent of the conjunctive form of the expression, will apply in general.

In the rest of this section, I shall assume that all automata are action ordered, that their states are numbered from $0, \ldots, n-1$, and that the vertices of a path with m vertices are numbered from $0, \ldots, m-1$.

Because of Lemma 7.4, each vertex $\left[\begin{smallmatrix}j\\i\end{smallmatrix}\right]$ of the intersection graph has at most two outgoing edges: a cross one and a vertical one. Moreover, since the automaton is action ordered, for each vertical edge $\left[\begin{smallmatrix}j\\i\end{smallmatrix}\right] \to \left[\begin{smallmatrix}h\\i\end{smallmatrix}\right]$, we have $h > j$.

The vertical edges $\left[\begin{smallmatrix}j\\i\end{smallmatrix}\right] \to \left[\begin{smallmatrix}h\\i\end{smallmatrix}\right]$ exist if there is an ϵ-transition from the state j to the state h of the automaton. This transition can always be crossed whenever the automaton is in state j, independently of the vertex of the graph on which

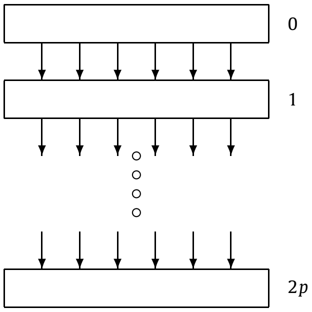


Fig. 3. The structure of the intersection graphs.

the automaton is positioned. Therefore, if there is an arc $\genfrac{[}{]}{0pt}{}{j}{i} \to \genfrac{[}{]}{0pt}{}{h}{i}$ for some i, there is one for *all* vertices of the path. If there are p variables, there are $2p$ such groups of vertical edges. We can therefore structure the graph as a stack of $2p+1$ subgraphs, which I shall call the *plateaus*, connected by the groups of down transitions, as in Fig. 3. There can be other connections between the plateaus, but the following lemma limits their form:

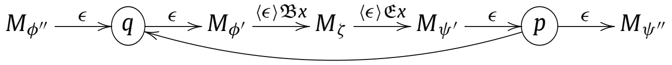
Lemma 8.1. *Suppose the intersection graph has an edge $\genfrac{[}{]}{0pt}{}{j}{i} \to \genfrac{[}{]}{0pt}{}{h}{i}$, then there are no edges $\genfrac{[}{]}{0pt}{}{p}{r} \to \genfrac{[}{]}{0pt}{}{q}{r+1}$ with $q \leqslant j$ and $p \geqslant h$, for any r.*

Proof. Suppose the edge exists. The edge $\genfrac{[}{]}{0pt}{}{j}{i} \to \genfrac{[}{]}{0pt}{}{h}{i}$ is an ϵ-transition with an action $\mathfrak{B}x$ or $\mathfrak{C}x$. Suppose that the action is $\mathfrak{B}x$. Then, without the edge $\genfrac{[}{]}{0pt}{}{p}{r} \to \genfrac{[}{]}{0pt}{}{q}{r+1}$, the automaton that recognizes the expression analyzed by the graph is of the form

$$M_\phi \xrightarrow{\langle\epsilon\rangle\mathfrak{B}x} M_\zeta \xrightarrow{\langle\epsilon\rangle\mathfrak{C}x} M_\psi \tag{83}$$

with state j being one of the final states of M_ϕ and h being the initial state of M_ζ. Consider now the edge $\genfrac{[}{]}{0pt}{}{p}{r} \to \genfrac{[}{]}{0pt}{}{q}{r+1}$, which implies a possible transition from p to q. Because $q \leqslant j$ and $p \geqslant h$, state q is in M_ϕ and state p is either in M_ζ or in M_ψ.

Given that the arcs $\mathfrak{B}x$ and $\mathfrak{C}x$ are inserted in pairs, it is easy to see that there can be no arc from M_ζ to M_ϕ, so state p must be in M_ψ. Insert ϵ-transitions to isolate states q and p in M_ϕ and M_ψ. This gives the automaton:



(84)

which recognizes the expression $\phi''(\phi'[x:\zeta]\psi')^*\psi''$, contradicting the hypothesis that the expression is peachy.

The case in which the action is $\mathfrak{C}x$ is analogous. □

That the algorithm has a polynomial time execution on paths with peachy expressions is a consequence of the following lemma:

Lemma 8.2. *The intersection of a path with n vertices and a peachy formula with p variables has at most $O(n^{2p+2})$ paths.*

Proof. Consider two given vertices of the graph and start creating a path between them. While building this path, arriving at the vertex $\genfrac{[}{]}{0pt}{}{j}{i}$, by virtue of Lemma 7.4, we have at most two choices: we can go either to $\genfrac{[}{]}{0pt}{}{k}{i+1}$ or to $\genfrac{[}{]}{0pt}{}{h}{i}$. Consider the worst case: at *each* vertex we can make two choices. If we make the *vertical* choice, we pass from one plateau to another with a higher plateau number. Because of Lemma 8.1, once we enter, say, plateau k, we can never go back to one of the first $k-1$ plateaus, that is, at every vertex, we can either stay in the same plateau or go to a lower plateau. Thus, we can choose *vertical* at most $2p$ times. The specific path between the two vertices depends on where we decide to do so. So, between two vertices there are at most n^{2p} paths. Considering that there are at most np starting vertices and np end vertices, we obtain the limit $O(n^{2p+2})$. □

From this lemma we obtain directly the complexity result for the algorithm:

Theorem 8.1. *A conjunctive peachy regular expression with variables of length m with p variables can be evaluated on a path of length n in time $O(m^\lambda n^{2p+2})$, with $\lambda \geqslant 1$ is a factor depending on the conversion from expression to finite automaton.*

8.2. Friendly expressions are not naughty

We now move on to consider a larger class of expressions: *friendly* expressions, that is, expressions that do not include any of the patterns of Definition 4.4. The study of peachy expressions carried out in the previous section has given us an important result: the structure of the graph entails that if two expressions ϕ_1 and ϕ_2 are not naughty, neither is their sequence $\phi_1\phi_2$. Therefore, if we prove that the "non-peachy" sub-expressions of a regular expression—those of the form ϕ^* with ϕ containing variables—are not naughty, the whole expression will also be non-naughty. In this section, therefore, we shall limit our analysis to expressions of the form ϕ^*. We shall not need to assume that ϕ is conjunctive.

Consider the intersection graph between a path and an automaton with actions. We are interested in those situations in which there are two distinct paths between vertices corresponding to the same state of the automaton:

$$\begin{tikzcd}
\begin{bmatrix} l \\ i \end{bmatrix} \arrow[r, rightsquigarrow, bend left, "\phi"] \arrow[r, rightsquigarrow, "\psi"'] & \begin{bmatrix} l \\ k \end{bmatrix}
\end{tikzcd} \tag{85}$$

Consider the vertex $\begin{bmatrix} l \\ i \end{bmatrix}$, where the paths diverge. Because of Lemma 7.4, the path must diverge in the presence of an ϵ-transition (which, it is worth remembering, always includes an action). The general situation for a splitting path is therefore the following:

$$\begin{tikzcd}
\begin{bmatrix} v \\ i \end{bmatrix} \arrow[dr, rightsquigarrow] & \\
& \begin{bmatrix} l \\ k \end{bmatrix} \\
\begin{bmatrix} u \\ i \end{bmatrix} \arrow[uu, "\epsilon"] \arrow[ur, rightsquigarrow] &
\end{tikzcd} \tag{86}$$

We are interested in a special type of configurations: those in which the final vertex represents the same state of the automaton as the initial one. The most general form of these configurations is

$$\begin{tikzcd}
& \begin{bmatrix} v \\ h \end{bmatrix} \arrow[ddr, rightsquigarrow, "\psi"] & \\
& \begin{bmatrix} u \\ h \end{bmatrix} \arrow[u, "\epsilon"] \arrow[dr, rightsquigarrow, "\phi"'] & \\
\begin{bmatrix} l \\ k \end{bmatrix} \arrow[ur, rightsquigarrow, "\zeta"] & & \begin{bmatrix} l \\ m \end{bmatrix}
\end{tikzcd} \tag{87}$$

I call this configuration a *kite*. Note that, from the point of view of the d-path, the expressions ϕ and ψ are two alternatives that match the same sub-path. If we superimpose to the kite the corresponding fragment in the d-path, we have

$$\pi_k \xrightarrow{\omega'} \pi_h \xrightarrow{\omega''} \pi_m$$

$$\begin{tikzcd}
& \begin{bmatrix} v \\ h \end{bmatrix} \arrow[ddr, rightsquigarrow, "\psi"] & \\
& \begin{bmatrix} u \\ h \end{bmatrix} \arrow[u, "\epsilon"] \arrow[dr, rightsquigarrow, "\phi"'] & \\
\begin{bmatrix} l \\ k \end{bmatrix} \arrow[ur, rightsquigarrow, "\zeta"] & & \begin{bmatrix} l \\ m \end{bmatrix}
\end{tikzcd} \tag{88}$$

where ω' is the string formed by the labels of the d-arcs that go from π_k to π_h, and ω'' that formed by the labels of the d-arcs that go from π_h to π_m. Note that $\omega'' \in L(\phi) \cap L(\psi)$, so in a kite $L(\phi) \cap L(\psi) \neq \emptyset$ always holds.

The kite begins and ends with the same state, therefore it corresponds to a portion of the input path that can be *pumped* in the sense of the pumping lemma [25]. Let $\pi = [\pi_1, \ldots, \pi_n]$ be a d-path, and suppose that, in the intersection graph, there is a kite that covers the sub-path $[\pi_p, \ldots, \pi_q]$. Then, as a consequence of the pumping lemma, every path of the form

$$[\pi_1, \ldots, \pi_{p-1}, \underbrace{\pi_p, \ldots, \pi_{q-1}, \pi_p, \ldots, \pi_{q-1}, \ldots, \pi_p, \ldots, \pi_{q-1},}_{k(q-p)} \pi_q, \ldots, \pi_n] \tag{89}$$

for $k \geqslant 0$ will be recognized by the automaton.

The kite is the building block of naughtiness, since it gives us two possible i-paths to go from a vertex $\begin{bmatrix} l \\ k \end{bmatrix}$ to a vertex $\begin{bmatrix} l \\ h \end{bmatrix}$. In an input graph like that of (89) this result in $\Theta(2^k) = \Theta(2^n)$ paths that we have to traverse.

For a problem to be naughty, however, it is not sufficient to have an intersection graph with an exponential number of paths; it is also necessary that each one of these paths induce a different value for the variables created during its traversal, so that any algorithm that returns all the possible values of the variables must traverse all the paths.

Moreover, the possibility of pumping the kite entails that the kites that correspond to actual sub-expressions must be *consistent*, that is, it must be possible to traverse an arbitrary number of them placed one after the other in such a way that the variables defined in it are always closed before being opened again.

Example 8. The following kite is not consistent:

$$\begin{tikzcd}
& \begin{bmatrix} v\\ h\end{bmatrix} \arrow[rdd, squiggly, "\psi"] & \\
& \begin{bmatrix} u\\ h\end{bmatrix} \arrow[u, "\mathfrak{B}x\langle\epsilon\rangle"] \arrow[rd, squiggly, "\rangle\mathfrak{E}x\langle\phi"'] & \\
\begin{bmatrix} l\\ k\end{bmatrix} \arrow[ru, squiggly, "\zeta"] & & \begin{bmatrix} l\\ m\end{bmatrix}
\end{tikzcd} \tag{90}$$

In fact, putting together two of them

$$\begin{tikzcd}
& \begin{bmatrix} v\\ h\end{bmatrix} \arrow[rdd, squiggly, "\psi"] & & \begin{bmatrix} v\\ h'\end{bmatrix} \arrow[rdd, squiggly, "\psi"] & \\
& \begin{bmatrix} u\\ h\end{bmatrix} \arrow[u, "\mathfrak{B}x\langle\epsilon\rangle"] \arrow[rd, squiggly, "\rangle\mathfrak{E}x\langle\phi"'] & & \begin{bmatrix} u\\ h'\end{bmatrix} \arrow[u, "\mathfrak{B}x\langle\epsilon\rangle"] \arrow[rd, squiggly, "\rangle\mathfrak{E}x\langle\phi"'] & \\
\begin{bmatrix} l\\ k\end{bmatrix} \arrow[ru, squiggly, "\zeta"] & & \begin{bmatrix} l\\ m\end{bmatrix} \arrow[ru, squiggly, "\zeta"] & & \begin{bmatrix} l\\ n\end{bmatrix}
\end{tikzcd} \tag{91}$$

there is, for every possible entry point (either $\left[\begin{smallmatrix} l\\ k\end{smallmatrix}\right]$ or $\left[\begin{smallmatrix} v\\ h\end{smallmatrix}\right]$) an exit point ($\left[\begin{smallmatrix} v\\ h'\end{smallmatrix}\right]$ or $\left[\begin{smallmatrix} l\\ n\end{smallmatrix}\right]$) for which there is at least one path on which the actions $\mathfrak{B}x$ and $\mathfrak{E}x$ are not balanced.

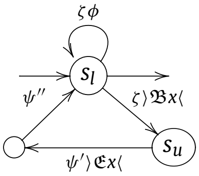
Finally, the ϵ-arc between $\left[\begin{smallmatrix} u\\ h\end{smallmatrix}\right]$ and $\left[\begin{smallmatrix} v\\ h\end{smallmatrix}\right]$ must contain an action, since otherwise it would have been removed by the reduction procedure. With these restrictions, we have only two significant valid templates for a kite that corresponds to a regular expression with variables:

$$\begin{tikzcd}
& \begin{bmatrix} v\\ h\end{bmatrix} \arrow[rdd, squiggly, "\psi'\rangle\mathfrak{E}x\langle\psi''"] & \\
& \begin{bmatrix} u\\ h\end{bmatrix} \arrow[u, "\mathfrak{B}x\langle\epsilon\rangle"] \arrow[rd, squiggly, "\phi"'] & \\
\begin{bmatrix} l\\ k\end{bmatrix} \arrow[ru, squiggly, "\zeta"] & & \begin{bmatrix} l\\ m\end{bmatrix}
\end{tikzcd} \tag{92}$$

and

$$\begin{tikzcd}
& \begin{bmatrix} v\\ h\end{bmatrix} \arrow[rdd, squiggly, "\psi'\rangle\mathfrak{B}x\langle\psi''"] & \\
& \begin{bmatrix} u\\ h\end{bmatrix} \arrow[u, "\mathfrak{E}x\langle\epsilon\rangle"] \arrow[rd, squiggly, "\phi"'] & \\
\begin{bmatrix} l\\ k\end{bmatrix} \arrow[ru, squiggly, "\zeta"] & & \begin{bmatrix} l\\ m\end{bmatrix}
\end{tikzcd} \tag{93}$$

The first corresponds to a fragment of an automaton

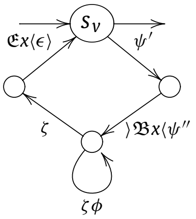

$\zeta\phi$
s_l
ψ''
$\zeta\rangle\mathfrak{B}x\langle$
s_u
$\psi'\rangle\mathfrak{E}x\langle$

(94)

(I have derived the non-deterministic automaton for the sake of clarity; we can of course apply the reduction procedure to this automaton, creating an equivalent one which is deterministic except for the presence of ϵ-transitions with actions.) This automaton corresponds to the expression

$$\big((\zeta\phi)^*\big(\zeta\big[x:\psi'\big]\big)^*\big)^* \tag{95}$$

with $L(\phi)\cap L(\psi'\psi'')\neq\emptyset$. The second kite corresponds to the automaton


s_v
$\mathfrak{E}x\langle\epsilon\rangle$
ψ'
ζ
$\rangle\mathfrak{B}x\langle\psi''$
$\zeta\phi$

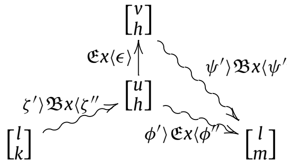
(96)

and to the expression

$$\psi'\big(\big[x:\psi''(\zeta\phi)^*\big]\zeta\big)^* \tag{97}$$

again with $L(\phi) \cap L(\psi'\psi'') \neq \emptyset$. Note that $L(\phi) \cap L(\psi'\psi'') \neq \emptyset$ if and only if $L(\zeta\phi) \cap L(\zeta\psi'\psi'') \neq \emptyset$, therefore the expressions (95) and (97) are the same as the forms i) and ii) of Definition 4.4.

Note also that, strictly speaking, there are two more possible forms of consistent kites, which can be obtained by placing an action of the suitable type in the middle of the expression ζ. However, these kites do not generate new forms of regular expressions. For instance, the kite


$\begin{bmatrix} v \\ h \end{bmatrix}$
$\mathfrak{E}x\langle\epsilon\rangle$
$\psi'\rangle\mathfrak{B}x\langle\psi''$
$\zeta'\rangle\mathfrak{B}x\langle\zeta''$
$\begin{bmatrix} u \\ h \end{bmatrix}$
$\begin{bmatrix} l \\ k \end{bmatrix}$
$\phi'\rangle\mathfrak{E}x\langle\phi''$
$\begin{bmatrix} l \\ m \end{bmatrix}$

(98)

corresponds to the expression

$$\left(\left(\zeta'[x:\zeta'']\right)^*\left(\zeta'[x:\phi']\phi''\right)^*\right)^* \tag{99}$$

which is a special case of (95) in which the expression $\zeta\phi$ contains the variable x as well. The same is true for the other form of consistent kite. The previous considerations prove the following:

Lemma 8.3. *Let ϕ be a regular expression. Then there is a path π such that the graph $\pi \times \phi$ has a kite if and only if ϕ is hard.*

Before deriving the complexity result of this section, we need to study some properties of a special class of binary trees.

Definition 8.1. A *colored tree* is a triple (T, C, λ), where $T = (V, E)$ is a binary tree with vertices V and edges $E \subseteq V \times V$, C is a finite set of *colors*, and $\lambda : V \to C$ is the function that associates a color to every vertex.

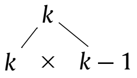
Definition 8.2. Let $\mathcal{T}$ be a colored tree. A triple of vertices $(n_1, n_2, n_3) \in V^3$ is called a *matching triangle* if:

i) n_2 and n_3 belong to a sub-tree with root n_1;
ii) n_2 and n_3 are at the same depth relative to n_1;
iii) $\lambda(n_1) = \lambda(n_2) = \lambda(n_3)$.

The following theorem limits the size of colored binary trees without matching triangles:

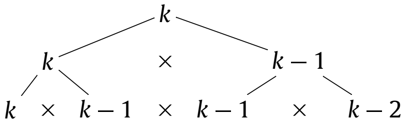
Theorem 8.2. *Let $\mathcal{T}$ be a complete colored binary tree that doesn't contain any matching triangle. If h is the height of the tree, then $h < |C|$.*

Proof. Set $k = |C|$ and consider the root of the tree with its two children. The root can be of any of the k possible colors. The two children can assume any of $k(k-1)$ combinations of two colors, all except the one in which both have the same color as the root, for that would create a matching triangle. Schematically



(100)

The same reasoning can be repeated at the following level, excluding the combinations that lead to matching triangles:


k
$k \quad \times \quad k-1$
$k \;\times\; k-1 \;\times\; k-1 \;\times\; k-2$

(101)

for a total of $k(k-1)^2(k-2)$ possible combinations of colors for the leaves. At height h the number of combinations is

$$N(h) = \prod_{q=0}^{h} (k-q)^{\binom{h}{q}} \tag{102}$$

For $h = k = |C|$ we have $N(h) = 0$, therefore if a complete tree has height $|C|$ there are no combinations of colors that keep the tree without matched triangles. □

We can now go back to the unfolding $\mathcal{U}(\Pi \times \phi)$ (Definition 7.2). In this tree, there are two kinds of combinations. The first, which we call *split* corresponds to a vertical arc:

$$\begin{tikzcd}
\begin{bmatrix} u \\ k \end{bmatrix} \arrow[d] & \begin{bmatrix} l \\ k \end{bmatrix} \arrow[l] \arrow[d] \\
\begin{bmatrix} v \\ k+1 \end{bmatrix} & \begin{bmatrix} p \\ k+1 \end{bmatrix}
\end{tikzcd} \tag{103}$$

and another, which we call a *stay*, corresponding to a cross arc

$$\begin{tikzcd}
\begin{bmatrix} l \\ k \end{bmatrix} \arrow[d] \\
\begin{bmatrix} p \\ k+1 \end{bmatrix}
\end{tikzcd} \tag{104}$$

To each unfolding of an intersection graph, we associate a colored binary tree as follows. If the automaton involved in the derivation of the intersection graph has p states, we choose the set of colors $C = \{1, \ldots, p\}$. We then transform each vertex $\begin{bmatrix} l \\ k \end{bmatrix}$ of $\mathcal{U}(\Pi \times \phi)$ into a vertex with color l. Each split is transformed into a vertex with two children:

$$\begin{tikzcd}
\begin{bmatrix} u \\ k \end{bmatrix} \arrow[d] & \begin{bmatrix} l \\ k \end{bmatrix} \arrow[l] \arrow[d] & & n_1 \arrow[dl, dash] \arrow[dr, dash] & \\
\begin{bmatrix} v \\ k+1 \end{bmatrix} & \begin{bmatrix} p \\ k+1 \end{bmatrix} & n_2 & & n_3
\end{tikzcd}
\qquad
\begin{aligned} \lambda(n_1) &= l \\ \lambda(n_2) &= v \\ \lambda(n_3) &= p \end{aligned} \tag{105}$$

and each stay to a vertex with one child

$$\begin{tikzcd}
\begin{bmatrix} l \\ k \end{bmatrix} \arrow[d] & n_1 \arrow[d, dash] \\
\begin{bmatrix} p \\ k+1 \end{bmatrix} & n_2
\end{tikzcd}
\qquad
\begin{aligned} \lambda(n_1) &= l \\ \lambda(n_2) &= p \end{aligned} \tag{106}$$

The number of vertices in $\mathcal{U}(\Pi \times \phi)$ is at most twice the number of vertices in the associated colored tree, so $\mathcal{U}(\Pi \times \phi)$ has $\Theta(2^n)$ vertices iff the colored tree has $\Theta(2^{n/2})$ vertices.

We are now ready to prove the main result of this section.

Theorem 8.3. *Let ϕ be a regular expression with variables. Then there is a family of paths $\Pi^{(n)}$ with $|\Pi^{(n)}| = n$ such that*

$$\left|\mathcal{U}\left(\Pi^{(n)} \times \phi\right)\right| = \Theta\left(2^n\right) \tag{107}$$

if and only if ϕ is hard.

Proof. Suppose first that ϕ is hard. Then, by Lemma 8.3 there is a path Π such that $\Pi \times \phi$ has a kite. Let $\Pi = \Pi'\hat{\Pi}\Pi''$m where $\hat{\Pi}$ is the portion of the graph that matches the kite as follows:

$$\overbrace{\pi_1 \xrightarrow{\ \omega_1\ }}^{\Pi'} \overbrace{\pi_k \xrightarrow{\qquad \omega_2 \qquad}}^{\hat{\Pi}} \overbrace{\pi_t \xrightarrow{\ \omega_3\ } \pi_n}^{\Pi''}$$

$$\begin{tikzcd}
 & \begin{bmatrix} v \\ h \end{bmatrix} \arrow[ddr, squiggly] & \\
 & \begin{bmatrix} u \\ h \end{bmatrix} \arrow[u] \arrow[dr, squiggly] & \\
\begin{bmatrix} l \\ k \end{bmatrix} \arrow[ur, squiggly] & & \begin{bmatrix} l \\ t \end{bmatrix}
\end{tikzcd} \tag{108}$$

By the pumping lemma, for every k, the path $\Pi'\hat{\Pi}^k\Pi''$ matches the expression and, since the length of Π' and Π'' is independent of k, it is $\lim_{n\to\infty} k/n = C \neq 0$. The graph that matches $\Pi'\hat{\Pi}^k\Pi''$ has k kites and therefore 2^k paths, so

$$\left|\mathcal{U}\left(\Pi^{(n)} \times \phi\right)\right| = \Theta\left(2^k\right) = \Theta\left(2^n\right) \tag{109}$$

Suppose now that there is a family of paths $\Pi^{(n)}$ such that

$$\left|T^{(n)}\right| = \left|\mathcal{U}\left(\Pi^{(n)} \times \phi\right)\right| = \Theta\left(2^n\right) \tag{110}$$

The tree $T^{(n)}$ has a matching path for each leaf, therefore $T^{(n)}$ has $\Theta(2^n)$ leaves. All its paths have length at most np. The tree has splits and stays, but the stays don't increase the number of leaves, so we can remove them obtaining an equivalent tree $T_1^{(n)}$ with only splits. $T^{(n)}$ is $\Theta(2^n)$ if and only if $T_1^{(n)}$. Let $\mathcal{T}^{(n)}$ be the colored tree associated to $T_1^{(n)}$. This tree has height at most np and $\Theta(2^n)$ leaves. For n sufficiently large, $\mathcal{T}^{(n)}$ must contain at least one matching triangle for if it didn't, because of Theorem 8.2, it would have at most 2^p leaves.

The triangle in $\mathcal{T}^{(n)}$ corresponds to the presence, in the tree $T^{(n)}$, of a structure

$$\begin{tikzcd}
& \begin{bmatrix} l \\ k \end{bmatrix} \arrow[d, squiggly] & \\
& \begin{bmatrix} u \\ h \end{bmatrix} \arrow[dl, squiggly] \arrow[dr, squiggly] & \\
\begin{bmatrix} l \\ m \end{bmatrix} & & \begin{bmatrix} l \\ m \end{bmatrix}
\end{tikzcd} \tag{111}$$

which is the unfolding of a kite. Therefore $\Pi^{(n)} \times \phi$ has, for n large enough, a kite. □

From this theorem and from Lemma 8.3 we derive

Corollary 8.1. *An expression ϕ is naughty on a family of paths if and only if it is hard.*

9. Relaxing some restrictions

The main result of the paper, Theorem 8.3, was derived under two hypotheses that limit its scope:

i) each variable can appear at most once in the expression;
ii) expressions are evaluated only on cold graphs.

In this section I shall consider these restrictions and see how, and to what extent, they can be relaxed.

9.1. Repeating variable names

So far, I have been considering only expressions in which each variable name appears only once. This was done mainly in order to arrive at a manageable definition of the semantics of the expressions, but the presence of this restriction could generate the doubt that lifting it would invalidate the results of the paper. In this section I shall present some informal arguments to show that, under a reasonable interpretation of the semantics of repeated names (given, basically, by the operational semantics induced by the automaton), this is not so.

Consider first the case in which the different instances of a variable name are separable, that is, they do not appear one within the scope of the other. Considering, for the sake of simplicity, two instances, this would amount to an expression

$$\phi = [x : \psi][x : \zeta] \tag{112}$$

In this case, a reasonable semantics for x would consist in the bindings of x in ψ followed by the bindings in ζ. For example, if the expression $c[x : a^*][x : b^*]c$ is applied to the graph

$$\pi_1 \xrightarrow{c} \pi_2 \xrightarrow{a} \pi_3 \xrightarrow{a} \pi_4 \xrightarrow{a} \pi_5 \xrightarrow{b} \pi_6 \xrightarrow{b} \pi_7 \xrightarrow{c} \pi_8 \tag{113}$$

one would expect an environment

$$u = \bigl(x \mapsto \bigl[[\pi_2, \pi_3, \pi_4, \pi_5], [\pi_5, \pi_6, \pi_7]\bigr]\bigr) \tag{114}$$

This is precisely the environment that would be created under the semantics of (6) so, even if I haven't explicitly said so, the model already covers this case.

If a variable name appears within the scope of another instance of the same name, the issue is a bit more complicated, and it is harder to find reasonable guidelines for the definition of a semantics. Consider the expression

$$c\bigl[x : a^*\bigl[x : b^*\bigr]c\bigr] \tag{115}$$

applied to the graph

$$\pi_1 \xrightarrow{c} \pi_2 \xrightarrow{a} \pi_3 \xrightarrow{a} \pi_4 \xrightarrow{b} \pi_5 \xrightarrow{b} \pi_6 \xrightarrow{c} \pi_7 \tag{116}$$

One reasonable semantics[4] would have the value of the innermost instance embedded in the outermost, leading, in this case, to the environment

$$u = \bigl(x \mapsto \bigl[\pi_2, \pi_3, \pi_4, [\pi_4, \pi_5, \pi_6], \pi_7\bigr]\bigr) \tag{117}$$

[4] Some semantics that one might find reasonable may be reduced to the case studied in this paper, e.g. by renaming one of the instances of x and then, at the end of the processing, concatenating the lists resulting for the two variables. These semantics are obviously covered by the expressions discussed in the paper, and I am not interested in them here.

This case is not contemplated by the semantics of Eq. (6), which in this case would return:

$$u = \bigl(x \mapsto [\pi_2, \pi_3, \pi_4, \pi_5, \pi_6, \pi_7]\bigr) \tag{118}$$

but it is easy to see that the environment (117) is exactly the value that would be returned by the automaton defined in this paper for that expression. Since the main results of this paper depend on the characteristics of the automaton, it follows that they can be extended to expressions with repeated variable names simply taking the operational semantics defined by the automaton.[5]

9.2. Warming up the graphs

The results of the paper hold only for graphs with a number of paths polynomial in the number of vertices. This class is not without interest, since it includes the most common applications of regular expressions: paths and trees, but it is interesting to see up to what point it can be relaxed.

The reason for introducing cold graphs is Lemma 4.1, which tells us that if an expression can be matched to a path in polynomial time, the same is true for the whole graph. The rest of the results depend on this lemma, so the point is whether we can relax the coldness assumption and still keep Lemma 4.1 valid.

In the worst case scenario the answer is not. If c is an expression that matches all symbols of the input alphabet Σ, then the expression $[x : c^*]$ will assign to x *all* the paths in the graph so, if a family of graphs is not cold, the results will be exponential in the number of vertices. There are, however, a number of cases in which the restriction can be relaxed. I shall still consider DAGs for the sake of coherence but, under the hypotheses of this section, regular expressions with variables can be applied to regular graphs as well, as long as they don't result in infinite result sets.

Consider first an expression that matches paths of limited length, that is, expressions ϕ that match paths of length at most k, where k is independent of the size of the graph.[6] In a DAG, the number of paths of length k is at most

$$\binom{n}{k+1} = \frac{n!}{(k+1)!(n-k-1)!} \leqslant \frac{n^{k+1}}{(k+1)!} \tag{119}$$

which, if k is independent of n, gives the desired polynomial bound on the number of matching paths, a condition strong enough to guarantee that Lemma 4.1 will hold.

A further relaxation can be obtained if, instead of a worst case scenario, we consider an average case analysis, that is, we consider the expected number of matching paths. The problem with the worst case expression $[x : c^*]$ is that c will match *all* the edges of the data graph. This is often not a realistic assumption. Rather, the condition on a transition of the automaton that implements the expression will match only a fraction of edges. Let the probability that an atomic expression match an edge in the data graph be $p < 1$. Then the probability that the expression will match a path of length k is p^k. This is not enough to make the problem tractable for general DAGs, since the expected number of matching paths is

$$\sum_{k=1}^{n} \binom{n}{k+1} p^k = \frac{1}{p}\bigl((1+p)^n - (1+np)\bigr) \tag{120}$$

It is however strong enough to make the problem tractable in cases of importance in applications, such as that of sparse graphs. If every vertex in the data graph has at most q outgoing edges, then in a general graph (not necessarily a DAG) the number of paths of length k that originate in a given vertex is at most q^k, and the number of paths of length k in the graph is at most nq^k. If $p < \frac{1}{q}$, then the expected number of matched paths is

$$\sum_{k=1}^{n} nq^k p^k = n\frac{1-(qp)^{n+1}}{1-qp} \leqslant \frac{n}{1-qp} \tag{121}$$

In this case, Lemma 4.1 will hold in an average case sense and, still on average, the results of the paper will continue to hold.

10. Related work

Regular expressions, and languages closely related to them, are an enormously popular formalism, a component of many tools with which all programmers are familiar, such as the scripting languages PERL [26], and Python [27], Unix utilities such as grep [28] and lex [29] as well as text editors such as *vi* and *emacs*. Regular expressions are a well-studied and

[5] It should, of course, not be surprising that the semantics of (6) and the automaton that corresponds to it give different results for the same expression: the equivalence between the two was proved only within the field of validity of the semantics, namely expressions with no repeated variable names, and there is no reason why it should continue to hold beyond that field of validity.

[6] Expressions without stars trivially satisfy this criterion, but the property is not limited to them. In many applications, the semantics of the data guarantees that an expression $\psi\phi\zeta$, where ψ and ζ are without stars match paths of limited length.

well-understood formalism, and the theoretical instruments for dealing with them, as well as efficient algorithms for processing them, have been known for quite some time [30,31], although practical implementations seem to have taken little notice [32]. Theoretical work on extensions of regular expressions, in spite of their common application, is more scarce.

10.1. Backreferencing

A common extension to regular expression, very much related to the work in this paper, is *backreferencing*.[7] This class of expressions, sometimes called *rewbr* [14], introduce an assignment operator similar to the one introduced here, and a variable *mentioning* operation that must come after the corresponding assignment operator and that matches an expression equal to the value assigned to the variable. It is assumed that the input alphabet and the set of variable symbols are disjoint. So, the expression

$$\psi \equiv [x:\phi]x \tag{122}$$

matches inputs composed of the juxtaposition of two copies of the *same* string, where each copy matches ϕ.[8] Note that the expression is different from $\phi\phi$: in the latter, any two strings, equal or not, that match ϕ will match the complete expression, while in the case of ψ the two parts of the input must coincide. That is, the language recognized by the expression is

$$L(\psi) = \{\omega\omega \mid \omega \models \phi\} \tag{123}$$

This language is not regular, proving that regular expressions with backreferencing are more expressive than simple regular expressions. Not only are regular expressions with backreferencing more expressive, their expressive power increases with the level of variable nesting that they allow [33]. The general problem of matching regular expressions with backreferencing is known to be NP-complete [14].

The definition of backreferencing contains semantic ambiguities that have been lamented by several authors, in particular in relation to the alternation operator. The behavior of an expression such as

$$\big([x:\phi] + \psi\big)^* x \tag{124}$$

has been shown to vary depending on the implementation when the last substring that matches $[x:\phi] + \psi$ does not match ϕ. For example, in some implementations, the expression $([x:a] + b)^*cx$ will match the string *aabca*. Note that according to a reasonable interpretation, the last match of the expression $[x:a] + b$ matches a b, so the value of x should be set to the empty string, making it impossible for it to match the a that follows the c. Many theoretical papers on backreferencing such as [14] and [35] do not define this case accurately. (The problem was pointed out in [34].) Problems have also been found for expressions such as $([x:\phi^*]x)^*$, so that many authors limit their analysis to what I have called *peachy* expressions. In many cases, backreferencing doesn't use a strict semantics of the type that I have used in this paper, but a *last iteration* semantics. If the definition of a variable is under a Kleene star, and the mention of the variable is not, then the mention matches the value of the variable given in the last iteration[9] [36]. This assumption simplifies processing considerably since, as Lemma 1 of [34] proves, with this semantics every expression is equivalent to a peachy one. In spite of this simplification, backreferencing is inherently harder than simply collecting the value of variables as we are doing here: the reduction from *graph cover* that [14] uses to prove NP-completeness results in an expression that has no variable definitions under the star, that is, it is once again a peachy expression.

It must be noted, however, that although backreferencing includes a variable definition operator similar to the one introduced here, the two approaches are conceptually different. Backreferencing is used to extend the expressive power of regular expressions as language recognizers. In this paper we considered regular expressions as query languages and, without altering their expressive power, we are interested in returning results from the query. On the one hand, this entails that we can't rely on simplified semantics such as *last iteration*; on the other hand, it means that, as we have seen, expressions that are intractable in the context of backreferencing become tractable in our case.

10.2. Other work

Backreferencing is certainly the best known extension to regular expressions, but it is by no means the only one. In order to solve the semantic ambiguities of backreferencing, Ref. [37] develops *pattern expressions*, a hierarchy of expressions with variables, in which expression i can only mention variables assigned in expressions $0, \ldots, i-1$. In [36] a model is proposed of sub-expression addressing similar in spirit to what was studied here, but with a semantics that is, *mutatis*

[7] As a matter of fact, all the implementations of regular expressions that I have mentioned in the opening of this section include backreferencing.

[8] The syntax of backreferencing varies among different authors. In many cases, the assignment operator is a postfix %, so that the previous expression would be written as $(\phi)\%xx$. This is the syntax used in [14] and [33]. Other papers, such as [34], as well as most implementations, dispense with variable names altogether and introduce a backreferencing operator $\backslash i$, $i \in \mathbb{N} - \{0\}$ that matches whatever string was matched by the sub-expression contained in the ith pair of parentheses (pairs of parentheses are counted by the order of their left parenthesis). So, the previous expression would be written as $(\phi)\backslash 1$.
All these syntactic differences are clearly marginal. I have decided, for the sake of consistency, to use the same assignment syntax used in this paper.

[9] Note however that the previous observations about the alternation operator show that many implementations fail to be consistent with this semantics.

mutandis, that of backreferencing, a semantics that, in the context of sub-expression matching, has been subject to some criticism [38].

The automata that we use here for processing the expressions are also related to automata used for other extensions of regular expressions. In particular, our automata can be seen as a specialization of the *finite automata with semantic actions* [39], but avoid the semantic ambiguity and the problems with non-peachy expressions that have been pointed out in [36].

Finally, there are languages that embed some subset of regular expressions in a more general procedural language. The best known such languages are probably *xquery* and the pattern language *xpath 2.0*, defined by the world wide web consortium [9]. Xpath 2.0, as a pattern language, has a limited expressivity, although there are extensions with first order [6] and monadic second order logic [11] expressivity. Xpath 2.0 does indeed allow the use of variables, but they can only be used in the procedural operators *for*, *some*, and *every*. They are not, properly speaking, part of the pattern matching portion of the language, and they are conceptually and semantically different from the variables that I am using here.

11. Conclusions

This paper has presented some considerations on the introduction of variables in regular expressions. I have argued that without variables, on graphs, matching a regular expression may be insufficiently informative, especially in data base application, as it tells us that (at least) a matching path exists between two nodes, but it doesn't tell us which and how many. Variables can solve this problem by returning all the paths that match any sub-expression of a given regular expression.

The paper has shown that answering queries with variables is inherently harder than answering queries without them, essentially because, even with relatively simple expressions on paths, the result set can become unmanageably large. I have presented a modification of the graph traversal algorithm that, while traversing the graph, detects the paths that satisfy the expressions, and builds environments in which proper values are assigned to the variables. I have also shown that there is a class of expressions (called *hard*) in which one of two patterns appear as a sub-expression, which are inherently hard to process, while for each expression not including sub-expressions if the given forms, there is a polynomial-time algorithm.

References

[1] P. Buneman, M. Fernandez, D. Suciu, UnQL: a query language and algebra for semistructured data based on structural recursion, VLDB J. 9 (2000) 76–110.

[2] M. Consens, A. Mendelzon, GraphLog: a visual formalism for real life recursion, in: Proceedings of the Ninth SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 1990, pp. 404–416.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, The Lorel language for semistructured data, Int. J. Digit. Libr. 1 (1) (1997) 68–88.

[4] L. Sheng, Z. M. Özsoyoğlu, G. Özsoyoğlu, A graph query language and its query processing, in: Proceedings of the 15th International Conference on Data Engineering, 1999.

[5] F. Neven, T. Schwentick, Automata- and logic-based pattern languages for tree-structured data, in: Semantics in Databases, in: Lecture Notes in Comput. Sci., vol. 2582, Springer-Verlag, Heidelberg, 2001, pp. 160–178.

[6] M. Marx, Conditional xpath, ACM Trans. Database Syst. 30 (4) (2005) 929–959.

[7] S. Santini, A. Gupta, A calculus and algebra for querying directed acyclic graphs, in: J. Riquelme, P. Botella (Eds.), XV Journadas de Ingeniería del software y bases de datos, CIMNE, Barcelona, 2006.

[8] S. Flesca, F. Furfaro, S. Greco, Weighted path queries on semistructured databases, Inform. and Comput. 204 (5) (2006) 679–696.

[9] J. Clark, S. DeRose, Xml path language (xpath) vers. 2.0, Recommendation, World Wide Web Consortium, on-line, 2007.

[10] M. Marx, First order paths in ordered trees, in: T. Eiter, L. Libkin (Eds.), Proceedings of the International Conference on Database Theory, in: Lecture Notes in Comput. Sci., vol. 3363, Springer-Verlag, Heidelberg, 2005, pp. 114–128.

[11] B.T. Cate, The expressivity of xpath with transitive closure, in: Proceedings of the SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM Press, 2006, pp. 328–337.

[12] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Vardi, Reasoning on regular path queries, SIGMOD Rec. 32 (4) (2003) 83–92.

[13] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Vardi, Query processing using view for regular path queries with inverse, in: Proceedings of the SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2000, pp. 58–66.

[14] A.V. Aho, Algorithms for finding patterns in strings, in: Handbook of Theoretical Computer Science, vol. A: Algorithms and Complexity, Elsevier/MIT Press, 1990.

[15] A. Mendelzon, P. Wood, Finding regular simple paths in graph databases, in: Proceedings of the International Conference on Very Large Data Bases, 1989, pp. 185–93.

[16] R.D. Tennent, Semantics of Programming Languages, Prentice Hall, Englewood Cliffs, 1991.

[17] H.B. Hunt, D.J. Rosenkrantz, T.G. Szymanski, On the equivalence, containment and covering problems for the regular and context-free languages, J. Comput. System Sci. 12 (1976) 222–268.

[18] F. Neven, Automata, logic, and xml, in: Computer Science Logic: 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, in: Lecture Notes in Comput. Sci., vol. 2471, Springer-Verlag, Heidelberg, 2002.

[19] F. Neven, T. Schwentick, On the power of tree-walking automata, Inform. and Comput. 183 (1) (2003) 86–103.

[20] M. Nivat, A. Podelski, Minimal ascending and descending tree automata, SIAM J. Comput. 26 (1) (1997) 39–58.

[21] E.A. Emerson, C.S. Jutla, Tree automata, mu-calculus, and indeterminacy, in: 32nd Annual Symposium of Foundations of Computer Science, 1991.

[22] A.O. Mendelzon, P.T. Wood, Finding regular simple paths in graph database, SIAM J. Comput. 24 (6) (1995) 1235–1258.

[23] P. Beame, P. Borodin, A. Raghavan, W.L. Ruzzo, M. Tompa, Time-space tradeoffs for undirected graph traversal by graph automata, Inform. and Comput. 130 (2) (1996) 101–129.

[24] T. Kamimura, G. Slutzki, Parallel and two-way automata on directed ordered acyclic graphs, Inf. Control 49 (1) (1981) 10–51.

[25] J.G. Brookshear, Theory of Computation: Formal Languages, Automata, and Complexity, Addison–Wesley, 1989.

[26] N. Chapman, PERL-The Programmer's Companion, Wiley, Chichester, 1997.

[27] M. Lutz, Programming Python, O'Reilly, Cambridge, 2011.

[28] T. Abou-Assaleh, W. Ai, Survey of global regular expression print (grep) tools, unpublished manuscript available on-line, 2004.

[29] M.E. Lesk, Lex – a lexical analyzer generator, Computer science technical report, AT&T Bell Laboratories, Reading, MA, 1975.

[30] M. Rabin, D. Scott, Finite automata and their decision problems, IBM J. Res. Develop. 3 (1959) 114–125.

[31] K. Thompson, Regular expressions search algorithm, Commun. ACM 11 (6) (1968) 419–422.

[32] R. Cox, Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, . . .), unpublished manuscript available on-line, 2007.

[33] K.S. Larsen, Regular expressions with nested levels of back referencing form a hierarchy, Inform. Process. Lett. 65 (1998) 169–172.

[34] C. Câmpeanu, N. Santean, Addressing an open problem on regex, Technical report 10, University of Waterloo, 2007.

[35] C. Câmpeanu, K. Salomaa, S. Yu, A formal study of practical regular expressions, Internat. J. Found. Comput. Sci. 14 (6) (2003) 1007–1018.

[36] V. Laurikari, Efficient submatch addressing for regular expressions, Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology, 2001.

[37] C. Câmpeanu, S. Yu, Pattern expressions and pattern automata, Inform. Process. Lett. 92 (2004) 267–274.

[38] C.L.A. Clarke, G.V. Cormack, On the use of regular expressions for searching text, ACM Trans. Program. Lang. Syst. 19 (3) (1997) 413–426.

[39] I. Nakata, M. Sassa, Regular expressions with semantic rules and their applications to data structure directed programs, Adv. Softw. Sci. Technol. 3 (1991) 93–108.